Universitat de les Illes Balears

Escola Politècnica Superior

Memòria del Treball de Fi de Grau

Sistema de realitat augmentada per a maquetació d'exteriors i planificació urbana

Miquel Àngel Ramis Llompart

Grau d'Enginyeria Informàtica

Any acadèmic 2019-20

DNI de l'alumne: 78221417G

Treball tutelat per Miquel Mascaró Portells
Departament de Matemàtiques i Informàtica

S'autoritza la Universitat a incloure aquest treball en el Repositori Institucional per a la seva consulta en accés obert i difusió en línia, amb finalitats exclusivament acadèmiques i d'investigació	Autor		Tutor	
	Sí	No	Sí	No
	X		X	

Paraules clau del treball:
realitat augmentada, Android, dispositius mòbils, sensors, GPS, giroscopi, rotacions, quaternions

Universitat de les Illes Balears

escola politècnica superior

Trabajo Fin de Grado

Sistema de realidad aumentada para maquetación de exteriores y planificación urbana

MIQUEL ÀNGEL RAMIS LLOMPART

Tutor

MIQUEL MASCARÓ PORTELLS

Escuela Politécnica Superior
Universitat de les Illes Balears
Palma, 01 de julio de 2019

Querría agradecer, primero de todo, a toda mi familia por apoyarme en todos estos últimos años de carrera tanto emocionalmente como económicamente que han permitido que pueda llegar hasta el final de este grado con los conocimientos necesarios para realizar este proyecto y acabar de formarme como Ingeniero Informático

También quiero agradecer a Miquel Mascaró Portells por permitirme realizar mi propuesta de TFG de realidad aumentada junto a él y por dejarme, por motivos de trabajo, aplazar la entrega de este proyecto.

Finalmente quiero agradecer a todos los profesores de esta carrera, que me han formado y han reconocido mis esfuerzos durante todos estos años.

Índice de Contenido

Índice de Figuras

Índice de Tablas

Índice de Ecuaciones

Acrónimos

A

API	Application Programming Interface
APK	Android Application Package
APP	Aplicación

B

BDD	Base de Datos

G

GPS	Global Positioning System
GSM	Global System for Mobile

M

MVP	Modelo Vista Presentador

R

RA	Realidad Aumentada
RAM	Random Access Memory

T

TFG	Trabajo Fin de Grado

U

UIB	Universitat de les Illes Balears

W

Wifi	Wireless Fidelity

Glosario

A

Acelerómetro: sensor que obtiene la aceleración aplicada en el dispositivo en cada una de sus tres direcciones en tiempo real.

Actividad: componente de la aplicación que contiene una pantalla con la que los usuarios pueden interactuar para realizar una acción.

Android: sistema operativo móvil desarrollado por Google, basado en el Kernel de Linux y otros software de código abierto.

Android Application Package: un archivo con extensión .apk que contiene una aplicación para el sistema operativo Android.

Android Manifest: necesario para ejecutar el código, es un archivo de configuración donde podemos aplicar las configuraciones básicas de nuestra aplicación y que proporciona información esencial sobre esta al sistema Android.

Android Studio: entorno de desarrollo integrado oficial para la plataforma Android.

Ángulo de Visión: ángulo que determina la parte de la escena que es captada por una lente.

Ángulos de Euler: conjunto de tres coordenadas angulares que sirven para especificar la orientación de un sistema de referencia de ejes ortogonales, normalmente móvil, respecto a otro sistema de referencia de ejes ortogonales normalmente fijos.

B

Base de Datos: conjunto de datos pertenecientes a un mismo contexto y almacenados sistemáticamente para su posterior uso.

Bitmap: clase encargada de cargar una imagen de la memoria interna a la memoria RAM del dispositivo. Esta clase es la más usada para operar y usar imágenes en Android e implementa múltiples funcionalidades para el uso y dibujado de imágenes.

Bloqueo de Cardán: pérdida de un grado de libertad en una suspensión cardán de tres rotores, que ocurre cuando los ejes de dos de los tres rotores se colocan en paralelo, bloqueando el sistema en una rotación en un espacio bidimensional degenerado.

Búfer: es un espacio de memoria, en el que se almacenan datos de manera temporal.

C

C: lenguaje de programación compilado de propósito general.

Caché: software o estructura que almacena datos para que las solicitudes futuras de estos datos puedan atenderse con mayor rapidez.

Callback: técnica por la cual, se le ofrece al cliente que está en cola de espera la posibilidad de devolverle la llamada en un tiempo determinado.

Campo de Visión: es el espacio que abarca la visión de una lente con su ángulo de visión.

Controlador de Virtualización: nombre dado en la memoria a la clase *VirtualCameraView* del Módulo de Realidad Aumentada. Es la clase encargada de iniciar y apagar la realidad aumentada en el dispositivo.

Cuaternión: extensión de los números reales, similar a la de los números complejos.

D

Datos del Usuario: nombre dado a la clase UserData encargada de almacenar toda la información y características del dispositivo o del usuario.

Distancia Angular: distancia del arco que une dos puntos en una circunferencia, medida en grados o radianes.

Distancia Relativa: longitud del segmento de recta concebido entre dos localizaciones.

E

Entorno: alrededores o espacio circundante a una localización. Normalmente hace referencia al espacio cercano al usuario.

G

Garbage Collector: programa que funciona en segundo plano cuya función es liberar la memoria no usada o huérfana de otros programas. Usado en aplicaciones en Java, sirve para limpiar la memoria no usada del dispositivo.

Geolocalización: capacidad para obtener la ubicación geográfica real de un objeto.

Gestor de Refresco: nombre que recibe la clase *GLRenderer* en la memoria. Es la clase encargada de dibujar por pantalla todas las Instancias de Imagen cercanas al dispositivo.

Giroscopio: sensor de orientación capaz de calcular la rotación actual del dispositivo.

Global Positioning System: sistema que permite determinar en toda la Tierra la posición de cualquier objeto.

I

Instancia de Imagen: instancia de imagen es el nombre dado a una instancia de la clase PictureObject en este documento. Una instancia de esta clase, se corresponde a una imagen situada por un usuario, en una localización con una rotación determinada.

iOS: Sistema Operativo móvil desarrollado para los dispositivos móviles de la multinacional Apple Inc.

J

Java: lenguaje de programación interpretado, de propósito general, concurrente y orientado a objetos.

L

Layout: estructura de XML usada para ordenar vistas en la pantalla del dispositivo.

Listener: clase en Java que se encargan de controlar eventos. Un evento consiste en una acción realizada por el usuario o por uno de los sensores. Los Listener esperan a que el evento se produzca y realiza una serie de acciones. Según el evento, necesitaremos un Listener específico para que lo controle. Cada tipo de Listener tiene

una serie de métodos que debemos implementar obligatoriamente, aunque solo queramos usar uno solo de ellos.

Localización: posición de un objeto o imagen en el eje de coordenadas real formado por la latitud, la longitud y la altitud.

M

Magnetómetro: sensor capaz de calcular el campo magnético en cada uno de los ejes del dispositivo en tiempo real.

Maquetación: conjunto de una o más imágenes en un lugar.

Matriz de Rotación: matriz que representa una rotación en el espacio euclídeo.

Matriz de Visualización: la matriz resultante de multiplicar la matriz de Proyección en Perspectiva por la matriz de rotación del dispositivo.

Modelo Vista Presentador: patrón de arquitectura de software, que separa los datos y la lógica de negocio de una aplicación de la interfaz de usuario y el apartado encargado de gestionar los eventos y las comunicaciones con el usuario.

Modelo de Datos: parte de la aplicación que posee toda la lógica del guardado de información y datos.

Módulo: parte de una aplicación cuya lógica esta oculta al resto de la aplicación. Mientras un módulo implemente los mismos métodos externos o públicos, el cambiar la lógica de este, no afecta al funcionamiento del resto de la aplicación.

Módulo de Cámara: módulo encargado de mostrar por pantalla las imágenes captadas por el sensor de imagen de forma periódica en la aplicación.

Módulo de Localización: módulo de la aplicación encargado de obtener localizaciones de forma periódica de la manera más precisa posible.

Módulo de Rotación: módulo de la aplicación encargado de obtener y normalizar la rotación obtenida se los sensores de rotación del dispositivo.

Módulo de Realidad Aumentada: módulo de la aplicación encargada de dibujar por pantalla todas las imágenes cercanas al usuario.

Monitorizar: capacidad de una parte de la aplicación de por sí solo, mediante peticiones periódicas, obtener y/o actualizar una información determinada.

O

OpenGL: especificación estándar que define una API multilenguaje multiplataforma para escribir aplicaciones que produzcan gráficos 2D y 3D.

OpenGL ES: variante simplificada de la API gráfica OpenGL diseñada para dispositivos integrados tales como teléfonos móviles.

P

Path: señala la localización exacta de un archivo o directorio mediante una cadena de caracteres concreta.

Permiso: en Android, un permiso consiste en un consentimiento, aceptado por el usuario, para que una aplicación pueda realizar un tipo de tareas que pueden poner en riesgo al usuario, otras aplicaciones o el sistema operativo.

Presentador: parte de una aplicación que usa el patrón MVP usada como intermediario entre la Vista y el Modelo de Datos de la aplicación. Esta parte también implementa toda la lógica no visual o gráfica de la aplicación.

Programa (OpenGL ES): clase interna usada por la clase *GLES20* en Java para especificar que transformaciones van a aplicarse a los vértices y a la textura antes de pintarse por pantalla.

Proveedor de Rotación: nombre de la clase *OrientationSensorProvider* de la aplicación en el proyecto. Esta clase se encarga de monitorizar y normalizar la rotación del dispositivo y guardarla en el Modelo de Datos.

Proyección en Perspectiva: tipo de proyección virtual de OpenGL usado en la aplicación para simular la visión de una lente en el entorno virtual.

Punto de visión: posición en OpenGL de la cámara virtual. En OpenGL ES esta posición es estática y se sitúa en el origen de coordenadas.

R

Realidad aumentada: visualización de parte del mundo real a través de un dispositivo tecnológico con información gráfica añadadida pos este dispositivo.

RelativeLayout: tipo de layout que permite ordenar las vistas tanto en profundidad, como por posición dentro de la pantalla.

Redes neuronales: sistema compuesto de múltiples nodos entrelazados. Dada una entrada de datos, devuelvan una salida o solución. Este sistema o estructura requiere una preparación o entrenamiento previo para poder dar la solución o salida deseada.

Resolución: densidad de pixeles de una vista o la pantalla de un dispositivo.

S

Scrum: metodología ágil de planificación de proyectos para el desarrollo de software.

Sensor: dispositivo o componente electrónico que informa del estado de una medida o métrica del entorno en tiempo real, ya sea la aceleración o la temperatura.

Sensor de Imagen: sensor óptico capaz de obtener mediante una lente imágenes del entorno real que observa a tiempo real él dispositivo.

Sensor de Rotación: conjunto de los sensores de giroscopio, acelerómetro y magnetómetro para el cómputo de la rotación del dispositivo.

Servidor: aplicación en ejecución capaz de atender las peticiones de un cliente y devolverle una respuesta en concordancia.

Shader de Vértices: subprograma en C que se enlaza a un Programa donde se definen las transformaciones aplicadas a los vértices de los polígonos dibujados usando el Programa.

Shader de Fragmentos: subprograma en C que se enlaza a un Programa donde se definen las transformaciones que se aplica al color de cada píxel dentro de un polígono antes de dibujarse por pantalla.

Sistema Integrado: sistema de control dedicado a la realización de tareas específicas. Es un sistema optimizado a ejecutar tareas específicas usando recursos limitados.

Sprint: termino usado en Scrum para referirse a un intervalo prefijado de tiempo durante el cual se crea un incremento de producto utilizable y potencialmente entregable.

SQLite: sistema de gestión de bases de datos relacionales para Java y otros lenguajes de programación.

T

Tabla de Hash: una estructura de datos que asocia *llaves* o *claves* con *valores*. Permite el acceso a los elementos o valores almacenados a partir de una clave generada por una función de hash. Una función de hash transforma las llaves asociadas a la posición donde se guarda el elemento.

Tarjeta SD: dispositivo de memoria externa con forma de tarjeta capaz de conectarse a otros dispositivos tanto para leer como para escribir datos.

Thread: agrupación de un trozo de programa junto con el conjunto de registros del procesador que utiliza y una pila de máquina. Esto permite a un programa ramificarse, de tal forma que dos o más procesos se estén ejecutando al mismo tiempo.

Textura: array de datos de una imagen en memoria usada en OpenGL para mapearla sobre un polígono.

V

Vista: bloque básico para los componentes de la interfaz de una aplicación en Android. Consiste en una representación de una pantalla rectangular, que permite dentro de esta la interacción del usuario con la aplicación.

Vista (MVP): parte de una aplicación que usa el patrón MVP que implementa toda la lógica relacionada con la visualización e interacción por pantalla de imágenes o elementos gráficos como botones o texto.

Vector director: es un vector que da la dirección de una recta y también la orienta, es decir, le da un sentido determinado.

Vértice: punto donde se encuentran dos o más elementos unidimensionales.

X

XML: es un metalenguaje (un lenguaje de programación que se utiliza para decir algo acerca de otro) extensible de etiquetas adaptado de SGML.

Resumen

El objetivo de este proyecto, es la creación de una app experimental para dispositivos Android que permita maquetar el entorno del usuario utilizando un sistema de Realidad aumentada (RA). Una app o aplicación es un programa informático diseñado como herramienta para permitir a un usuario realizar uno o diversos tipos de tareas. Esta aplicación va dirigida a la planificación urbana de exteriores, por ejemplo, para poder comprobar el impacto visual de añadir un nuevo elemento decorativo a un parque público. La aplicación tiene que permitir al usuario, añadir y eliminar imágenes alrededor del usuario, creando así una maquetación para ese entorno.

Esta maquetación tiene que realizarse en exteriores, permitiendo al usuario guardar imágenes en su proximidad que puedan ser visualizadas en tiempo real en la pantalla del dispositivo. También, la aplicación tiene que ser capaz de guardar esta maquetación del entorno en la localización donde se ha realizado. De tal forma que un usuario, al volver a esta localización, pueda volver a visualizar esta maquetación.

Este proyecto no incluye la implementación de un servidor para compartir las maquetaciones entre distintos dispositivos, debido al ya amplio volumen de trabajo de este TFG. De esta manera, las maquetaciones realizadas por un usuario solo se guardarán en su propio dispositivo. El objetivo de este proyecto es crear una app con estas capacidades:

1. Capacidad de almacenamiento de las imágenes junto a su localización.
2. Capacidad de localizar el dispositivo, para detectar automáticamente la localización de las imágenes respecto a este.
3. Capacidad de interconectar los sensores de orientación del dispositivo junto a la cámara y un motor gráfico para la creación de un sistema de RA en tiempo real.
4. Capacidad de interacción del usuario con la aplicación.

La app para este proyecto ha sido creada con Android Studio usando Java como único lenguaje de programación junto a XML para la parte de maquetación de los componentes interactivos de la aplicación.

Esta app usa la geolocalización vía GPS como método principal de localización del dispositivo, debido a la necesidad de poder obtener una localización en exteriores de manera automática. Para detectar la orientación del dispositivo se ha utilizado una combinación de diferentes sensores basada en una modificación propia de la implementación de código de libre uso por Alexander Pacha.

Para el apartado visual de la aplicación se ha utilizado una combinación del sensor de la cámara junto a OpenGL ES, para la correcta visualización de las imágenes virtuales en el dispositivo.

Finalmente se ha creado el sistema de guardado de imágenes en memoria utilizando SQLite. La estructura de la aplicación usa un diseño MVP, capaz de en un futuro o en próximas versiones de la aplicación, poder compartir las maquetaciones del entorno entre múltiples dispositivos usando un servidor.

La app funciona en dispositivos Android que cumplen estos requisitos. No se asegura el correcto funcionamiento en todos los dispositivos que los cumplan, debido a la inmensa cantidad de dispositivos Android que existen actualmente en el mercado con diferentes especificaciones.

Capítulo 1: Introducción al proyecto

Justificación del proyecto

La principal finalidad de este proyecto es la investigación de la RA en dispositivos móviles y en la capacidad actual de la tecnología para cumplir este propósito. Actualmente los dispositivos electrónicos más usados son los móviles y es necesario aprender a programar y a utilizar todas las características en este tipo de dispositivos. Se requiere investigar qué resultados se pueden lograr aplicando la realidad aumentada en dispositivos móviles y que impedimentos pueden surgir en este tipo de aplicaciones. Esta aplicación tiene como foco principal, la maquetación de exteriores para delimitar el ámbito de ejecución de la práctica a un uso real y práctico de la RA.

Ámbito del proyecto

Este proyecto está dedicado exclusivamente al desarrollo de una aplicación capaz de poner en práctica las capacidades adquiridas tanto de aprendizaje como de adaptación por parte del alumno para la creación de software en dispositivos móviles. Para realizar este proyecto han sido necesarios el estudio de programación en Android, el entendimiento de la funcionalidad y uso de cada uno de los sensores usados, teoría de algebra sobre matrices y cuaterniones y finalmente, el uso de librerías gráficas como OpenGL en sistemas integrados. Pero, la cualidad más importante que requiere este proyecto, es la capacidad que demuestra el alumno de aplicar y combinar todos estos conceptos en una sola aplicación para que funcionen de manera conjunta suplementándose entre ellos.

Retos del proyecto

Retos del proyecto	**Razones**
Aprender a programar en Android	La programación en dispositivos Android no es una competencia que se adquiera en el itinerario de computación de la UIB y mucho menos, con la habilidad necesaria para la realización de este proyecto
Aprendizaje del funcionamiento de múltiples sensores en dispositivos Android	Para la creación de esta práctica ha sido necesario el aprendizaje del funcionamiento y manejo de tipos distintos de sensores
Aprendizaje del motor gráfico de OpenGL ES	A nivel de programador, esta librería gráfica se diferencia mucho de la librería basada en OpenGL aprendida para C en la asignatura de Informática Gráfica cursada en el itinerario de computación de la UIB
Aprendizaje y uso de cuaterniones para la rotación de objetos en espacios tridimensionales	Este conocimiento no es una competencia adquirida en el itinerario de computación de la UIB

Gestión de memoria y recursos utilizando múltiples estructuras de datos	Al estar programando en dispositivos móviles, es necesaria la creación de estructuras de caché, así como un sistema de cargado y eliminado automático de datos en memoria, debido a los pocos recursos disponibles.
Unión de todos estos conceptos para la creación de una app de RA	El apartado más difícil de esta práctica es la unión de todas las dificultades mencionadas para que funcionen armónicamente entre ellas, debido a la gran cantidad de procesos en segundo plano, la comunicación necesaria entre ellas y los datos usados por cada una por separado

Tabla 1. Tabla de Retos del TFG y Razones.

Alcance del proyecto

El objetivo principal del proyecto es la creación de una aplicación Android capaz de añadir y eliminar imágenes en el entorno para la maquetación urbana de zonas exteriores.

Los requisitos básicos de esta aplicación son:

1. Crear una aplicación en un dispositivo Android utilizando los sensores de este para crear una visión de realidad aumentada en el dispositivo, capaz de poder ver las imágenes insertadas cerca del dispositivo en la pantalla de este.
2. Tiene que ser capaz de poder almacenar automáticamente estas imágenes en la memoria del dispositivo.
3. Las imágenes maquetadas tienen que ser guardadas en la posición real donde se han insertado, de tal manera que se pueda volver a ese entorno para volver a visualizarlas.
4. El usuario tiene que ser capaz de interactuar con la aplicación para añadir/eliminar/visualizar las imágenes en tiempo real.
5. La entrega de este documento, junto a un apartado de manual de usuario, para el correcto uso de esta aplicación.

Los requisitos no incluidos en la práctica son:

1. La capacidad de funcionar en múltiples dispositivos móviles. Aunque esta app se ha creado para que pueda funcionar en otros dispositivos, no se asegura un funcionamiento correcto en todos ellos.
2. La capacidad de compartir maquetaciones entre dispositivos usando un servidor. La aplicación está preparada para la inclusión de un servidor para una versión futura.
3. La capacidad de insertar objetos tridimensionales en el entorno. La aplicación en una versión posterior podría funcionar perfectamente con objetos en tres dimensiones, debido al uso de OpenGL ES.

Cabe destacar que tanto el lenguaje de programación como las tecnologías utilizadas en el proyecto se dejaban a libre elección por parte del alumno.

Requisitos necesarios del dispositivo

Esta aplicación ha sido **creada** y **testeada** usando el modelo **Xiaomi A2 Lite** en **Android 9.0**.

Los requisitos mínimos obligatorios para que la aplicación funcione en un dispositivo móvil son:

	Requisitos
Tipo de Dispositivo	Android
Versión	7.0 Nougat o superior
Cámara trasera	Obligatoria
Geolocalización	Obligatorio
Giroscopio	Obligatorio
Magnetómetro	Obligatorio
Acelerómetro	Obligatorio
OpenGL ES 2.0	Obligatorio
Memoria RAM	**Mínima**: 1 GB **Recomendada**: 2 GB
Procesador	**Mínimo:** Snapdragon 410 **Recomendado:** Snapdragon 625
Pantalla	**FullHD** (1920x1080)

Tabla 2. Tabla de requisitos del dispositivo móvil

Recomiendo encarecidamente usar un procesador igual o superior al recomendado, debido a la gran carga de trabajo necesaria para procesar toda la información usada por la realidad aumentada.

También recomiendo usar un dispositivo con pantalla **FullHD**, ya que no se ha testeado la aplicación en dispositivos con pantallas de resoluciones superiores a esta. Podría pasar que el tamaño o los formatos de los menús variaran dificultando el uso de la aplicación en caso de no usar este tipo de resolución.

Esta aplicación tiene un gran consumo de batería, por lo que recomiendo tener cargado el móvil antes de su uso.

Algunos dispositivos móviles o procesadores no son compatibles con OpenGL ES 2.0. La mayoría de dispositivos móviles que cumplan con los requisitos recomendados serán compatible con esta implementación de OpenGL.

Capítulo 2: Investigación Previa

Investigación sobre la Realidad Aumentada

Actualmente, estamos en las etapas iniciales de la implementación de sistemas de realidad aumentada. Así pues, se están creando nuevas tecnologías y aplicaciones para poder aplicar RA a múltiples campos como la medicina, decoración, estudios, videojuegos, educación, etc.

La realidad aumentada, está actualmente en sus etapas iniciales de vida y, por lo tanto, aún no existen sistemas perfectos que la implementen.

Para crear un sistema de realidad aumentada, es necesario que las imágenes u objetos virtuales en tres dimensiones en pantalla actúen y se transformen de manera similar a los objetos reales. Por ejemplo, un objeto virtual tiene que disminuir de tamaño en la misma proporción en que lo haría un objeto real al alejarse de ambos una misma distancia. También, este objeto tendría que conservar unas coordenadas estáticas para que al desplazarse el usuario una cierta distancia, el objeto mantuviera su localización.

En estos últimos años se han realizado múltiples avances en este sector. Por ejemplo, uno de los avances más innovadores está siendo creado por la empresa ***Niantic*** para sus propios videojuegos de RA como el famoso juego de ***Pokemon Go***. Este sistema, mediante redes neuronales entrenadas, permite al juego conocer si un objeto virtual, está a mayor o menor distancia de los objetos reales mostrados por la cámara del dispositivo. Este sistema permite ocultar parcialmente objetos virtuales detrás de objetos reales. Le sugiero encarecidamente que vea la muestra gráfica del susodicho efecto en este video ofrecido por el propio equipo de desarrollo.

Otro problema que también plantea la realidad aumentada consiste en poder aplicar la iluminación real en objetos virtuales. Este supone un problema mucho más complejo, pero actualmente, ya existen grupos o empresas trabajando para poder crear sombras en objetos virtuales a partir de las luces reales captadas por la cámara de dispositivos móviles.

Empresas tan grandes como ***Google***, ***Amazon*** y ***Microsoft*** también están apostando en la investigación de la realidad aumentada en múltiples ámbitos como la medicina, la educación y la decoración de interiores. Estas tres empresas ya han creado sus propios prototipos de gafas de realidad aumentada como son las ***HoloLends*** de ***Microsoft*** o las ***Google Glass*** de ***Google***.

La inversión tanto en hardware como en software para investigar el campo de la RA, es muy costosa. Por ejemplo, la empresa de videojuegos ***Epic Games*** en 2018, invirtió más de 1.25 billones de dólares en investigación de realidad aumentada, situándose como la mayor inversión realizada en los últimos años en este sector.

La meta final de la realidad aumentada, es llegar a un punto donde no pueda distinguirse a simple vista un objeto virtual de un objeto real en nuestro entorno. Para conseguirlo, aún faltan décadas y muchos recursos a destinarse en su investigación.

Proyectos de Realidad Aumentada en la actualidad

La realidad aumentada desde hace unos años ha ganado mucha influencia en el sector de la decoración. El objetivo de las aplicaciones que la implementan, es permitir al usuario decorar su casa con objetos virtuales. Una de las aplicaciones más punteras de este tipo es ***Amikasa***. **Amikasa** es una aplicación gratuita para iOS que permite decorar una habitación con muebles virtuales:

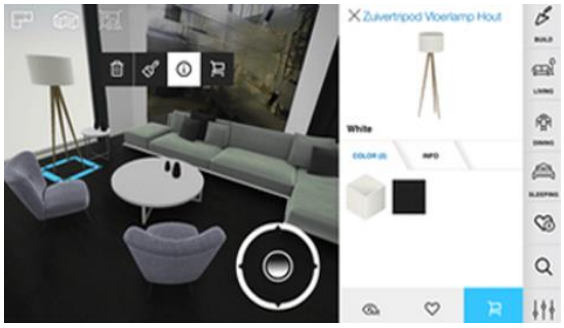

Figura 1. Ejemplo del funcionamiento de la aplicación Amikasa

También existen aplicaciones para dibujo como ***Ink Hunter*** para dispositivos Android y iOS. Esta aplicación, al dibujarse una marca de tres rayas en la piel, permite situar un tatuaje encima de esta marca y adaptarlo a la orientación y a la forma de la mano:

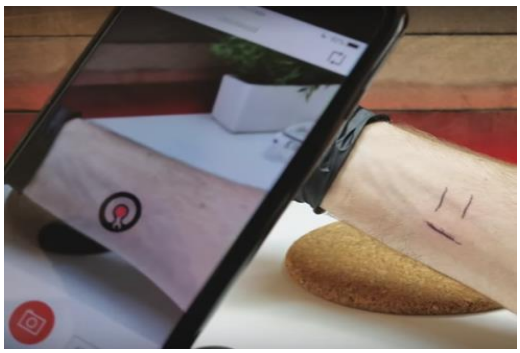

Figura 2. Ejemplo del marcado para situar el tatuaje en Ink Hunter

Figura 3. Ejemplo del cambio de orientación del tatuaje usando Ink Hunter

El problema de esta aplicación es que para poder posicionar y rotar el tatuaje de manera automática, es necesario marcar el brazo previamente. ***Google,*** usando un sistema similar de reconocimiento de marcas o letras, ya ha creado aplicaciones de realidad aumentada, capaces de traducir textos directamente desde la cámara del dispositivo:

Figura 4. Traducción de texto en tiempo real con Google Translate App

Otro ámbito donde se ha aplicado la realidad aumentada, es en el ámbito de la medicina y la educación. Por ejemplo, existen múltiples aplicaciones como ***BBC Civilizations AR*** o **Human Anatomy Atlas** usadas para mostrar por pantalla esculturas, objetos históricos, muñecos de anatomía humana, etc.

Figura 5. Ejemplo de muñeco de anatomía humano en RA usando la app Human Anatomy Atlas

La realidad aumentada aplicada a videojuegos cada vez muestra mejores resultados. Como ya se menciono en este capítulo el juego de la compañía **Niantic**, **Pokemon Go** posee uno de los sistemas mas reales de realidad aumentada en la actualidad. El sistema de RA implementado en este juego, se asemeja a las características que tiene que tener la implementación de realidad aumentada en nuestra aplicación. Este juego puede mostrar objetos en tres dimensiones en una posición alrededor del usuario. El usuario puede acercarse y alejarse de estos objetos desde todas las direcciones. De esta forma, en este juego podemos observar el objeto desde cualquier ángulo. Puedes observar un ejemplo de esta funcionalidad en este video. También si este objeto virtual está detrás de un objeto real, mediante uso de redes neuronales, es capaz de dibujar encima del objeto virtual, la parte del objeto real solapándose encima de este.

Existen dos funcionalidades que no implementa la realidad aumentada de este videojuego. La primera, es un sistema para calcular las sombras de los objetos producidas por la luz real captada por la cámara del dispositivo. Y la segunda, y la que necesita implementar nuestra aplicación, la capacidad de visualizar imágenes en localizaciones reales. En **Pokemon Go**, los objetos se posicionan de forma relativa a una cierta distancia del usuario y no en una localización real.

En nuestra aplicación, tampoco podemos implementar el acelerómetro para acercarnos y alejarnos de los objetos como se hace en **Pokemon Go**. Esto se debe, a que, en un uso prolongado del acelerómetro, la posición del objeto se ve modificada a lo largo del tiempo debido a la suma de sus errores en el cómputo de la posición del usuario. En el juego, esto no afecta al funcionamiento real del videojuego, pero en el caso de nuestra aplicación, necesitamos que la localización de cada uno de nuestros objetos sea lo más precisa posible e invariable.

Programación en Android usando Java

Android no posee un lenguaje de programación propio para el sistema operativo. Así pues, el lenguaje oficial para programar en Android es Java. También, existen otro tipo de lenguajes de programación como ***Kotlin*** (lenguaje basado en Java), ***C#***, ***C++***, etc. El principal motivo de elegir Java como el principal lenguaje de programación de mi aplicación se debe a la gran cantidad de información que existe actualmente en internet para aprender a programar Android usando este lenguaje. Al ser mi primera vez programando en Android, se decidió usar un lenguaje conocido para simplificar el entendimiento y el aprendizaje del mismo. Para empezar a aprender a programar en dispositivos Android recomiendo la lectura, como mínimo, de los primeros capítulos de este libro.

A continuación, se explicarán algunos conocimientos básicos, necesarios para entender los próximos apartados de esta memoria. En estas explicaciones se omitirá cualquier impresión de código en Java para facilitar la lectura del capítulo, pero se dará por entendido que el lector tiene unas nociones básicas de programación de este lenguaje. En caso contrario, recomiendo leer este artículo.

Android Manifest

Todas las aplicaciones en Android al programarse, poseen un documento llamado Android Manifest donde se especifica tanto las características de la aplicación, ya sean, la mínima versión requerida para usar el programa, el nombre de la aplicación, los sensores que van a usarse o los permisos que va a necesitar la aplicación. Este manifiesto se puede definir, como el esquema de los requisitos de compilación y ejecución de la aplicación. En la programación en Android, es imperativo modificar este documento con los requisitos reales o necesidades de la aplicación. Ya que, en caso de no cumplirse alguno de estos, la aplicación no va a poder ser instalada en el dispositivo.

Actividades

Una aplicación en Android se divide normalmente en diferentes Actividades. Una actividad, a grandes rasgos, y existiendo excepciones, representa o compone una pantalla de la aplicación, siendo una pantalla, la vista de una ventana con la que el usuario puede interactuar. Para crear una actividad es necesario crear una clase que herede sus atributos de la clase *Activity*. Para implementar esta herencia es necesario sobrescribir distintos métodos importantes. Estos métodos se llamarán automáticamente tanto al iniciar, pausar, cambiar de ventana o cerrar la aplicación y gestionan el ciclo de vida de la actividad.

El gráfico de la Figura 6 muestra en más detalle el orden de ejecución de estos métodos en la vida de esta actividad.

Respetar el ciclo de vida de una actividad es muy importante. Para una buena programación en Android, es necesario que la aplicación se encargue de:

1. Al **crearse** (*onCreate*), inicializar la información necesaria ya sea enlazar la app a una BDD o enlazar toda la aplicación con las diferentes vistas en XML.
2. Al **destruirse** (*onDestroy*), terminar todos los Threads en funcionamiento o cerrar todos los enlaces a BDD.
3. Al **empezar**(*onStart*) o **volver a mostrarse** (*onResume*) la actividad, recuperar o inicializar los recursos de la aplicación (inicializar conexión a una BDD, lanzar Threads, reservar memoria, cargar imágenes, etc.).

4. Al **cerrar** (*onStop*) o **pausar** (*onPause*) la actividad, dependiendo de la aplicación, será imperativo guardar y liberar los recursos usados por la aplicación antes de que la actividad deje de ser visible.

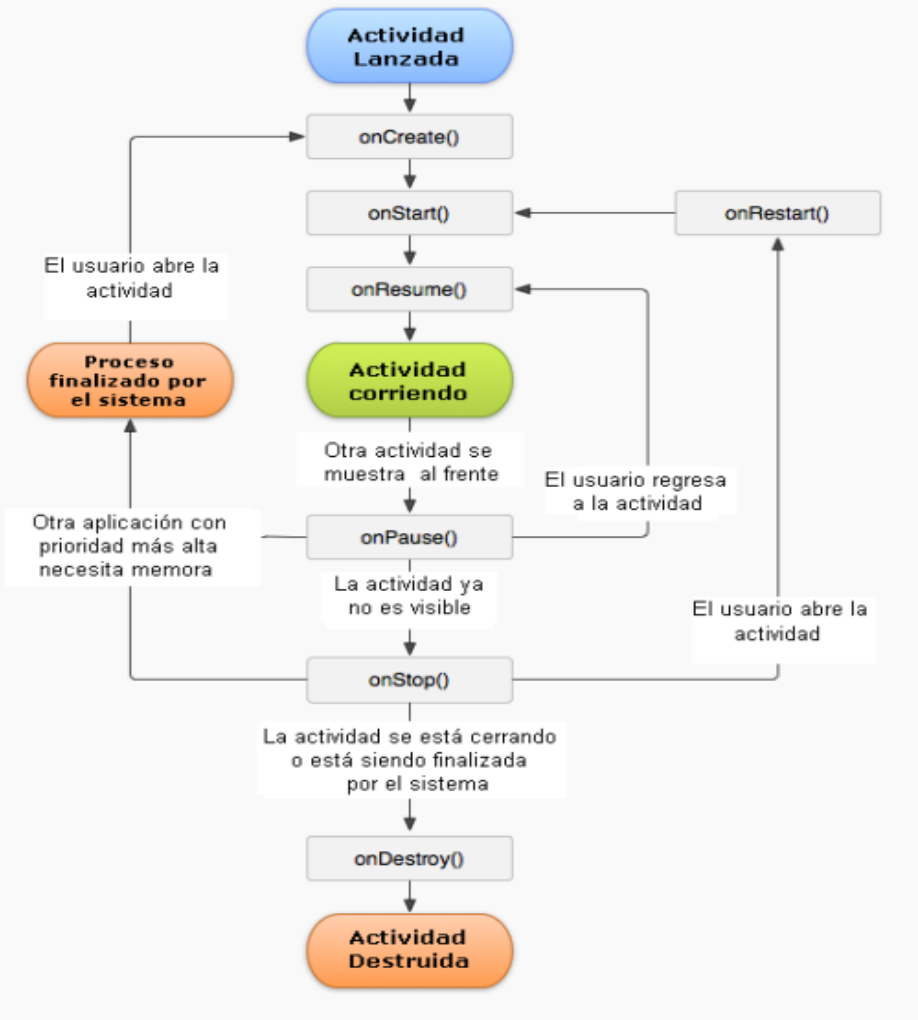

Figura 6. Ciclo de vida de una actividad en Android

Siguiendo esta praxis, se asegura que el consumo de recursos o datos de la aplicación no puedan afectar o ser afectado por otras actividades del dispositivo.

Permisos y seguridad en aplicaciones Android

Un punto central de la arquitectura de seguridad de Android consiste en que ninguna aplicación, de manera predeterminada, tiene permiso para realizar operaciones que pudieran tener consecuencias negativas para el usuario, otras aplicaciones o el sistema operativo.

Una aplicación Android a diferencia de una aplicación de ordenador necesita que el usuario, ya sea antes de ejecutarse la aplicación (Android 5.1 o versiones anteriores) o en tiempo de ejecución (Android 6.0 o versiones superiores), acepte los diferentes tipos de permisos que necesite la app para funcionar.

Para obtener estos permisos que necesita aceptar el usuario, primero de todo, se deben incluir todos los permisos necesarios para que funcione la app en el documento de Android Manifest. En versiones de Android superiores a la 5.1 será necesario implementar dentro de la ejecución de la app la petición de permisos.

En la clase *ActivityCompact* existe un método llamado *requestPermissions* que mediante el uso de una llamada con callback, el programador es capaz de pedir estos permisos al usuario y obtener la información de si han sido aceptados o rechazados. En caso de no ser aceptado un permiso, toda funcionalidad que lo necesite, queda completamente inutilizable por la aplicación.

Es por esto que para la creación de una aplicación en Android, es necesario un riguroso control de estos permisos.

Los **permisos necesarios** para el funcionamiento de esta aplicación son:

1. Permisos de **Cámara**: Para visualizar en tiempo real las imágenes percibidas por la cámara.
2. Permisos de **Localización**: Para obtener la localización del dispositivo.
3. Permiso de **Sensores Corporales**: Para poder calcular la orientación del dispositivo.
4. Permisos de **Almacenamiento**: Para poder guardar las imágenes en una BDD en memoria.

Maquetación de vistas usando XML

En Android, XML es usado para definir diferentes vistas de manera ordenada utilizando diferentes tipos de distribuciones con las que las actividades son enlazadas para mostrar e interaccionar con el usuario.

Una vista se define como un panel de N*M pixeles donde en su interior, la actividad puede mostrar imágenes, información e interactuar con el usuario.

En XML se permite asignar un identificador único a cada componente de tal forma que seamos capaces de enlazar las vistas con procesos o actividades.

Por ejemplo, si necesitamos mostrar las imágenes captadas por la cámara del dispositivo en pantalla, es necesario crear una vista V en XML y enlazarla con un proceso encargado de monitorizar la cámara. Este proceso enlazado a esta vista V, se encargará de repintar las imágenes captadas por la cámara en las dimensiones y en la posición de la vista V en la pantalla.

También, usando XML podemos crear componentes visuales, ya sean botones o menús deslizantes (*NavigationView*) como el usado en este proyecto, que, mediante interrupciones, puedan comunicar al usuario con la aplicación.

Finalmente, XML posee un tipo de elementos que permiten ordenar las posiciones de las vistas respecto a otras. Estos elementos de ordenación se llaman Layouts, y son imprescindibles para la buena distribución de las vistas por la pantalla del dispositivo.

Algunos tipos de Layouts, permiten solapar una vista encima de otra. De tal forma, que la imagen de la vista menos prioritaria se posicione detrás de otra vista y quede oculta por esta última.

Como ejemplo práctico, en nuestra aplicación usamos un RelativeLayout para solapar la vista de la cámara real debajo de la vista de realidad aumentada, de tal forma que se vean las imágenes virtuales por encima de las imágenes de la cámara.

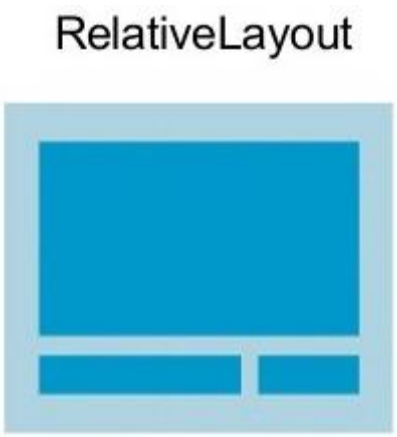

Figura 7. Ejemplo de RelativeLayout con tres vistas

Sensores

Existen diferentes tipos de sensores encargados de darnos información sobre el entorno, ya sea la localización del dispositivo, la orientación o la imagen de la cámara.

Al ser cada sensor diferente, en los posteriores capítulos del proyecto, se explicará con más detalle el funcionamiento de cada uno de los sensores usados en la aplicación por separado.

Por ahora, solo necesitamos saber que en este proyecto solo se usan tres tipos de sensores:

1. **De localización**: Este sensor, usando GPS y Wifi es capaz de obtener la localización del dispositivo con un margen de error.
2. **De rotación**: Combinación de sensores como el giroscopio, el magnetómetro y el acelerómetro que nos permiten obtener una orientación bastante precisa del dispositivo.
3. **De cámara**: La cámara es un sensor de imagen, que nos permite obtener una imagen del entorno real que observa el dispositivo y mostrarla de forma continua en una vista pre enlazada.

Almacenamiento

Al necesitar cargar en memoria las imágenes guardadas en el dispositivo, es necesario entender que diferencias existen entre el sistema de almacenamiento de un dispositivo Android y un ordenador normal.

En un dispositivo Android, las apps son instaladas dentro de la memoria interna. Existen dos tipos de memorias internas:

1. **Partición del sistema**: Espació de memoria de solo lectura donde el sistema operativo Android funciona.
2. **Espacio reservado para aplicaciones**: En este espacio, como si fuera un ordenador, cada aplicación posee una cantidad de memoria reservada donde se instala y se almacena su información.

Las diferencias entre la gestión de memoria en Android frente a las de un ordenador, radica en que una aplicación sin permisos de almacenamiento en Android, solo es capaz de leer y escribir dentro de su propio espacio reservado de almacenamiento. De tal forma, que las aplicaciones no pueden compartir un espacio de memoria si no se las autoriza.

Con las tarjetas SD pasa lo mismo, su memoria es considerada publica, y ninguna aplicación puede acceder sin permisos de almacenamiento previos.

En el caso de la memoria RAM, funciona igual que un ordenador normal, al ejecutarse una app, toda la información necesaria para ejecutarse se carga en memoria RAM. El dispositivo limpia esta memoria periódicamente o al cerrarse una aplicación.

Nuestra app, al usar Java, no necesita encargarse de liberar memoria RAM debido al Garbage Collector.

Por lo tanto, en caso de querer leer imágenes externas del dispositivo o crear una base de datos externa a este, será necesario obtener permisos de almacenamiento previamente.

Capítulo 3: Planificación del Proyecto

En este apartado se explicarán los motivos y la implementación de la metodología ágil Scrum en la creación de mi proyecto.

Elección de la metodología usada en el proyecto

Debido al tamaño del proyecto era necesaria la creación de una estrategia de planificación y elegir un tipo de metodología para que ayudase a gestionarlo.

No podía ser una metodología muy costosa en tiempo, debido a que era el único integrante en este proyecto. Los requisitos de la aplicación estaban muy bien predefinidos debido al conocimiento de mi tutor en el tema. Pero debido a que, al empezar la creación de la aplicación, desconocía por completo la estructura y el funcionamiento de las tecnologías que usaría este proyecto, necesitaba una metodología capaz de adaptarse rápidamente a los cambios y a la incertidumbre elevada que surgiría en toda su creación.

Por estos motivos, el uso de la metodología ágil Scrum, era el que más se adaptaba al proyecto, debido tanto, a su facilidad en la gestión de incertidumbre, como su fácil uso en un equipo de trabajo autoorganizado y de un solo integrante.

El uso de Scrum también fue causa de la importancia de aprender una metodología muy usada actualmente en el sector de desarrollo informático, que facilitaría adaptarme en futuros empleos.

Antes de empezar a explicar la implementación de Scrum en el proyecto, si no comprende algún apartado del documento o la terminología usada, le recomiendo la lectura de este artículo.

Implementación de Scrum en el proyecto

Para la implementación de esta metodología, usaba la pizarra de mi habitación, dividida en tres partes: una parte para las tareas no empezadas, otra para las tareas en proceso y finalmente, una para las tareas terminadas en esa semana.

Al ser solo un integrante, normalmente, si no era por causa de algún error, solo se tenía una tarea en proceso en la pizarra.

Para la implementación de la metodología Scrum en este proyecto usaba diversas pautas que cumplía semanalmente.

Creación de un Sprint semanal

Cada sábado o domingo planificaba la creación de una parte del programa para la siguiente semana. En esta etapa de planificación me preguntaba y respondía todas las siguientes cuestiones en orden:

1. **¿He cumplido todos los objetivos de la semana pasada?**
2. **¿Cuánto tiempo voy a dedicar al proyecto esta semana?**
3. **¿Cuánto tiempo voy a tardar en realizar todas las tareas atrasadas?**
4. **¿Qué módulo o parte del programa se ajusta más al tiempo restante del qué dispongo?**
5. **¿Qué tareas diarias surgen de la división de este módulo de trabajo?**

Al contestar todas estas preguntas, obtenía una meta para la semana y un listado de tareas donde cada una de ellas, ocupaba un tamaño menor a un día de trabajo.

Un ejemplo de meta podía ser: implementación del sensor de imagen, optimización de recursos en la carga de imágenes en memoria, etc.

Estas tareas podían consistir tanto en la mejora o solución de errores de un apartado de la aplicación ya existente, como en la creación de nuevas funcionalidades o módulos.

Scrum diario

Al ser el único integrante del equipo, no era necesario una reunión diaria por la mañana. Normalmente solo consultaba 5 minutos las tareas que quedaban en la pizarra para saber si necesitaría aumentar el tiempo de trabajo esa semana.

Cada vez que terminaba o empezaba una tarea, ponía el **post-it** en el lugar correspondiente. Ese día, me dedicaba a ejecutar la tarea más prioritaria en proceso.

Gestión de las Tareas

Cada nueva tarea se creaba utilizando un **post-it** con los siguientes datos:

1. **Número del Sprint al que pertenecía**
2. **Nombre o descripción de la tarea**
3. **Fecha de creación**
4. **Fecha de finalización**
5. **Prioridad:** Número que representaba en que orden se realizaría la tarea en la semana.
6. **Color / Tipo de tarea:** Cada color de los **post-its** que usaba representaba un tipo de tarea diferente.
7. **Bloqueo**: Este campo indicaba que tareas eran obligatoriamente necesarias antes de poder realizar o completar esta tarea.

Al ser solo un integrante y hacer las tareas secuencialmente, normalmente no necesitaba el campo de bloqueo. En la misma prioridad de la tarea, tenía en cuenta estos bloqueos para su ordenación. En algunos casos, sobre todo, al final de la creación de la aplicación, el campo de bloqueo resultaba imprescindible para la gestión de las distintas tareas de unión de módulos y funcionalidades.

Los campos, de redactor y ejecutor de la tarea no eran necesarios debido a que no había más de dos personas en el equipo.

Las tareas se podían distinguir en tres tipos:

1. **Tarea de investigación:** Estas tareas se usaban para investigar las funcionalidades de la aplicación antes de su implementación.
2. **Tarea de ejecución**: Para la programación de un apartado de la aplicación.
3. **Tareas de errores**: Tareas sin límite de tiempo, que se basaban en la solución de errores en el código.

Las **tareas de investigación** duraban aproximadamente un día entero, para asegurarme de reducir toda la incertidumbre de una o más funcionalidades lo máximo posible.

Las **tareas de ejecución** las creaba para que duraran menos de dos o tres horas. Este era el tiempo que normalmente podía dedicarle al día al proyecto. Muchas veces, debido a la incertidumbre de la tarea, acababan durando bastante más del tiempo estimado y

era necesario dedicarle más horas ese día, o atrasar alguna tarea del Sprint para la semana siguiente.

Las **tareas de errores** eran las únicas en las que no administraba el tiempo. Cuando surgía una tarea de este tipo en la aplicación, se creaba inmediatamente un **post-it** y se le administraba la máxima prioridad sin un límite de tiempo. Estos fallos se solucionaban antes de realizar nuevas **tareas de ejecución**, para no arrastrar y conservar los errores en futuras tareas relacionadas.

Revisión del Sprint y Retrospectiva del Proyecto

Antes de empezar a planificar el próximo Sprint, revisaba que tareas faltaban por acabar y valoraba los motivos de cada uno de estos retrasos. Normalmente se debían completamente a la incertidumbre o a la falta de investigación anterior a un tema técnico que no se había planificado. Todos estos errores se anotaban en una libreta para no repetirse en futuros Sprints.

También se daba el caso, de que tardaba más, o a veces menos tiempo en realizar un Sprint que lo planificado. Si pasaba eso, para futuros Sprints, se aumentaba o reducía el tiempo de ese tipo de tareas en concreto.

Finalmente, hacia una retrospectiva del proyecto para calcular aproximadamente, que porcentaje de la aplicación llevaba realizada y como podía evitar o solucionar los errores de planificación realizados para la siguiente semana.

Al acabar un Sprint, todos los **post-it** acabados se quitaban de la pizarra, para dejarla limpia antes de la creación del nuevo Sprint semanal. Para la siguiente semana, todas las tareas que quedaran por realizar, pasarían a ser más prioritarias que las nuevas.

Conclusiones sobre la aplicación de Scrum en el proyecto

Encuentro que aplicar Scrum en mi proyecto, ha sido imprescindible para el buen funcionamiento y gestión del trabajo.

Al gestionar volúmenes de trabajo tan grandes en grupos de trabajo tan pequeños, recomiendo encarecidamente el uso de esta metodología por su simpleza y rapidez.

El tiempo dedicado a su implementación es mínimo y lo compensa encarecidamente con el tiempo ahorrado en gestión de errores y tareas incompletas. Sin usar esta metodología, me atrevo a afirmar, que hubiera tardado muchas más horas en la creación de esta aplicación.

Capítulo 4: Estructura de la aplicación

Al principio del desarrollo del proyecto, este presentaba dos problemáticas principales, este proyecto tenía que ser creado para permitir el cambio completo de la estructura de datos, de tal manera que, si en un futuro se modificaba la aplicación para usar un servidor, fuera posible modificar la aplicación de manera sencilla. También, al empezar el desarrollo, se presentaba un problema de incertidumbre con el apartado gráfico. Al principio, no podía conocer que solución usuaria para la creación del apartado gráfico de la aplicación, o que cambios tendría que realizar para llevarla a cabo.

Por estas dos razones, era necesario crear la aplicación usando un patrón de estructuración de código capaz de proteger todos los datos, así como el funcionamiento de los sensores de localización y giroscopio (sensores no visuales) de las modificaciones del apartado gráfico debido a la incertidumbre.

Como resultado, opté por crear mi aplicación completamente usando un patrón Modelo-Vista-Presentador (MVP).

Modelo Vista Presentador (MVP)

MVP es un patrón arquitectónico de interfaz de usuario diseñada para facilitar pruebas de unidad automatizada y mejorar la separación de inquietudes en lógica de presentación.

Este patrón arquitectónico se divide en tres partes principales:

1. **Modelo de Datos**: Posee toda la lógica del guardado de información y datos de la aplicación. Para este, se define una interface que indica que datos pueden mostrarse o modificarse. Esto permite, tanto proteger la gestión de datos como la capacidad de modificación del modelo sin afectar al resto de la aplicación mientras implemente las mismas funcionalidades de la interface.

2. **Presentador**: Intermediario entre la Vista y el Modelo de Datos, es el encargado de normalizar los datos entre estos y de realizar las tareas no gráficas de la aplicación.

3. **Vista**: Se encarga de la gestión del apartado gráfico de la aplicación junto a la interacción con el usuario.

Este patrón es normalmente utilizado en proyectos realizados para Android debido a la necesidad de compartir y proteger una lógica o funcionalidad común entre múltiples actividades.

Implementación del patrón MVP en la aplicación

Como podemos observar en la **Figura 8**, la aplicación se estructura en tres partes, cada una de ellas, dividida en diferentes módulos o estructuras:

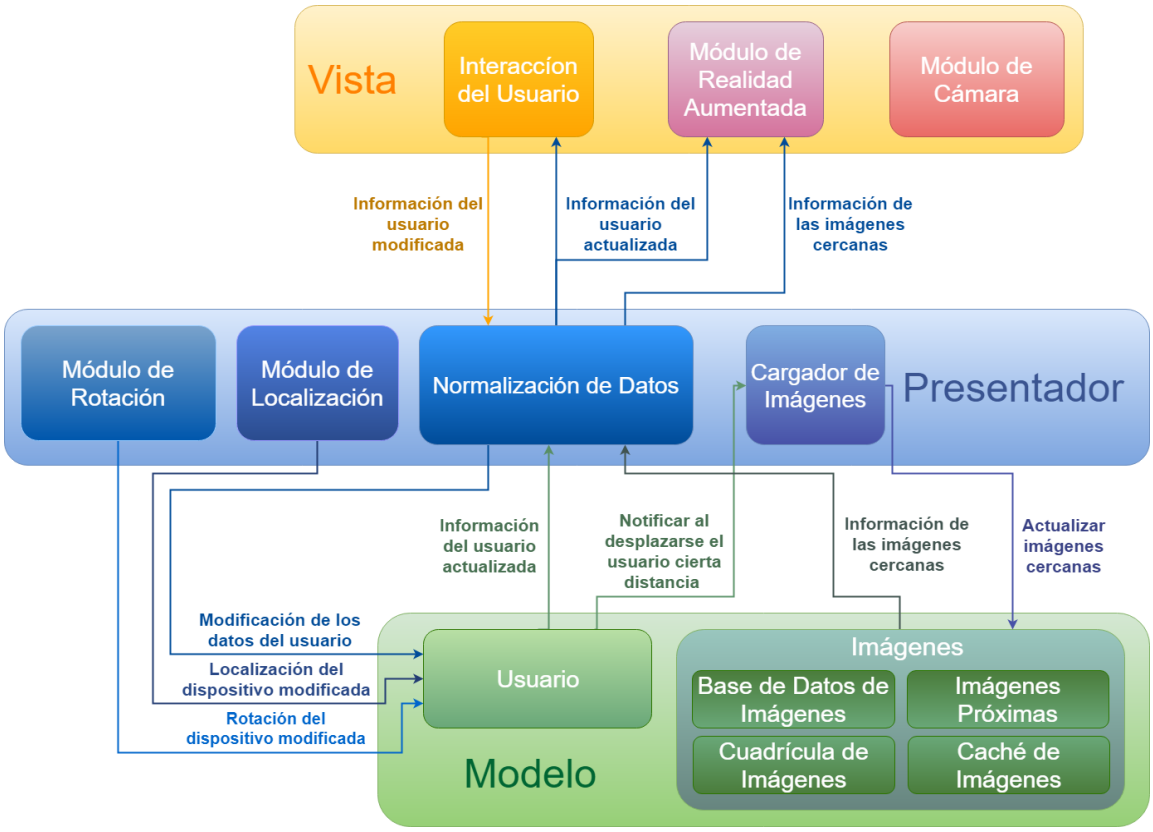

Figura 8. Esquema de la estructura de la aplicación

Como es propio del patrón de MVP, en este esquema observamos la división de nuestra aplicación en las partes de la Vista, el Presentador y el Modelo de Datos. En esta aplicación, la Vista no posee ninguna conexión o comunicación directa con el Modelo de Datos, de tal forma que es el propio Presentador, el encargado de proporcionar y normalizar los datos para ella. En segundo lugar, podemos observar como el Presentador posee toda la lógica tanto del Módulo de Localización como del Módulo de Rotación, debido a que su funcionamiento no depende de ningún campo visual. El único sensor usado por la aplicación en la Vista es el sensor de imagen usado dentro del Módulo de Cámara. Todos estos módulos o apartados de la aplicación poseen un capítulo propio en este documento.

Estructuración de la Vista

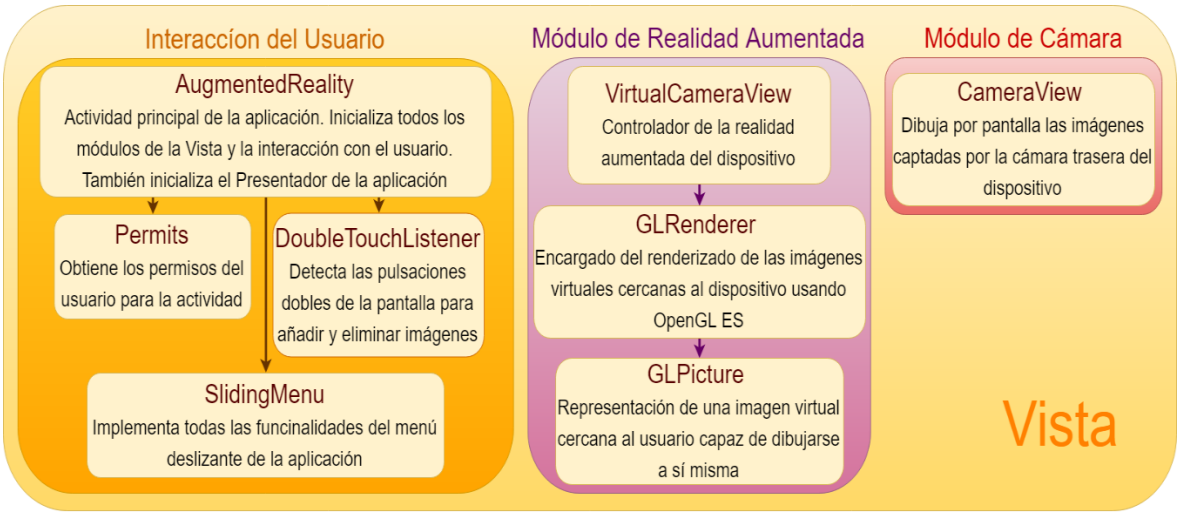

Figura 9. Esquema de las clases creadas para la Vista de la aplicación

Como se muestra en la Figura 8, la vista está dividida en tres partes completamente independientes entre ellas:

1. **Interacción del Usuario**: implementación de todo el apartado con el que puede interactuar el usuario, ya sea para modificar su altura, para aumentar o disminuir la distancia a la que quiere insertar la imagen o para realizar la acción de insertar o eliminar una imagen. Este apartado se comunica con el Presentador, para mostrar tanto la configuración actual de la aplicación al usuario, como para poder cambiar la configuración de este usuario y actualizarla en el Modelo de Datos.
2. **Módulo de Realidad Aumentada:** módulo completamente visual encargado de mostrar las imágenes cercanas al usuario en la pantalla del dispositivo. Usando OpenGL, la rotación y la geolocalización del dispositivo, esta genera el efecto de realidad aumentada en la aplicación.
3. **Módulo de Cámara**: módulo encargado de mostrar las imágenes captadas por el sensor de imagen en la pantalla del dispositivo de forma periódica.

Estructuración del Presentador

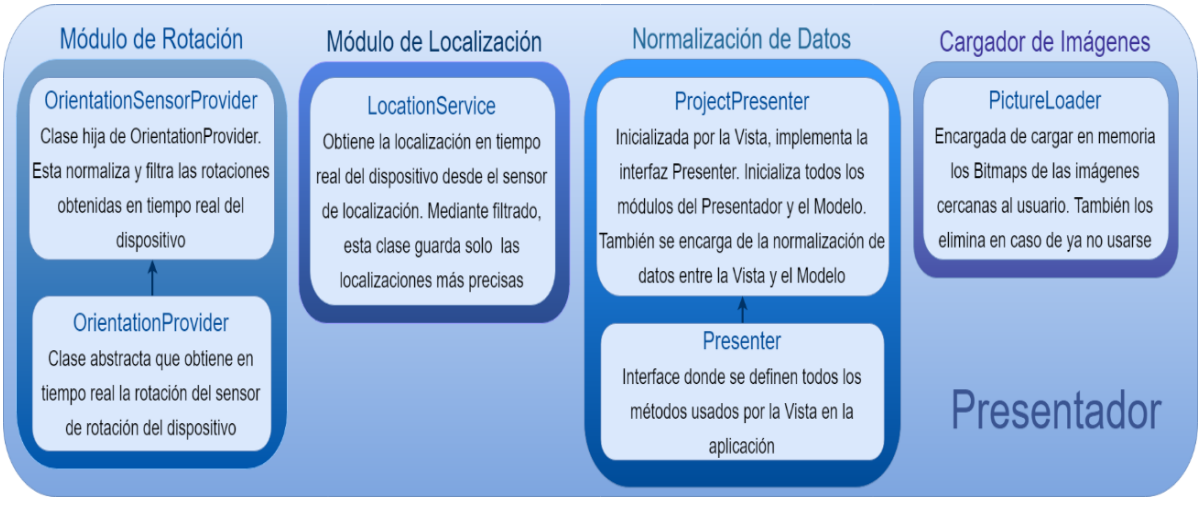

Figura 10. Esquema de las clases creadas para el Presentador de la aplicación

El Presentador, agrupa toda la lógica no visual de la aplicación. Este tampoco interactúa directamente con el usuario. Está dividido en tres submódulos y una clase con un conjunto de métodos encargados del traspaso de información entre el Modelo de Datos y la Vista. Por ejemplo, cuando el Módulo de Realidad Aumentada pide la localización del usuario al Presentador, este se encarga de obtener y normalizar la geolocalización guardada como ángulos dentro del Modelo de Datos para transformarla en metros. Los tres módulos independientes dentro del Presentador son:

1. **Módulo de Rotación**: módulo encargado de monitorizar la rotación del dispositivo y actualizarla periódicamente. Este guarda la rotación normalizada dentro del Modelo de Datos, concretamente, en el apartado de información de usuario.
2. **Módulo de Localización**: módulo encargado de monitorizar la geolocalización del dispositivo y actualizarla en el Modelo de Datos al recibir una nueva localización del dispositivo.
3. **Cargador de Imágenes**: módulo encargado de actualizar y cargar las imágenes cercanas al dispositivo usando un Thread separado para no interrumpir la visualización de las imágenes.

Estructuración del Modelo de Datos

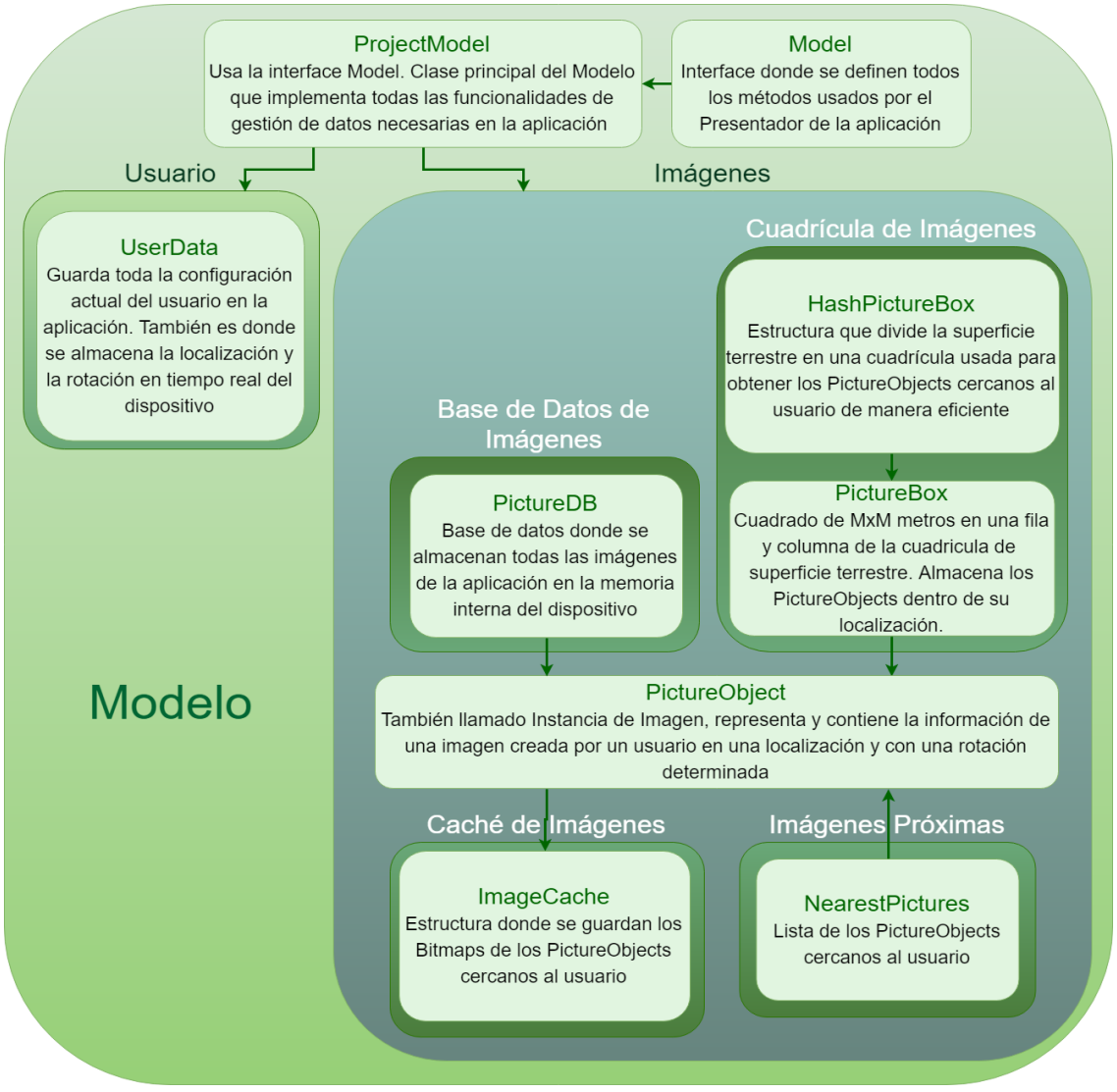

Figura 11. Esquema de las clases creadas para el Modelo de Datos de la aplicación

El Modelo de Datos de este proyecto contiene dos estructuras de datos principales: una para guardar toda la información o configuración actual del dispositivo llamada Usuario y otra más compleja para la gestión de imágenes. Para la gestión de las imágenes junto a su información fue necesaria la creación de cuatro estructuras diferentes:

1. **Base de Datos de Imágenes**: Implementación de una BDD simple para el almacenamiento de imágenes en memoria interna, de tal forma, que las imágenes permanezcan intactas entre distintas ejecuciones de la aplicación sin eliminarse.
2. **Cuadrícula de Imágenes**: Esta estructura fue creada por la necesidad de poder obtener las imágenes cercanas al dispositivo.
3. **Imágenes Próximas**: Listado ya procesado de las imágenes cercanas al usuario para que el Módulo de Realidad Aumentada pueda mostrarlas en pantalla.
4. **Caché de Imágenes**: Estructura encargada de guardar las imágenes en memoria RAM, de tal manera que no sea necesario cargar múltiples veces una imagen desde la memoria interna del dispositivo. Esta estructura permite ahorrar significativamente el consumo de memoria RAM y tiempo de cómputo en el dispositivo.

Capítulo 5: Implementación del Modelo de Datos

Empezando la explicación del Modelo de Datos desde sus raíces, la aplicación posee dos tipos distintos de datos básicos: usuarios e imágenes.

Tipos de datos

Para el guardado de los datos de un dispositivo o usuario se creó la clase *UserData*, mientras que, para el guardado de una imagen se creó la clase *PictureObject*.

Datos del Usuario

Llamaremos Datos del Usuario a la implementación de la clase *UserData*.

Esta clase posee información esencial del dispositivo y sirve para guardar toda la configuración seleccionada por el usuario en tiempo real dentro de ella. También, es esta clase la que guarda la información de la localización y rotación en tiempo real del dispositivo para su uso en la Vista de la aplicación.

La clase UserData fue creado para que en futuras versiones de la aplicación, se añada la capacidad de poder cambiar la configuración de un usuario a otro y poder guardar la configuración del usuario en la memoria interna del dispositivo al cerrarse la aplicación.

Esta clase implementa la interface *Observable*. La clase que implemente esta interface, mediante una notificación, es capaz de iniciar una interrupción a todos los observadores conectados a ella. Un observador, corresponde a una clase que implementa una interface *Observer*. Para que un observador pueda recibir estas notificaciones, previamente ha de enlazarse a una clase que implemente la interface *Observable*. Cada vez que la clase *Observable* notifique a la clase *Observer*, genera una interrupción en esta última para que realice un cometido determinado.

Puede encontrar más información sobre estas interfaces en este ejemplo.

Instancia de Imagen

Llamaremos Instancia de Imagen a una instancia creada de la clase *PictureObject*.

Es la estructura base en la que se fomenta todo el Modelo de Datos. Una instancia de la clase *PictureObject*, representa una imagen situada por un usuario, en una localización con una rotación determinada. Esta instancia también posee un ratio de pixeles por centímetro para modificar el tamaño de la imagen.

Una Instancia de Imagen posee estos atributos:

1. Id única de la imagen
2. Id del usuario
3. El Path a la imagen
4. La localización real de la imagen
5. La rotación de la imagen
6. El ratio de pixeles por centímetro de la imagen (Tamaño)

Ninguna Instancia de Imagen posee un atributo o variable que apunte a una imagen cargada en memoria. La carga de imágenes se realiza en una estructura aparte. Esto se debe a que solo es necesario cargar una imagen para todos las Instancia de Imagen que poseen el mismo Path hacia una imagen.

Una Instancia de Imagen nunca se actualiza después de su creación. De esta forma, se puede afirmar que cada Instancia de Imagen representa un registro que guarda el Path de una imagen, con todo un listado de atributos con los que el usuario la insertó en el espacio virtual.

Estructura de la Base de Datos

En este proyecto, solo era necesario guardar en memoria las Instancias de Imagen del usuario. Para este problema, se debatieron dos soluciones simples, debido a la poca complejidad de este.

La primera, era la creación de un fichero de Instancias de Imagen, que se cargaría en memoria al cargar la aplicación y se guardaría al cerrarse esta.

La segunda, era la creación de una base de datos que nos permitiera guardar y eliminar elementos en tiempo de ejecución. Esta estaría formada una sola tabla donde cada fila correspondería a la información de una Instancias de Imagen.

La primera solución no era incorrecta, el volumen de imágenes que puede poseer un solo usuario en la aplicación es mínimo. Incluso si insertara 1000 Instancias de Imagen en memoria, la aplicación no necesitaría mucho tiempo para el cargado y guardado del listado en un fichero. El principal problema de esta solución, radicaba en la inseguridad de guardar los datos al cerrar la aplicación en un sistema Android. Por ejemplo, si el usuario cancelaba los permisos de almacenamiento de la aplicación en tiempo de ejecución, todas las modificaciones realizadas al listado de imágenes ya no podrían guardarse.

Por ese motivo se decidió usar la segunda opción y crear una base de datos con una sola tabla de Instancias de Imagen.

Esta BDD se creó usando SQLite, un gestor de bases de datos relacional muy popular en el desarrollo Android debido a la simplicidad de su uso. Para usar SQLite, se debe crear una nueva clase que herede de la clase *SQLiteOpenHelper*. En la aplicación, la clase que implementa esta base de datos recibe el nombre de *PictureDB*. En esta nueva clase se define la estructura de las tablas y sus relaciones. Para usarse, solo es necesario abrir una conexión a esta base de datos en el método *onResume* de la actividad. Con la conexión ya abierta al iniciarse la aplicación, ya podremos ejecutar consultas a la BDD mientras la aplicación esté ejecutándose. Al ejecutarse el método *onPause*, será necesario cerrar esta conexión para evitar posibles fallos.

Estructuras para la optimización de recursos

Uno de los problemas principales de este proyecto es la optimización de recursos para dispositivos incrustados. En esta práctica se han considerado múltiples problemas de optimización para mejorar el rendimiento y la escalabilidad del Modelo de Datos.

Se han creado tres soluciones tres problemas de optimización que afectaban al rendimiento de la aplicación significativamente.

Cuadrícula de Imágenes

Uno de los principales factores de consumo de recursos en una aplicación de realidad aumentada, se produce al procesar e dibujar imágenes en pantalla.

Por eso, tenemos que regular y reducir de alguna forma la cantidad de información procesada por el Módulo de Realidad Aumentada al mínimo, de tal forma que sea capaz

de mostrar una mayor cantidad de Instancias de Imagen con una mayor cantidad de fotogramas por segundo.

Por ejemplo, no podemos pasar una lista completa de todas las Instancias de Imagen en la base de datos al Módulo de Realidad Aumentada para su procesado. Muchas Instancias de Imagen estarán demasiado lejos para poder observarse. Si las mandamos a procesar, solo aumentará el tiempo de procesado y el consumo en memoria RAM innecesariamente. De esto deducimos, que, si pudiéramos conocer que Instancias de Imagen están situadas dentro de un rango visible para el dispositivo y solo procesáramos esas, seriamos capaces de reducir significativamente la cantidad de procesamiento del apartado gráfico de la aplicación.

Para este problema existen distintas soluciones: una de ellas, por ejemplo, sería calcular la distancia entre el dispositivo y todas las Instancias de Imagen para luego filtrar las que posean una distancia inferior a un radio de visión preestablecido.

El problema con esta solución, se debe a que si calculáramos la distancia entre la localización del dispositivo y todas las Instancias de Imagen de la BDD, cada vez que el dispositivo cambia su localización, será necesario recalcular todas estas distancias. Por ese motivo, este sistema de filtrado seguiría siendo bastante ineficiente para un volumen superior a 1000 Instancias de Imagen.

Otra forma de solucionar este problema, sería crear una estructura que fuera capaz de agrupar Instancias de Imagen por proximidad, de tal forma que no fuera necesario comparar la distancia del usuario con la todas las Instancias de Imagen.

De esta manera surge la idea de crear una estructura a la que llamaremos **Cuadrícula de Imágenes**.

Esta estructura se basa en la división del espacio terrestre en cuadrados de **M x M** metros, donde **M** es una variable de inicialización que predefine el **tamaño en metros del lado de los cuadrados** de esta cuadrícula terrestre.

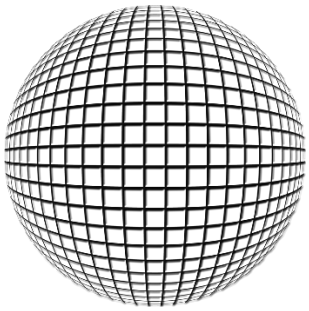

Figura 12. Representación de una cuadrícula en el espacio terrestre

Para crear esta estructura, en un primer momento, podríamos plantear la creación de una Matriz de **N x N**, donde **N** es el **número de cuadrados** de **M x M** tamaño que caben en el diámetro terrestre (12.742 kilómetros). Cada uno de estos contendría una lista con todas las Instancias de Imagen dentro de esas coordenadas.

El problema principal de esta solución, se debe a que el mapeado completo de esta cuadrícula, siendo los cuadrados, por ejemplo, de unas dimensiones de 40 x 40 metros, se necesitaría crear una Matriz de 318.550 x 318.550 filas y columnas. Esto se traduce en la creación de más de **100 millones de punteros a listas**. Esta estructura es completamente inviable. Solo para reservar la memoria de los punteros de estas listas, se necesitarían más de **378 Terabytes de memoria RAM**.

La solución adecuada en este caso, y la implementada en esta aplicación, es la creación de una Tabla de Hash que actuaría de cuadrícula, donde solo se crearían y mapearían los cuadrados que contuvieran Instancias de Imagen dentro de sus localizaciones. Al solo poseer un usuario, el número de cuadrados que mapearía la aplicación sería ínfimo comparado al total planteado en la matriz. Esta estructura tendría muy poco coste en recursos y seguiría siendo igual de rápida.

La **Cuadrícula de Imágenes** de la aplicación, se implementa en la clase *HashPictureBox* y consiste en una Tabla de Hash cuya clave es la posición del cuadrado en la cuadrícula terrestre y el valor o elemento un objeto de la clase *PictureBox* o **Cajas de Imágenes**. Cada una de estas **Cajas de Imágenes** se corresponde a un cuadrado de la cuadrícula terrestre y contiene la fila y la columna donde está situada junto a una lista de todas los Instancias de Imagen dentro de esta caja o cuadrado.

Con esta estructura, para conocer las Instancias de Imagen próximas al usuario, es tan simple como a partir de la localización del usuario y dividiéndola por el tamaño **M** de una **Caja de Imágenes**, obtener las coordenadas de la *PictureBox* donde está situado el usuario y las adyacentes. Cuando ya hemos obtenido todas estas **Cajas de Imágenes**, solo debemos obtener las Instancias de Imagen dentro de estas:

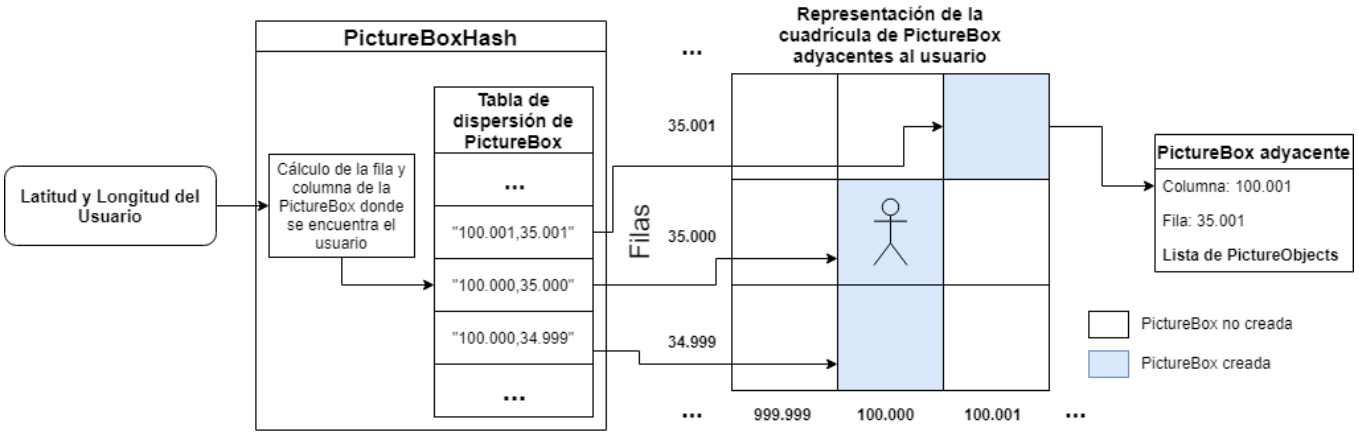

Figura 13. Estructura de la Cuadrícula de Imágenes

El único problema de esta estructura, es que necesita el cargado previo de todas las Instancias de Imagen de la Base de Datos a memoria. Aunque, haciendo pruebas con más de 5000 Instancias de Imagen, el tiempo de carga de esta estructura sigue siendo inmediato.

Sistema de Caché de Imágenes

Para el cargado en memoria de imágenes usando Java en dispositivos Android, es necesario utilizar una clase llamada Bitmap. Esta clase es necesaria para el cargado de texturas en OpenGL ES. En Android, un Bitmap se encarga de cargar y gestionar una imagen en memoria RAM, guardando una matriz de píxeles con el color de cada uno de estos junto a otras propiedades de la imagen como el tamaño.

Estos Bitmaps ocupan bastante memoria en el dispositivo, por este motivo, es necesario reducir al mínimo la cantidad de Bitmaps cargados. Una manera de ahorrar memoria, es utilizar un mismo Bitmap para todas las Instancias de Imagen que comparten una misma imagen.

La solución implementada para solucionar este problema, consiste es usar un hash de Bitmaps, donde la clave de esta Tabla de Hash corresponde al Path a la imagen y el valor es un puntero al Bitmap de la imagen ya cargada en memoria.

Cada vez que se pinte una Instancias de Imagen por pantalla, será necesario buscar en este sistema de caché, el Bitmap que corresponde al Path dentro de la Instancias de Imagen. En caso, de que no exista, cargaremos el Bitmap y lo añadiremos al sistema de caché. Esta estructura, también permitirá limpiar de manera simple, los Bitmaps no utilizados.

Lista de Imágenes Próximas al Usuario

Esta lista de Instancias de Imagen fue creada para cachear el resultado de la búsqueda de las imágenes próximas de la Cuadrícula de Imágenes. De esta manera, el Módulo de Realidad Aumentada no necesita ningún cómputo extra para obtener Instancias de Imagen a la hora de pintarlas por pantalla.

Esta lista funciona junto a una variable booleana que indica a la aplicación si la lista ha sido modificada recientemente. De esta forma, el Módulo de Realidad Aumentada pueda conocer si se ha producido una modificación en esta lista o en las Instancias de Imagen cercanas al usuario.

Capítulo 6: Cargador de Imágenes

Implementación del Cargador de Imágenes

El **Cargador de Imágenes** se corresponde a la implementación de la clase *PictureLoader* en la aplicación.

Uno de los objetivos de cualquier aplicación gráfica, consiste en separar cualquier cómputo de datos de la Vista. Esto significa, que el cargado de imágenes en caché de nuevos Bitmaps y la actualización de la lista de imágenes cercanas no puede producirse en el mismo Thread que usa la Vista. Por ese motivo, estos cálculos necesitan procesarse en otro Thread separado del gráfico. La implementación del Thread de esta clase se basa en este artículo de Rodrigo Santamaría.

De esta idea de crear un Thread que se encargue únicamente del cargado de Bitmaps en memoria surge el **Cargador de Imágenes**, una clase que hereda sus propiedades de la clase Thread.

Una instancia de la clase del **Cargador de Imágene**s, se encarga, en un Thread separado, de tres tareas distintas:

1. Actualizar las imágenes cercanas al usuario al traspasar este, la frontera entre dos **Cajas de Imágenes** dentro de la Cuadrícula de Imágenes.
2. Cargado e inicialización de todos los Bitmaps en cache necesarios para la lista de imágenes próximas al usuario.
3. Limpieza y eliminación de los Bitmaps sin usarse por alguna Instancias de Imagen cercana al usuario en el sistema de caché de imágenes.

Para reducir el coste de esta clase, esta no ejecuta estas acciones mediante una monitorización de la localización del usuario. Una instancia de esta clase, al realizar todas estas tareas, se bloquea mediante el uso del método *wait* de la propia clase, a la espera de alguna notificación que lo despierte.

Esta clase, implementa la interface *Observer*. Esta interface permite obtener notificaciones de los Datos del Usuario que implementa la interface *Observable*. Estas notificaciones son lanzadas cuando el usuario cambia la **Caja de Imágenes** en la que está situado. Toda esta implementación es transparente para el **Cargador de Imágenes**, de tal forma, que él no sabe a qué clase o estructura del Modelo de Datos se ha enlazado.

Ciclo de Vida

Al ejecutarse el método *onResume* en la actividad principal, la Vista pide al Presentador iniciar un Thread de esta clase. El Cargador de Imágenes, por primera vez, crea la lista de imágenes cercanas.

Mientras la aplicación este ejecutándose, el **Cargador de Imágenes** seguirá bloqueado hasta recibir una notificación del Modelo de Datos. Al recibir una notificación, ejecutará sus tareas y volverá a bloquearse.

Al iniciarse el método *onPause* en la actividad principal, el Presentador enviará una notificación al **Cargador de Imágenes** para que finalice su ejecución.

Capítulo 7: Implementación de los sensores

Uno de los factores principales que permite crear la realidad aumentada, son los sensores del dispositivo móvil. Esta aplicación usa tres tipos de sensores: de localización, de rotación y de imagen. Cada uno de estos sensores pertenece a un módulo que separa su funcionamiento del resto de la aplicación.

En este apartado, explicaremos el funcionamiento y estructura de estos tres módulos. Antes de empezar la explicación del Módulo de Localización serán necesarios conceptos generales sobre geolocalización.

Conceptos básicos de la geolocalización

La geolocalización es la capacidad de un dispositivo de obtener la ubicación real de un objeto. En el caso de los dispositivos móviles, la geolocalización es usada para obtener su propia localización.

La ubicación o localización real de un objeto en la Tierra viene dada por tres medidas: la latitud, la longitud y la altitud.

La latitud, es la distancia angular que existe desde cualquier punto de la Tierra con el Ecuador.

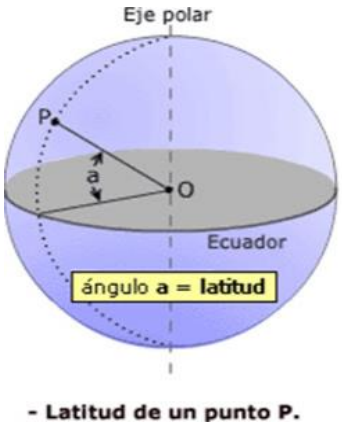

Figura 14. Representación de la distancia angular en latitud

La longitud, es la distancia angular que existe desde cualquier punto de la Tierra respecto al **meridiano de Greenwich**:

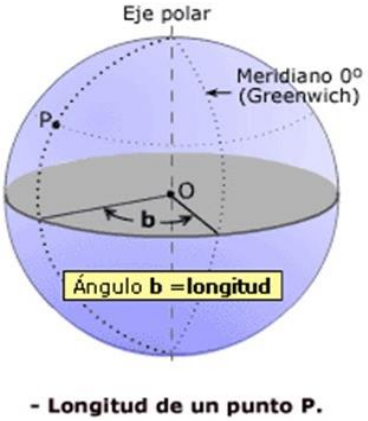

Figura 15. Representación de la distancia angular en longitud

El **meridiano de Greenwich**, también conocido como el meridiano cero, es la circunferencia imaginaria que une los polos y que recibe su nombre por cruzar por el distrito londinense de Greenwich:

Figura 16. Imagen del Meridiano de Greenwich

Tanto **la latitud** y **la longitud**, poseen una distancia angular de **-180º hasta los 180º**, rodeando toda la superficie terrestre.

La altitud y no la altura, consiste en la distancia vertical desde un plano horizontal, considerando este plano la altura media del mar:

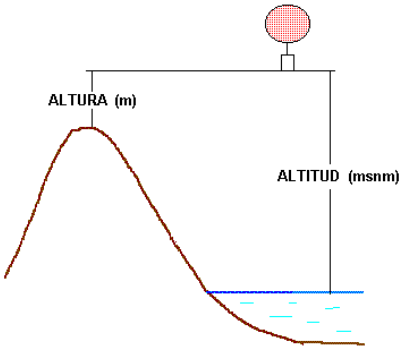

Figura 17. Ejemplo de la diferencia entre la altura y altitud

Normalmente, todos los dispositivos de medición suelen dar la medida de la altitud en metros.

Tipos de geolocalización usables en un dispositivo móvil

Existen tres maneras diferentes de obtener la geolocalización en un dispositivo móvil.

La primera, y la más conocida es el GPS o **Sistema de Posicionamiento Global**. Este sistema obtiene la ubicación del disipativo utilizando la señal de cuatro o más satélites distintos. Con tres satélites, serias capaz de estimar la localización aproximada del dispositivo, pero solo suponiendo que el dispositivo está situado a **altitud** cero (nivel del mar).

Cada uno de estos satélites emite una señal con información de su ubicación cada cierto periodo de tiempo. El dispositivo capta estas señales y teniendo en cuenta la latitud, longitud, altura del satélite y el tiempo en el que ha tardado en recibir la señal desde que el satélite la creó, se calcula una ubicación aproximada. Cuantos más satélites tomen parte en el proceso, más exacta será la triangulación.

Figura 18. Ejemplo de geolocalización por GPS

La segunda manera de obtener la ubicación del dispositivo es usando el **Sistema Global para Comunicaciones Móviles** o GSM. Este sistema utiliza las torres de comunicación cercanas que dan servicio al teléfono para obtener, mediante la intensidad de señal y el tiempo que tarda el dispositivo en recibir estas señales, una localización aproximada del dispositivo. Esta localización puede tener un margen de error aproximado de 200 metros.

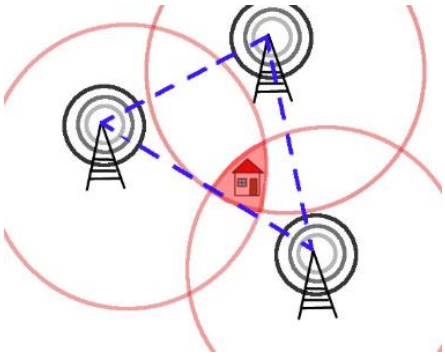

Figura 19. Ejemplo de geolocalización por GSM

Finalmente, la tercera manera de obtener la ubicación del dispositivo consiste en el uso de redes Wifi. Cada red Wifi emite una señal identificativa, de tal forma, que sabiendo la distancia entre el dispositivo y el rúter de la red a la que está conectado, se puede conocer de manera aproximada su geolocalización:

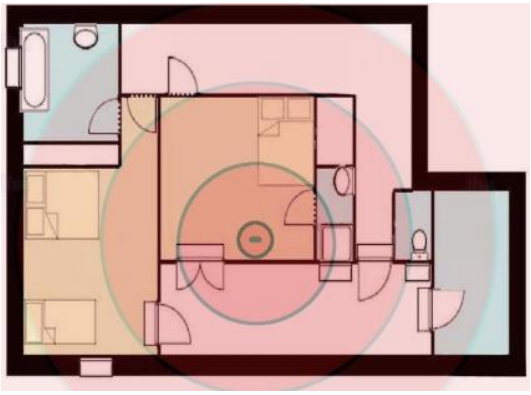

Figura 20. Ejemplo de geolocalización por Wifi

Uso de la geolocalización en el proyecto

Inconvenientes del uso de geolocalización para Realidad Aumentada

El problema del uso del GPS, Wifi o GSM, es la imprecisión de estos sistemas para obtener la localización real del dispositivo. Los principales problemas de usar sistemas de geolocalización en aplicaciones de realidad aumentada son:

1. La altitud que se obtiene de estos, posee un margen de error mínimo de entre 30 y 50 metros. Este margen de error es demasiado alto para usarse en una aplicación de realidad aumentada.
2. Cuando se obtiene una nueva localización, esta posee un atributo que nos indica la precisión aproximada de los resultados obtenidos. Pero, al filtrar los resultados con un margen de error elevado, aumenta el tiempo en que se tarda en obtener una nueva localización válida. Esto significa, que, cuanto más queramos aumentar la precisión de la localización, necesitaremos mayor cantidad de tiempo para obtenerla, pudiendo llegar a esperar hasta 10 segundos para obtener un nuevo resultado.
3. Las localizaciones precisas que obtiene el dispositivo pueden llegar a tener un margen de error de hasta 5 metros de distancia en latitud o longitud. Solo los dispositivos posteriores a 2018 que posean el **chipset BCM47755** de geolocalización GPS serán capaces de obtener localizaciones precisas de menos de 30 centímetros de error tanto en latitud como en longitud.

Estos problemas, no pueden solucionarse completamente mediante software, debido a que es problema del propio sensor y su inexactitud. Pero, pueden buscarse soluciones que mitiguen o permitan una implementación de estos sensores de geolocalización para el uso en realidad aumentada.

Solución planteada al problema de inexactitud para la Altitud

En esta aplicación, nos vemos obligados a descartar la altitud que nos proporcionan los sensores de localización para añadir imágenes en el espacio. Una imprecisión de 30 a 50 metros en resultados precisos es demasiado elevada como para poder normalizarse en un rango de tiempo menor a un minuto.

Una posible solución para este problema sería, que el usuario, al iniciar la aplicación introdujera la altura a la que está situado el dispositivo móvil. Luego, utilizando el acelerómetro se podría calcular la altura relativa partiendo desde esta posición, calculando la suma de todas las aceleraciones que se van produciendo en cada uno de sus ejes en el dispositivo durante la ejecución de la aplicación.

Pero esta solución posee un fallo. Cada vez que el acelerómetro nos da un resultado, este posee un **margen de error**. Si sumamos las distancias recorridas utilizando el acelerómetro, este error irá incrementándose mientras la aplicación siga funcionando. De esta forma, va a llegar un momento (dependiendo de la cantidad de movimientos verticales que realice el usuario con el dispositivo), donde el error acumulado sea tan elevado, que una Instancia de Imagen situada por el usuario, pueda llegar a mostrarse a una altitud errónea superior a 1 o más metro de la ubicación real.

Al no encontrarse una solución válida que obtenga la altitud del dispositivo de forma automática, se ha optado por añadir un campo regulable por el usuario, que le permita modificar manualmente la altura en la que está situado el dispositivo.

Solución a la inexactitud de la Latitud y la Longitud

Como pasa en la altitud, en el caso de la latitud y longitud existe un margen de error en la exactitud de sus ángulos. Pero, por el otro lado, el margen de error es más reducido, siendo de aproximadamente 5 metros para resultados con precisión elevada.

Al ser el margen de error más pequeño, en este caso, la aplicación va a encargarse de normalizar o calcular de alguna forma, la localización exacta del dispositivo.

Para esto, se plantean dos maneras simples de realizarlo: usando una media aritmética o usando una media aritmética ponderada.

La media aritmética se calculará usando las **últimas veinte localizaciones** para conocer la posición exacta del dispositivo. Usar la media, nos permite conocer una posición más exacta, debido a que los márgenes de error de los diferentes resultados se anulan entre ellos. El problema de esta solución es que no permite desplazamientos rápidos. La aplicación puede llegar a tardar unos 30 segundos en actualizar completamente la posición del usuario, debido a la lentitud en la que se actualizan estas localizaciones al querer tanta precisión.

La otra solución, es el uso de una **media aritmética ponderada**, basándose en el mismo principio que la solución anterior, esta realizará una media entre las últimas veinte localizaciones recibidas. Esta, dará más peso a las últimas localizaciones obtenidas por el sensor, aumentando la precisión en la que se actualizará la localización si el usuario realiza un desplazamiento. El problema, es que es menos exacta que la media aritmética.

La aplicación, implementa los dos tipos de soluciones, para que el usuario pueda elegir la que prefiera. El usuario, también puede elegir la opción de obtener la última localización, por si quiere desplazarse más deprisa, a costa de la exactitud.

Siendo L una lista de latitudes o longitudes de las últimas t localizaciones obtenidas y L(1) la latitud y o longitud más reciente y L(t) la más antigua, la fórmula usada para el cómputo de la media aritmética ponderada en la aplicación es:

$$lat/long = \frac{\sum_{n=1}^{t} L(n) * ((t+1) - n)}{\sum_{i=1}^{t} i}$$

Ecuación 1. Fórmula usada para la media aritmética ponderada de la latitud y longitud

Conversión de distancia angular a metros

En esta aplicación es necesaria una fórmula capaz de transformar la distancia angular de la latitud y la longitud a metros. Esto se debe tanto a la necesidad de que el usuario pueda insertar imágenes a una cierta distancia del dispositivo, como a que el Módulo de Realidad Aumentada debe calcular distancias relativas entre el usuario y las distintas Instancia de Imagen en el entorno.

Primero de todo, se debe entender que existe un margen de error al calcular distancias usando una conversión de la latitud y longitud real a sistema métrico. Al usar este método algebraico para la conversión de la distancia angular a metros, se presupone que la superficie terrestre es una esfera perfecta con un radio igual en toda su superficie. Esta es una afirmación errónea, el radio terrestre no es uniforme. Debido a esto, de dos distancias angulares idénticas no sé obtiene dos distancias métricas exactas.

El método que se va a explicar a continuación, se vuelve ineficaz para el cálculo de posiciones a distancias muy largas. Pero, para distancias cortas donde la superficie

terrestre posee las mismas características, esta aproximación es lo suficientemente precisa como para que el usuario no pueda notar la diferencia.

Por ejemplo, una imagen que se añada a 20 metros del usuario usando esta conversión, puede crearse con un margen de error de 36,8 centímetros de la posición correcta. Una imagen situada a un metro posee aproximadamente un margen de error de 2 a 3 centímetros.

Para esta fórmula, primero es necesario calcular que distancia angular corresponde a un metro.

Par realizar este cálculo, supondremos que la Tierra forma una esfera perfecta, siendo su radio, la media de la distancia desde el centro del planeta hasta la superficie (6.378.137 metros).

$$Radianes\ por\ metro = \frac{1}{\left(\left(1º * \frac{\pi}{180}\right) * \ RadioTerrestre\right)}$$

Ecuación 2. Fórmula de conversión de la distancia angular terrestre a metros

Esta fórmula da como resultado 0.000008998 radianes por metro. Simplificando un poco el cálculo, esto supondría que 1º de distancia angular terrestre se corresponde a 111,11 kilómetros.

Para el cálculo de la latitud (en radianes) se debe dividir su distancia angular por este coeficiente:

$$latitud\ en\ metros = \frac{latitude}{radianes\ por\ metro}$$

Ecuación 3. Fórmula de conversión de la latitud de radianes a metros

En el caso de la longitud, el radio varía proporcionalmente a la altura o distancia desde el centro terrestre.

Figura 21. Ejemplo de la disminución del radio longitudinal al acercarse a los polos

Por ese motivo, tenemos que reducir proporcionalmente el radio dividiéndolo por el coseno de la latitud en radianes:

$$longitud\ en\ metros = \frac{longitud * radianes\ por\ metro}{Coseno\ (latitud * radianes\ por\ metro)}$$

Ecuación 4. Fórmula de conversión de la longitud a metros

Módulo de Localización

Desde este momento, llamaremos al Módulo de Localización, a la implementación de la clase *LocationService* de la aplicación.

Esta clase se creó con la ayuda de la documentación de Android Developers para el uso de sensores de localización.

El objetivo de este módulo, es obtener de forma autónoma, la localización del dispositivo en tiempo real y guardarla dentro de las propiedades del usuario para que el resto de la aplicación pueda usarla.

La implementación del Módulo de Localización está formado por una sola clase cuyo nombre es *LocationService*. Esta clase, se encarga mediante el uso de un tipo de Listener llamado *LocationListener* y un *LocationManager*, de generar interrupciones en el Thread principal del programa cuando se obtiene una nueva localización para el dispositivo.

Para usar este módulo, y poder conseguir localizaciones usando el sensor de localización es necesario obtener previamente, los permisos de localización por parte del usuario.

Inicialización del Módulo de Localización

Al crear este módulo, se crea tanto un *LocationListener* como un *LocationManager*. Estas dos clases son necesarias para el uso de los sensores de localización en dispositivos Android.

Un *LocationListener* consiste en un Listener u oyente, capaz de lanzar interrupciones al proceso principal con una frecuencia determinada para obtener la localización actual del dispositivo. Como cualquier Listener, es necesario implementar o sobrescribir, una serie de métodos como el *onLocationChanged* que se ejecutara, al obtenerse una nueva localización.

Un *LocationManager*, es un tipo de clase que puede enlazar un *LocationListener* junto a unos criterios o propiedades para inicializar la monitorización de la localización. Este también permite a la aplicación iniciar, parar y reanudar las actualizaciones de la localización.

Para modificar los atributos con los que se ejecuta esta monitorización, será necesario crear una instancia de la clase *Criteria***.** Esta clase permite modificar el funcionamiento y preconfigurar el *LocationManager* antes de su uso.

Los criterios o atributos usados por esta aplicación son:

1. **Precisión Elevada**: Permite a la aplicación obtener localizaciones más precisas a costa de reducir la frecuencia de veces en las que se obtienen.
2. **Gasto energético elevado**: Permite aumentar los recursos usados por el dispositivo para el cómputo de la localización aumentando también la batería consumida.
3. **Deshabilitar el uso de la altitud**. Ya que no usaremos la altitud debido a su imprecisión, al deshabilitarla, aumenta la frecuencia de actualización de la localización.

Una vez establecidos estos criterios, será necesario decidir qué tipos de geolocalización usará el *LocationManager*. En Android, existen dos tipos de cómputo para la localización del dispositivo. El primer tipo de cómputo es el ***Fine Location*** o de **Localización**

Precisa, este usa una combinación de GPS***,*** GSM *y* Wifi para la obtención de la localización más precisa posible. En segundo lugar, tenemos la ***Coarse Location*** o de *Localización Aproximada*, este sistema usa solo la geolocalización por GSM *y* Wifi para obtener una localización aproximada del dispositivo. La segunda opción, no necesita GPS y, por lo tanto, no requiere los permisos de usuario pasa su uso y reduce significativamente el coste en recursos y batería.

Ya que nuestra aplicación necesita una precisión lo más exacta posible, la aplicación usa la combinación de GPS, GSM, y Wifi usando el cómputo de localización por **Fine Location**.

Por último, para completar la configuración inicial, es necesario definir en qué ***frecuencia de tiempo*** **mínima** nos va a proporcionar el *LocationListener* una nueva localización. Debido, a que nuestra aplicación necesita actualizar la localización constantemente, vamos a definir un tiempo mínimo de 50 milisegundos entre actualizaciones.

Al completar toda esta configuración inicial, se ejecuta el método de la clase *LocationManager, requestLocationUpdates****.*** Esté método enlaza el *Criteria* con el *LocationListener* para iniciar la monitorización de la localización del dispositivo.

Actualizaciones de la localización

AL inicializarse el Módulo de Localización y ejecutarse el método *requestLocationUpdates* del *LocationManager****,*** nuestra aplicación empezará a recibir localizaciones del dispositivo. Estas localizaciones se recibirán mediante el método implementado en el *LocationListener, onLocationChanged*. Este método se llamará cada vez que se calcule una nueva localización para el dispositivo. Esta localización vendrá representada en una instancia de clase *Location*,

Un objeto de la clase *Location* contiene la información de la latitud, longitud y altitud de la localización obtenida, junto a parámetros muy importantes como la precisión del resultado y la antigüedad de este.

Antes de guardar la localización en el Modelo de Datos se filtrarán las localizaciones demasiado imprecisas. Para el filtrado, se ejecutará el método *isBetterLocation* de nuestra clase *LocationService* para decidir si va a guardarse la nueva localización o descartarse. Si se guarda, se actualizará la localización la información del usuario en el Modelo de Datos, volviéndose a calcular tanto la media aritmética como la media aritmética ponderada de las últimas 20 localizaciones.

Esquema de decisión para el filtrado de localizaciones

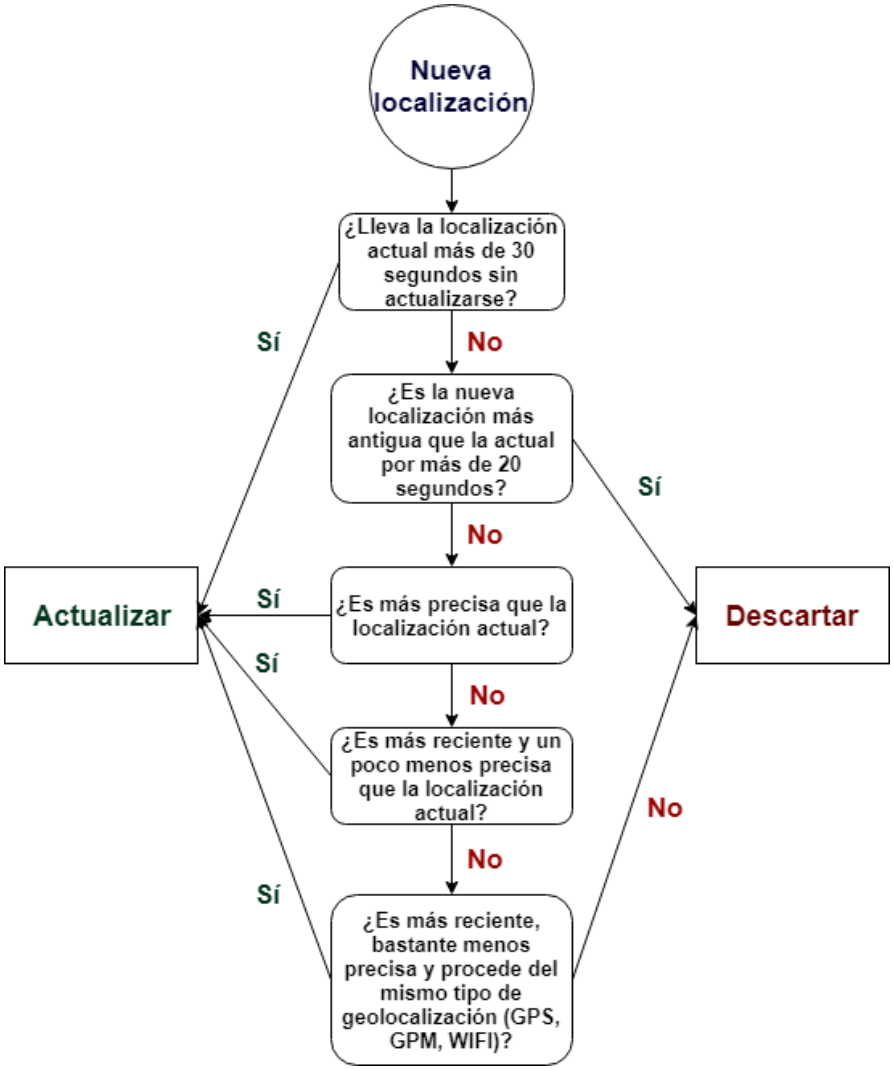

Figura 22. Esquema de decisión para el filtrado de localizaciones

Ciclo de vida

El Módulo de Localización se inicializa en el método *onResume* de la actividad principal y se pausa en el método *onPause* de la misma. La aplicación debe liberar los sensores de localización antes de cerrarse o pausarse. El menú deslizante de la aplicación implementa un botón, que permite habilitar y deshabilitar la geolocalización del dispositivo.

Conceptos generales sobre rotaciones y sensores de rotación

Antes de pasar a la implementación del Módulo de Rotación, es necesario conocer y asimilar los conceptos que van a introducirse en este capítulo.

Una rotación, es un movimiento de cambio de orientación de un cuerpo sobre un eje de rotación, donde, un **eje de rotación**, es una línea recta con respecto a la cual una figura geométrica puede rotar:

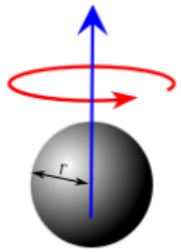

Figura 23. Representación de una esfera rotando sobre el eje de rotación azul

Cualquier rotación de un objeto situado en un espacio en tres dimensiones, puede representarse como una rotación en los tres ejes de coordenadas del espacio:

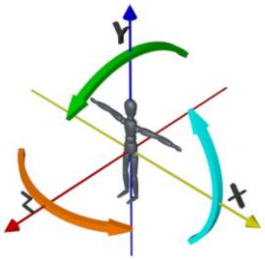

Figura 24. Rotación en un eje de coordenadas de tres dimensiones

Existen múltiples maneras para representar una rotación en el espacio. Siendo la más común, la representación de una rotación en Ángulos de Euler.

Los Ángulos de Euler constituyen un conjunto de tres coordenadas angulares usadas para especificar la orientación de un sistema de referencia de ejes ortogonales, normalmente móviles, respecto a otro sistema de referencia de ejes ortogonales normalmente fijos.

Por ejemplo, siendo el sistema de **coordenadas móvil xyz** y el **origen común o fijo XYZ**, es posible especificar la posición de un sistema en términos del otro usando tres **ángulos α, β y γ**:

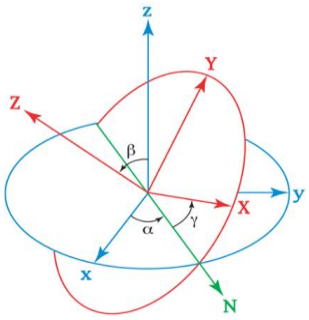

Figura 25. Imagen de dos ejes de coordenadas ortogonales uno azul y el otro rojo, superpuestos

Este sistema de rotaciones fue comúnmente usado para dispositivos móviles, debido a la capacidad de representar la orientación del dispositivo mediante el uso de estos tres ángulos. Para su uso, estos ángulos reciben los nombres de **Yaw** (Guiñada) como α, **Pitch** (Cabeceo) como β y **Roll** (Alabeo) como γ:

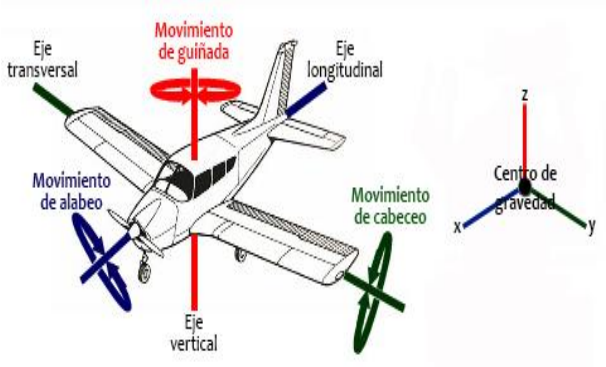

Figura 26. Ejemplo de rotación usando Ángulos de Euler

Una misma rotación en Ángulos de Euler puede poseer más de una combinación de ángulos válida. Por lo tanto, para aplicar una rotación a un objeto, existe más de una combinación de **Pitch**, **Yaw** y **Roll** que la genere.

Representación de rotaciones en computación: matrices y cuaterniones

En el siglo 20, los Ángulos de Euler fueron el sistema más popular para representar rotaciones, ya que eran bastante simples de modelizar y entender. Para su implementación, se usaban matrices de rotación tridimensionales.

Las matrices, son un sistema muy usado para la representación de objetos en espacios tridimensionales, donde una matriz, puede determinar la localización de un punto en el espacio, el movimiento de un objeto, su rotación, etc. A todas las operaciones capaces de crear una nueva figura a partir de una previamente dada se les llama **transformaciones geométricas**. Para más información sobre estas transformaciones se recomienda la lectura de este documento.

Todo el sistema gráfico de OpenGL se basa en el uso de matrices y estas transformaciones geométricas para crear y maquetar el entorno virtual.

Una **matriz de rotación** es una matriz que representa una rotación en el espacio euclídeo. Esta matriz, al multiplicar otra matriz, como, por ejemplo, la representación de un vértice, le aplica su rotación. Esta rotación es acumulativa. Esto significa que esta matriz no rota el objeto o vértice a una posición fija del espacio, sino que aplica o añade una cierta rotación cada vez que se multiplica por este.

Aproximadamente hace veinte años, era bastante común el uso de Ángulos de Euler y matrices de tres dimensiones para la representación de rotaciones. Estos dejaron de usarse debido a un error o fallo que poseen este tipo de rotaciones llamado el Bloqueo de Cardán. Este problema aparece cuando se usan ángulos de Euler en matemática aplicada, como, por ejemplo, modelado 3D, sistemas de navegación o en mi caso, aplicaciones de realidad aumentada. El Bloqueo de Cardán consiste en la pérdida de un grado de libertad cuando los ejes de dos de los tres rotores de un giroscopio se colocan en paralelo, bloqueando el sistema en una rotación en un espacio bidimensional. Con bloquear, nos referimos a que al producirse esta condición existe un eje sobre el que ninguno de los rotores puede girar.

A efectos prácticos, esto significa que no todo cambio en el espacio de rotaciones puede ser expresado como un cambio en el espacio de los Ángulos de Euler y por lo tanto, existen rotaciones que los Ángulos de Euler no puede representar.

Para solucionar este problema se crearon los cuaterniones.

Un cuaternión, es una notación matemática que sirve para representar orientaciones y las rotaciones de objetos en tres dimensiones. Comparados con los Ángulos de Euler, son más simples de componer y evitan el problema del Bloqueo de Cardán. Estos son muy útiles en aplicaciones de gráficos por computadora y han sido necesarios para implementar la rotación en esta aplicación.

Un cuaternión se define como un objeto de cuatro dimensiones tal que:

$$q = w + x * i + y * j + z * k$$

Ecuación 5. Estructura de un cuaternión

Donde, x, y, z y w son números reales que representan las coordenadas de cuatro dimensiones, siendo w una cuarta dimensión que nos permite evitar el problema del Bloqueo de Cardán. La i, j y k son unidades imaginarias definidas por medio de un sistema de igualdad:

$$i^2 = j^2 = k^2 = i * j * k = -1$$

Ecuación 6. Fórmula de igualdad de un cuaternión

Este sistema de igualdad permite, por ejemplo, eliminar toda la dimensión imaginaria w permitiendo rotar un vector de solo tres dimensiones. Para aplicar cualquier tipo de rotación en OpenGL será necesario transformar el cuaternión a una matriz de cuatro dimensiones.

Debido a la extensión del tema, esta memoria no va a incluir la explicación de cómo operar rotaciones usando este tipo de notación matemática. Si desea informarse y aprender sobre el uso de cuaterniones en rotaciones espaciales, léase este artículo.

Combinaciones de sensores para el cómputo de rotaciones en Android

Para calcular la rotación del dispositivo en el espacio, es necesaria la combinación de múltiples sensores. Los sensores usados para el cálculo de la rotación son:

1. **Acelerómetro**: Permite el cálculo de la aceleración en cada uno de los tres ejes (x, y, z) incluyendo la fuerza de la gravedad.
2. **Magnetómetro**: Mide el campo magnético aplicado a cada uno de los tres ejes físicos del dispositivo.
3. **Giroscopio**: Mide la ratio de rotación del dispositivo en radios/segundo para cada uno de los tres ejes físicos.

Programar en Android nos permite, el uso, tanto por separado de estos sensores como usar agrupaciones de estos para calcular la rotación del dispositivo.

Los dos tipos de agrupaciones existentes para el cálculo de la rotación son:

1. **Sensor de Orientación**: combinación del magnetómetro y el acelerómetro para el cálculo de rotación. Este sistema utiliza los Ángulos de Euler para obtener el resultado.

2. **Sensor de Rotación**: combinación del giroscopio, magnetómetro y acelerómetro para el cálculo de la rotación. Este sistema utiliza un cuaternión para el obtener el resultado.

El Sensor de Rotación fue introducido en la API 9 de Android para reemplazar al **Sensor de Orientación** del dispositivo. Actualmente, casi todas las aplicaciones Android son producidas usando el Sensor de Rotación. Esto se debe a que este presenta tres mejoras significativas respecto al **Sensor de Orientación**. La primera, se debe al uso de cuaterniones en vez del uso de Ángulos de Euler para guardar la orientación del dispositivo, evitando el problema de Bloqueo de Cardán. La segunda mejora, consiste en el incremento de precisión en el cálculo de la rotación al usar el sensor de giroscopio para el cómputo de la rotación. Finalmente, la tercera mejora se debe a que el uso de cuaterniones es más eficiente computacionalmente que el uso de Ángulos de Euler.

Uso de la rotación en Realidad Aumentada

En esta aplicación, la rotación del dispositivo se requiere principalmente para tres funciones:

1. Obtener la rotación a donde está mirando la cámara para poder mostrar Instancias de Imagen por pantalla.
2. Calcular la localización de una nueva Instancia de Imagen.
3. Obtener la orientación con la que se insertan nuevas Instancias de Imagen.

Para cumplir estos objetivos, nuestra aplicación usa el Sensor de Rotación para obtener la rotación en tiempo real. Esto significa, que, para la creación de estas funcionalidades, como punto de partida, poseemos un cuaternión con la rotación actual del dispositivo.

Esta rotación, no puede ser directamente aplicada a nuestra aplicación. Esta rotación posee diversos problemas o características para poder aplicarlo directamente a este proyecto. Por este motivo, veo necesario abordar todos los problemas que se presentaron en implementar un sistema de rotación en una aplicación de realidad aumentada.

Problemas de vibración y desviación de la rotación

La rotación obtenida por el Sensor de Rotación no es perfecta. En las rotaciones existen márgenes de error que causan que se perciba una vibración de la visualización del entorno virtual. También, si se zarandea el dispositivo, al volver a situarlo a una misma orientación, la rotación después de zarandearlo no será la misma que antes de hacerlo.

El problema de la vibración se puede solucionar mediante el uso de filtrados, reglas y coeficientes para regular los cambios producidos en la orientación del sensor. El problema del zarandeo, aunque es imposible solucionarlo completamente, es posible mitigar bastante sus efectos mediante el filtrado de rotaciones con una aceleración demasiado elevada.

El volumen de trabajo para obtener coeficientes, para normalizar y filtrar estas rotaciones, posee tal cantidad de pruebas y tiempo de testeo, como para ser un proyecto completamente separado de este. Debido a este motivo, y al existir ya versiones o implementaciones que solucionan estos errores, basé casi por completo mi código de rotación al usado por Alexander Pacha en su estudio sobre la normalización de rotaciones en dispositivos Android. Este estudio es de código libre y puede usarse sin ninguna restricción.

Su proyecto implementa un sistema de rotación que compara entre sí, múltiples maneras de calcular la rotación en un dispositivo móvil. Siendo la mejor de ellas y la implementada en este proyecto, una de las más exactas que puede conseguirse actualmente con los sensores de rotación en dispositivos móviles. Recomiendo al lector encarecidamente, la descarga de la app "Sensor fusión" de **Alexander Pacha** en la Play Store, que implementa ejemplos gráficos sobre múltiples formas de calcular la rotación en el dispositivo.

Diferencia entre los ejes de rotación del dispositivo

En un dispositivo móvil, los ejes de coordenadas se corresponden a los de la siguiente imagen:

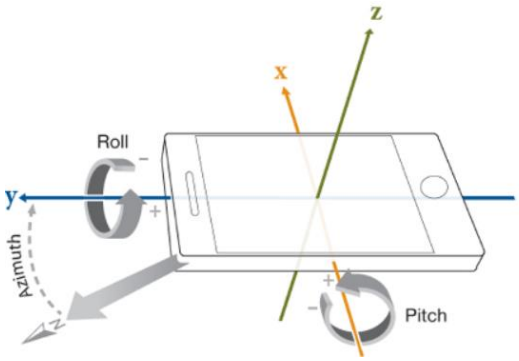

Figura 27. Ejes de rotación en un dispositivo móvil

Comparando la Figura27 con la Figura 26, se observa como la parte delantera del avión se corresponde con la parte de arriba del dispositivo. Esto resulta problemático. Para la aplicación de realidad aumentada, nosotros necesitamos que la rotación o el **eje frontal del dispositivo** sea el eje que sale de la cámara, el **z**. Si intentáramos rotar un objeto o matriz por este cuaternión, se visualizaría la dirección donde apunta el **eje y** del dispositivo, mientras que nosotros deseamos mostrar la dirección apuntando el **eje z**.

Para solucionar este problema, se plantearon **dos soluciones**; **la primera**, **normalizar o rotar el** cuaternión de tal forma que el eje frontal del dispositivo fuera el que saliera de la cámara; **la segunda opción**, y la más sencilla, era aplicar la rotación del dispositivo a un punto situado encima del eje de rotación z y **calcular un vector director**.

Debido a la complejidad que suponía normalizar un cuaternión y al ser la primera vez que los utilizaba, los problemas que podía ocasionar, ya fuera invertir rotaciones no deseadas o degradar el sistema debido a un error acumulativo en la normalización, hicieron que descartara esta solución. Pero, existía otra solución más simple, el calcular un vector usando dos puntos, que representara la dirección a la que apuntaba la cámara del dispositivo.

Un vector director, es un vector que da la dirección de una recta en un sentido determinado. En este caso, queremos obtener un vector con la misma dirección a la que apunta la cámara trasera del dispositivo o lo que es lo mismo, en el eje de coordenadas **z** de la Figura 27.

Para calcular este vector, son necesarios dos puntos en tres dimensiones. En este apartado, un punto se representará como (x, y, z), siendo x, y, y z la posición del punto en sus respectivas dimensiones.

El primer punto que usaremos para calcular el vector director, es el situado en el origen de coordenadas o la posición del dispositivo: (0, 0, 0). Para calcular el segundo punto,

se necesita rotar por el cuaternión un punto en el eje z situado delante de la posición de la cámara del dispositivo, a una unidad 1 de distancia: (0, 0, 1).

De esta forma obtendríamos un vector director, formado por dos puntos:

(0, 0, 0) -> (0, 0, 1) * cuaternión

Al rotar este punto desde el centro del dispositivo, obtendríamos otro punto a la misma distancia del origen de coordenadas con una nueva dirección.

Obtención de la rotación de la cámara

Para obtener la rotación de la cámara, la cual usa OpenGL para conocer a dónde mira el dispositivo en el espacio virtual, es tan simple como transformar el vector director actual del dispositivo a una matriz de rotación.

Existe un método, de la clase *Matrix*, llamado *lookAt*, que dados dos puntos que forman un vector director, calcula una matriz de rotación de cuatro dimensiones que sirve para rotar la cámara hacia esa dirección en el espacio.

Obtención de la posición de una imagen al crearla

Al usuario añadir una Instancia de Imagen, es necesario poder conocer la rotación del dispositivo para calcular a que distancia del usuario tanto en latitud, longitud y altura, está situada la Instancia de Imagen.

Para situar un objeto a una **distancia de X metros del usuario**, la aplicación aplica el mismo sistema para el cómputo del vector director. Necesitamos rotar un punto a una distancia **X en el eje z** o de la cámara: **(0, 0, X).** Al rotar este punto por el cuaternión, conseguiremos un punto en unas posiciones **(x, y, z)** cuya distancia es **X** del usuario. Dependiendo de la rotación del dispositivo cada una de estas coordenadas podrá ser positiva o negativa.

Debido a que el Sensor de Rotación está pensado para funcionar conjuntamente con la geolocalización, el eje de coordenadas de la rotación posee el mismo ángulo que el eje de coordenadas de la geolocalización terrestre. Siendo **X** el eje que va de Oeste (-) a Este (+), **Z** el eje que va de Sud (-) a Norte (+) e **Y** el eje en altitud.

Por lo tanto, a efectos prácticos, las coordenadas en el **eje Z** del cuaternión de rotación se corresponden a la distancia **en latitud**, las de **X**, a la distancia **en longitud** y las de **Y**, a la distancia **en altitud**.

Siendo **I** la nueva imagen, **U** la localización del usuario y **P** el punto calculado de rotar otro punto (0, 0, X) por el cuaternión de rotación, para el cálculo de la localización de una nueva Instancia de Imagen se utilizan estas **tres fórmulas**:

$$I.latitud = U.latitud + P.z(latitud)$$

$$I.longitud = U.longitud + P.x(longitud)$$

$$I.altitud = U.altura + P.y(altitud)$$

Ecuación 7. Fórmulas para el cómputo de la localización al crear nueva Instancia de Imagen

U.altura representa la altura en metros que ha introducido el usuario en el menú deslizante de la aplicación.

Obtención de la rotación de una imagen al crearla

Al crear una Instancia de Imagen, esta se guarda en el Modelo de Datos con una rotación determinada. La rotación de esta imagen vendrá dada por el **vector director contrario al del dispositivo**. En caso de que el usuario se desplace por el entorno, esta Instancia de Imagen conservará tanto la posición como la rotación con la que se ha creado. Esto sirve para mejorar el efecto de realidad aumentada de la aplicación, debido a que las imágenes se transforman o deforman respecto a la posición del usuario como si de un objeto real se tratara. Para conseguir este efecto, es necesario guardar la rotación en la que el usuario ha insertado la instancia de la imagen.

Para conseguir esta rotación, es tan simple como obtener el vector director de la cámara e invertirlo. De tal forma que se obtiene un vector director invertido, que apunta desde la Instancia de Imagen hasta la cámara.

Por ejemplo, si tenemos un vector formado por un punto **(0, 0, 0)** con dirección al punto **(0, 0, 1)**, es tan simple como invertir el orden de los puntos tal que el punto **(0, 0, 1)** se dirija al punto ***(0, 0, 0)***.

A la práctica, usaremos el método *lookAt* de la clase *Matrix* que, mediante el vector director invertido, nos dará la matriz de rotación en cuatro dimensiones de la Instancia de Imagen. Cabe remarcar, que rotar un objeto utilizando *lookAt*, solo es aconsejable para rotar objetos estáticos a los que no se les modifica la rotación gradualmente.

Implementación del Módulo de Rotación

Para la implementación del Módulo de Rotación, se han utilizado dos clases; la primera, llamada *OrientationProvider*, posee todos los campos y métodos básicos para activar el Sensor de Rotación en el dispositivo; la segunda, *OrientationSensorProvider*, es una clase que hereda de la primera y contiene toda la capacidad de normalizar y filtrar los datos de los sensores para guardar su resultado en el Modelo de Datos.

A partir de este momento, llamaremos Proveedor de Rotación a la implementación de la clase *OrientationSensorProvider* y de este, explicaremos tanto las funcionalidades propias, como las heredadas de la clase *OrientationProvider* para simplificar la lectura.

Inicialización del Proveedor de Rotación

La lógica del Proveedor de Rotación está implementada dentro del Presentador. Al crearse el Presentador, este crea una instancia de esta clase.

Esta clase solo posee dos métodos públicos: *start*, que inicia la monitorización del Sensor de Rotación y *stop*, que pausa esta monitorización.

El Proveedor de Rotación implementa la interface *SensorEventListener*. Este Listener permite mediante el uso de interrupciones, obtener información en tiempo real de los sensores que se hayan preconfigurado.

Para preconfigurar y poder usar el *SensorEventListener*, es necesario crear una instancia de la clase *SensorManager*. Un *SensorManager*, implementa un método llamado *registerListener*, que permite enlazar un *SensorEventListener* junto a un tipo de sensor e iniciar la monitorización de este.

Para poder iniciar esta monitorización de un sensor usando el método *registerListener*, es necesario establecer qué tipo de sensores va a usar el *SensorEventListener* y con qué frecuencia de tiempo van a actualizar su información.

En nuestra aplicación, al ejecutarse el método *start*, **se ejecutará dos veces** el método *registerListener*. **La primera** vez, para **enlazar el Sensor de Giroscopio** y **la segunda**, **para el** Sensor de Rotación (combinación de los sensores de giroscopio, acelerómetro y magnetómetro).

El uso del Sensor de Giroscopio por separado, nos permite mejorar el sistema de filtrado de la rotación y detectar ciertos acontecimientos que no sería posible detectarse con solo usar el Sensor de Rotación.

Debido a que nuestra aplicación necesita actualizaciones de rotación constantes para aumentar la fluidez con la que se muestran las Instancias de Imagen, vamos a enlazar estos dos sensores junto al *SensorEventListener* con un tiempo de actualización o refresco de unos 20 milisegundos. Esto permite que el usuario no pueda percibir cambios de rotación bruscos en la pantalla del dispositivo.

Actualización del Proveedor de Rotación

Al haber inicializado el *SensorEventListerner* junto a un grupo de sensores. El hilo principal del programa empezará a recibir interrupciones. Estas interrupciones ejecutarán un método de la interface *SensorEventListener* llamado *onSensorChanged*.

Este método devolverá la información en tiempo real de todos los sensores enlazados a este *SensorEventListener:* el Sensor de Rotación y el Sensor de Giroscopio. Por defecto, se ejecutará una vez cada 20 milisegundos automáticamente.

El método *onSensorChanged* posee como parámetro de entrada una instancia de un *SensorEvent*. Una instancia de la clase *SensorEvent*, contiene la información sobre que sensor ha producido el evento y cuál es la información actual del sensor.

Al ejecutarse este método, los datos obtenidos tanto por el Sensor de Rotación como por el Sensor de Giroscopio, servirán para calcular un cuaternión normalizado de la rotación actual del dispositivo. Este cuaternión es un atributo de la propia clase y se actualizará con los resultados obtenidos por los dos tipos de sensores. Después de actualizarse, se guardará este cuaternión normalizado dentro del Modelo de Datos, para que pueda usarse en el resto de la aplicación.

Debido a la complejidad de la normalización y al no estar creada por mí, no se va a explicar su cálculo en esta memoria. Para obtener toda la información respecto a esta implementación, lea la documentación del proyecto de Alexander Pacha.

Finalización o apagado del Proveedor de Rotación

Al ejecutarse el método *stop* del Proveedor de Rotación se ejecutará por cada sensor monitorizado, el método *unregisterListener*. Este método desconecta las actualizaciones automáticas del tipo de sensor que se le indique.

En nuestro caso, este método se ejecuta dos veces, una para eliminar las notificaciones automáticas del Sensor de Rotación y la otra para el Sensor de Giroscopio.

Ciclo de vida

Igual que todos los otros módulos dentro del Presentador, la monitorización del Proveedor de Rotación se inicializa en el método *onResume* de la actividad principal y para al ejecutarse el método *onPause* de la misma. Por lo tanto, en todo momento en que la aplicación esté ejecutándose en primer plano la aplicación va a monitorizar la rotación del usuario. En este caso, y al contrario que con el Módulo de Localización, el usuario no puede interrumpir la monitorización del Proveedor de Rotación manualmente.

Módulo de Cámara

En este apartado, llamaremos Módulo de Cámara a la implementación de la clase *CameraView* de la aplicación.

El Módulo de Cámara es el encargado de mostrar por pantalla, todas las imágenes que obtenga el Sensor de Imagen del dispositivo en tiempo real. Este módulo no necesita ninguna información del Presentador ni del Modelo de Datos para su funcionamiento.

Para el uso del Módulo de Cámara, es necesario anteriormente poseer los permisos de usuario correspondientes. En caso contrario, la aplicación no podrá mostrar imágenes por pantalla.

También, el Módulo de Cámara necesita el identificador de la *TextureView* donde va a mostrar las imágenes obtenidas del Sensor de Imagen en la pantalla del dispositivo. Una *TextureView* es un tipo de vista, capaz de mostrar imágenes de manera sucesiva en las dimensiones que ocupa sobre la pantalla. Esta *TextureView*, como cualquier vista, se crea en XML y se le asigna un identificador único para que las aplicaciones sean capaces de utilizarla o enlazarse a ella.

En el caso de esta aplicación, la *TextureView* ocupa toda la pantalla y está situada en la última posición en profundidad, de tal forma que cualquier vista quede solapada encima de esta.

El Módulo de Cámara hereda de la clase Thread, de tal forma que se ejecuta en un hilo de ejecución separado del principal debido al coste computacional de actualizar constantemente la imagen. También usa la ***librería Camera2*** para la gestión y uso del Sensor de Imagen trasero del dispositivo. Esta librería, sustituye a la librería *Camera* ya deprecada.

Inicialización del Módulo de Cámara

Preparación de la vista de la cámara

Al crearse una instancia de esta clase es necesario pasarle por parámetros el identificador de la *TextureView* donde se van a mostrar las imágenes.

Para poder usar un *TextureView* se necesita crear un *SurfaceTextureListener*. Este Listener, controla la creación y eliminación de la vista al entrar o salir de la actividad en la aplicación.

Para poder configurar este *TextureView* al inicializarse se va a sobrescribir el método *onSurfaceTextureAvailable* para poder inicializar y enlazar la cámara a la vista. Este método está dividido en tres pasos.

El primero, es **obtener el identificador del Sensor de Imagen** trasero del dispositivo: Mediante el uso de una instancia de *CameraManager* vamos a poder obtener el listado de Sensores de Imagen del dispositivo junto a sus características dentro de una clase de la instancia *CameraCharacteristics*. Filtrando por estas características, vamos a ser capaces de obtener el identificador del Sensor de Imagen trasero.

La clase *CameraManager* consiste en la clase principal de gestión de Sensores de Imagen del dispositivo. Esta clase permite tanto preconfigurar la cámara como empezar la monitorización del Sensor de Imagen. La clase *CameraCharacteristics* contiene todos los atributos o rasgos de un Sensor de Imagen. Por ejemplo, de esta podemos obtener el ángulo de visión, la posición de la cámara, su resolución, etc.

El segundo, es **obtener el tamaño de visualización del Sensor de Imagen** trasero. Usando el identificador del Sensor de Imagen trasero y mediante el uso de una instancia de *StreamConfigurationMap* creada con las *CameraCharacteristics* del dispositivo, vamos a ser capaces de obtener el tamaño o las dimensiones de la matriz de píxeles en las que el Sensor de Imagen crea la imagen para el dispositivo

Finalmente**, el tercero**, consiste en **transformar el tamaño y ángulo del Sensor de Imagen para que se adapte al *TextureView***. Para hacerlo, será necesario crear una matriz con las dimensiones del *TextureView* que reescale el tamaño de las imágenes obtenidas por el Sensor de Imagen debido a que este obtiene las imágenes en una resolución normalmente superior a la pantalla del dispositivo. También, en caso de que el Sensor de Imagen este **rotado** un ángulo de **90º** o **270º** será necesario rotar esta matriz para que al mostrar las imágenes por pantalla posea la orientación correcta.

Al finalizar estos tres pasos y al haber creado esta matriz de reescalado, se enlazará al *TextureView* usando el método de su clase, *setTransform*. De esta forma, cada vez que se introduzca una nueva imagen al *TextureView*, se le aplicará el reescalado y rotación de esta matriz automáticamente.

Obtención e inicialización del Sensor de Imagen

Para poder obtener imágenes del Sensor de Imagen es necesario usar una instancia del *CameraManager*. Esta clase posee el método *openCamera* que permite enlazar un Sensor de Imagen a un *StateCallback*. Un *StateCallback*, es una clase que mediante uso de callbacks permite establecer que acciones se llevarán a cabo al crearse, destruirse o fallar la conexión de la aplicación con el Sensor de Imagen. Al conectarse al Sensor de Imagen correctamente, se va a ejecutar su método *onOpened.* Dentro de este, mediante el uso de múltiples clases y pasos de la librería *Camera2*, se enlazará el *TextureView* al Sensor de Imagen, de tal forma, que monitorice y actualice automáticamente la imagen mostrada en el TextureView de la cámara trasera del dispositivo.

Actualización del Módulo

Este módulo no requiere ningún dato o información de clases externas a esta que se modifique en tiempo de ejecución. Debido a este motivo, no existe la necesidad de cambiar el funcionamiento de refresco de imágenes predeterminado de la librería *Camera2*. Por lo tanto, el Módulo de Cámara realiza el refresco de imagen automáticamente sin la intervención del código de la aplicación.

Suspensión del Módulo de Cámara

Debido a la implementación del *StateCallback* y el *SurfaceTextureListener*, el apagado se realiza automáticamente al detectarse el cierre de la conexión con el Sensor de Imagen o de la *TextureView*. La instancia de *StateCallback* implementada en este módulo llamará al método *onDisconnected* al detectar el cierre de la actividad principal, encargándose de cerrar la conexión de la aplicación con el Sensor de Imagen. El *SurfaceTextureListener*, realizará la misma acción usando el método *onSurfaceTextureDestroyed* en caso de que no se ejecute el método anterior.

Ciclo de Vida

El Módulo de Cámara funcionará en todo momento sin interrupciones. Este módulo empezará la monitorización en el método *onResume* de la actividad y se cerrará automáticamente mediante el uso de callbacks al pausarse esta. El usuario no puede pausar manualmente la visualización de la cámara del dispositivo en la aplicación.

Capítulo 8: Módulo de Realidad Aumentada

En este capítulo llamaremos Módulo de Realidad Aumentada a la implementación de las clases *VirtualCameraView*, *GLRender* y *GLPicture* en la Vista de la aplicación.

Este módulo se encarga de mostrar por pantalla todas las Instancias de Imagen cercanas al usuario. Al solapar la vista del Módulo de Cámara debajo de la vista del Módulo de Realidad Aumentada se genera la visión de realidad aumentada en la pantalla del dispositivo.

La información externa de la aplicación que necesita este módulo para su funcionamiento es:

1. Una *GLSurfaceView* donde se enlazará este módulo para mostrar las imágenes por pantalla.
2. Localización y rotación del dispositivo del usuario.
3. Lista con las localizaciones, las rotaciones y los Bitmaps de las Instancias de Imagen cercanas al usuario.

Una *GLSurfaceView* es un tipo de vista capaz de mostrar fotogramas del entorno virtual en OpenGL. Este módulo actualiza la imagen de forma automática de tal forma que la aplicación solo tiene que encargarse de encender y apagar esta visualización cuando es necesario.

Conocimientos básicos de OpenGL y OpenGL ES

Este capítulo solo pretende resumir al lector el funcionamiento de OpenGL ES. Para ampliar la información expuesta en este subcapítulo le recomiendo al lector leer el libro "OpenGL ES 2 for Android" por Kevin Brothaler. Este libro es un perfecto punto de partida para lectores sin conocimientos previos de OpenGL.

OpenGL(Open Graphics Library) es una librería estándar que sirve para escribir aplicaciones que produzcan gráficos en dos o tres dimensiones. Esta librería **permite dibujar escenas tridimensionales complejas** a partir de primitivas geométricas simples tales como puntos, líneas y triángulos. **Mediante el uso de matrices de cuatro dimensiones** es capaz de transformar el aspecto de este espacio tridimensional. Esta librería oculta y simplifica la complejidad de implementar programas que usen gráficos en dispositivos con diferentes tarjetas gráficas.

OpenGL ES consiste en una variante simplificada de OpenGL diseñada para dispositivos integrados tales como teléfonos móviles. Esta implementación reduce y mejora el uso de recursos en este tipo de dispositivos.

El apartado gráfico de esta aplicación fue **creado** usando **OpenGL ES 2.0**.

A grandes rasgos, OpenGL ES permite modificar o añadir diferentes tipos de elementos en el espacio. Estos elementos son:

1. **Objetos**: Llamaremos objeto, a toda estructura formada por vértices ya sea una línea o cualquier estructura geométrica compleja en el espacio. A estos objetos pueden añadírseles texturas o colores.
2. **Cámaras**: OpenGL ES permite cambiar el sentido o la rotación a la que apunta la cámara en el entorno o espacio gráfico. Al contrario que otras implementaciones de OpenGL, OpenGL ES no permite desplazar o mover la cámara del origen de coordenadas del espacio.

3. **Luces**: Al insertar una luz en una posición del espacio esta iluminará todo el entorno virtual cercano. Esta luz puede proyectarse de diferentes formas, ya sea simulando la luz del Sol en el entorno, un foco o una esfera.
4. **Efectos**: En OpenGL se pueden implementar sombras, niebla para ocultar objetos lejanos u otros efectos gráficos.

OpenGL usa un sistema de **coordenadas en tres dimensiones X, Y**, y **Z** para representar la posición de los objetos en el espacio virtual. El centro de este espacio está situado en el eje de coordenadas.

Debido a que en esta aplicación no es necesaria la implementación de luces o efectos gráficos, solo se van a explicar las características de los objetos y de la cámara virtual.

Objetos en OpenGL y OpenGL ES

Para poder crear un objeto, ya sea un cuadrado o un triángulo en OpenGL, es necesario situar los vértices que componen al objeto en el espacio y enlazarlos. Por ejemplo, para definir un rectángulo es necesario crear cuatro vértices, uno a cada esquina de la figura geométrica. En OpenGL ES no es posible crear una figura en tres dimensiones automáticamente. Todas las figuras en tres dimensiones se componen de agrupaciones de figuras geométricas en dos dimensiones. Por ejemplo, un cubo se corresponde al conjunto de seis cuadrados iguales rotados en direcciones opuestas desde su centro. Y una esfera, en una agrupación de distintos rectángulos o triángulos de distinto tamaño y orientación:

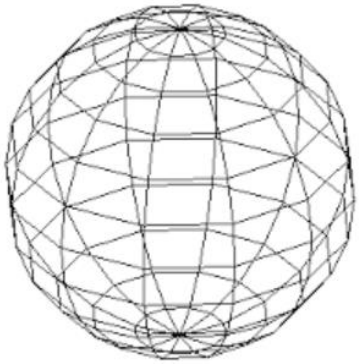

Figura 28. Ejemplo de una esfera formada por rectángulos y triángulos bidimensionales

En OpenGL ES todos los objetos se crean uniendo triángulos entre sí, al contrario que en OpenGL donde se permite la creación de objetos en dos dimensiones con múltiples combinaciones de vértices directamente. Por ejemplo, en OpenGL ES, un rectángulo se corresponde a dos triángulos unidos por su hipotenusa.

Para poder desplazar y rotar estos objetos, es necesario el uso de matrices de cuatro dimensiones. Al multiplicarse por los distintos vértices, estas matrices cambian la posición en que el vértice se posiciona en el espacio. Así pues, calculando la posición relativa de todos los vértices y aplicándoles las mismas transformaciones geométricas, podremos desplazar un objeto sin modificar su aspecto o proporciones iniciales.

Cámara Gráfica en OpenGL y OpenGL ES

Una cámara en OpenGL, consiste en un punto de visión donde se observa el espacio o entorno en una dirección determinada. En OpenGL esta cámara puede moverse por el espacio y modificar la rotación o la dirección en la que observa el espacio virtual.

En OpenGL ES, este punto de visión no puede moverse y o trasladarse. Esto significa que la cámara no puede desplazarse por el espacio. En caso de querer simular movimiento, es necesario recalcular la posición de todos los objetos respecto a este punto de visión. En OpenGL ES, la cámara gráfica o el punto de visión está situado en el centro u origen de coordenadas del espacio gráfico. Este origen de coordenadas se represente con el punto (0x, 0y, 0z).

Existen distintas formas de ver el espacio virtual desde el punto de visión. A estas formas de mostrar el espacio virtual por pantalla, se les llama proyecciones. En OpenGL se pueden aplicar múltiples tipos de proyecciones, por ejemplo, la proyección ortogonal, la proyección Frustum o la Proyección en Perspectiva. En el caso de OpenGL ES, se necesita crear una matriz de proyección de uno de estos distintos tipos de proyecciones y aplicarla a todos los objetos que quieran mostrarse en la pantalla del dispositivo.

Uso de OpenGL ES para Realidad Aumentada

En este apartado se exponen los problemas y soluciones aportadas para implementar el sistema de realidad aumentada usando OpenGL ES en la aplicación.

Unión de la vista real y la vista virtual

Para poder implementar un sistema de realidad aumentada es necesario que todas las imágenes u objetos representados en OpenGL se muestren encima de la imagen a tiempo real que capta la cámara del dispositivo.

Para conseguir este cometido, es necesario que la vista de la cámara del dispositivo esté situada a una prioridad o capa de visión inferior que la vista de OpenGL mapeando la vista de la actividad con XML.

Pero todo no es tan simple, una vista en OpenGL posee un fondo opaco. Esto significa que, si solapamos las dos vistas, la visión a tiempo real de la cámara quedará escondida por debajo de la visión del fondo opaco de OpenGL.

Para solucionar este problema existen dos soluciones. La primera y la implementada en esta aplicación, consiste en transparentar el fondo de OpenGL de tal forma que todas las capas inferiores a la vista usada por el Módulo de Realidad Aumentada puedan mostrarse. La segunda solución, consiste en traspasar y transformar la imagen a tiempo real de la cámara como fondo de OpenGL. De tal forma que, OpenGL obtenga constantemente las imágenes de la cámara y las repinte en el fondo del escenario.

El problema de la segunda opción es posee demasiados problemas: el primero, debido al coste computacional necesario para que OpenGL se encargue del repintado de la cámara; y el segundo, que usar el Sensor de Imagen de manera manual, fotograma a fotograma, no es sencillo y puede generar errores y problemas de refresco en la imagen.

La solución de transparentar el fondo en OpenGL, parece la mejor. Pero posee una problemática muy grave, para poder transparentar esta imagen no es posible hacerlo de manera normal con alguna función o parámetro de OpenGL ES. Para poder transparentar el fondo mediante OpenGL ES, es necesario sacar la vista usada por OpenGL de la **Jerarquía de Vistas** de la actividad y situar la vista en frente de cualquier vista que pueda crearse en la aplicación. La **Jerarquía de Vistas** es la estructura en XML encargada de solapar la visión de las distintas vistas por pantalla para obtener la imagen final mostrada al usuario.

Esto plantea uno de los problemas principales de la aplicación: debido que la vista de OpenGL esta por sobre el resto de las otras en la actividad principal, también se sitúa

por encima de la vista encargada de la interacción con el usuario, sin permitir mostrar los elementos como los botones de interacción. Este problema no pudo solucionarse, debido a que no existe otra forma en OpenGL ES que permita transparentar la cámara sin causarlo.

Por este motivo, cuando se muestra el menú de interacción con el usuario, es necesario parar el Módulo de Realidad Aumentada para poder mostrar los distintos botones sin ninguna Instancia de Imagen encima de estos.

Unión de los ejes de coordenadas reales y virtuales

Al representar objetos en el espacio es necesario adaptar o situar el eje de coordenadas reales a las virtuales.

El **sistema de coordenadas real** está formado por los ejes ortogonales de **latitud**, **longitud** y **altitud**. El **sistema de coordenadas en OpenGL** también está formado por tres ejes ortogonales **X**, **Y**, y **Z**. Al ser estos ejes ortogonales en ambos casos, se puede representar cada una de las coordenadas reales con uno de los ejes de coordenadas virtuales.

En esta aplicación, el eje de coordenadas real de la altitud corresponde al eje de coordenadas virtual Y, el de longitud al eje X y el de latitud al eje Z.

Pero esta representación posee un problema: a efectos prácticos, la orientación de la latitud es inversa a la orientación del eje de coordenadas virtual Z.

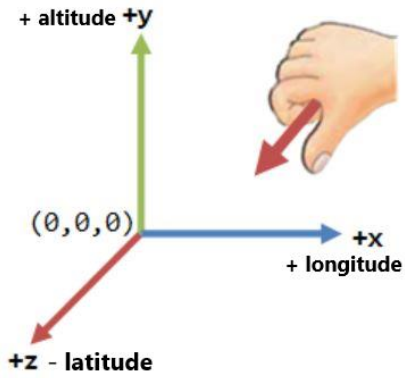

Figura 29. Comparativa entre el sistema de coordenadas real y el sistema de coordenadas virtual

Por lo tanto, una latitud positiva equivaldría a su representación negativa en el eje de coordenadas Z. De manera simple, si una Instancia de Imagen se añadiera más al Norte del usuario, OpenGL la dibujaría a más distancia en dirección Sud.

La manera más simple de solucionar este problema, es negar o invertir la distancia en latitud tanto de la posición de las Instancias de Imagen o el usuario, como del vector director de ambos. Si una Instancia de Imagen está situada a una latitud positiva (más al Norte del usuario), le pasaremos a OpenGL su latitud negada. En caso de la rotación, también se modificará la posición en latitud a la que apunta el vector director del dispositivo.

Creación del entorno y mapeado virtual cercano al Usuario

OpenGL necesita conocer la posición de las Instancias de Imagen respecto al dispositivo en el entorno virtual.

Debido a que el punto de vista en OpenGL ES es estático y no puede moverse por el espacio, es necesario crear todo el entorno respecto a la posición del dispositivo. Por este motivo, para la creación del entorno virtual, se calcula la distancia de las Instancia de Imagen respecto al dispositivo en cada uno de los ejes de coordenadas.

Esta distancia relativa entre el dispositivo y las diferentes Instancia de Imagen situadas en el entorno cercano al usuario, se calcula restando la latitud, longitud y altitud del dispositivo, respecto a las homónimas de estas instancias.

Al obtener las posiciones a las que se encuentran las distintas Instancia de Imagen respecto al usuario, se deben transformar estas posiciones a su equivalente en OpenGL.

Para transformar una posición real a una virtual, es necesario tanto especificar la equivalencia entre un metro y una unidad en OpenGL como elegir a que eje virtual se corresponde cada uno de los ejes reales (ya establecidos en el apartado anterior).

En esta aplicación, una unidad en OpenGL en uno de sus ejes de coordenadas, representará la distancia real de un metro en uno de los ejes reales.

Por ejemplo, una distancia relativa de 3 metros en longitud, 2 metros en latitud y 1 metro en altitud (3Long, 2Lat, 1Alt) de una imagen respecto al dispositivo, se correspondería a la distancia virtual (3x, 2z, 1y) en OpenGL.

Transformación de la rotación real a la virtual

Al rotar un objeto o la cámara del usuario en OpenGL ES, es necesario transformar o modificar la posición de los vértices de las Instancias de Imagen dentro del entorno. Para rotar un objeto o la cámara virtual en OpenGL ES es necesario una matriz de rotación de cuatro dimensiones para cada uno de las Instancias de Imagen y otro para la cámara. La obtención de estas matrices ya se explicó en este subcapítulo.

Con el uso del vector director del dispositivo, somos capaces de crear casi directamente las rotaciones en OpenGL. El único paso necesario para transformar la rotación real a OpenGL, es negar la latitud del vector director antes de crear las matrices de rotación.

La explicación del motivo de realizar este paso se puede encontrar en este apartado.

Normalización del ángulo de visión virtual al del Sensor de Imagen

Al situar una imagen en el entorno virtual, es necesario que aumente o reduzca su tamaño al acercarse o alejarse de ella. Una Instancia de Imagen en OpenGL, tiene que empequeñecerse igual que un objeto real al alejar el dispositivo de ambos una misma distancia.

De esta forma, la aplicación tiene que conseguir, que una imagen virtual reduzca o aumente su tamaño de la manera más similar posible a la que lo hacen los objetos reales captados en la lente de la cámara del dispositivo.

Tanto el ojo humano como la cámara poseen un campo de visión, este campo de visión se ve afectado por el ángulo de visión que posea la lente con la que se observa el entorno real. El campo de visión es el espacio que abarca la visión de una lente y a un ángulo de visión mayor, el campo de visión aumenta, permitiendo ver más objetos en su periferia:

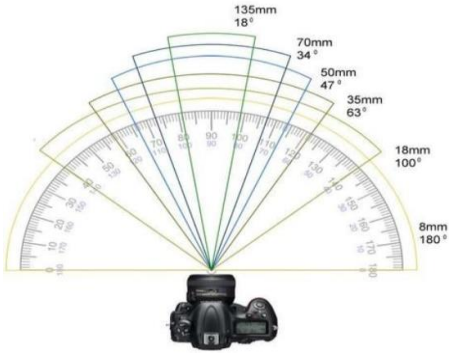

Figura 30. Comparativa entre ángulos de visión y sus respectivos campos de visión

Para poder simular este efecto, es necesario un tipo de proyección o visualización del entorno en OpenGL que permita simular en cualquier dispositivo, el ángulo de visión de la cámara para generar un campo de visión de un tamaño similar al real.

En OpenGL, existe un tipo de proyección llamada Proyección en Perspectiva. Este tipo de proyección es el usado por esta aplicación y permite dado un ángulo de visión para el eje Y, y un ratio de visión entre la anchura y la altura de la pantalla del dispositivo, crear un campo de visión muy similar a un campo de visión real. En esta proyección también se especifica a que distancia cercana se empiezan a observar los objetos y a que distancia lejana dejan de observarse. De esta forma, en cada una de estas distancias se formará un plano de visión que delimita los límites del campo de visión virtual:

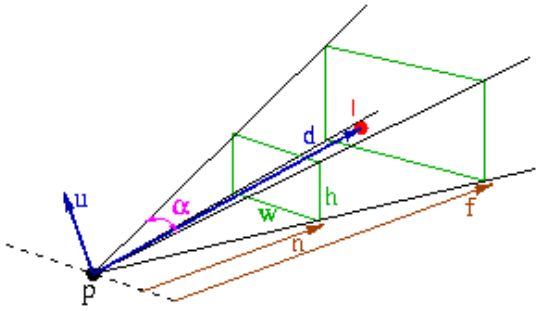

Figura 31. Ejemplo de Proyección en Perspectiva en OpenGL

Variables	Definición	Valor usado en la aplicación
p	Punto de visión	Siempre está situado en el origen (0x, 0y, 0z)
d	Vector director	Vector director de la rotación del dispositivo actual
l	Centro de enfoque	- (por defecto)
u	Dirección hacia arriba	El eje real de altitud o el eje virtual Y
n	Distancia al plano anterior	0.001 unidades en OpenGL (1 unidad = 1 metro)
f	Distancia al plano posterior	100 unidades en OpenGL (1 unidad = 1 metro)
r = w/h	Ratio de aspecto del dispositivo	w = anchura en píxeles de la pantalla del dispositivo h = altura en píxeles de la pantalla del dispositivo
α	Ángulo de visión en el eje Y del dispositivo	Ángulo obtenido del eje Y de la cámara usada por el dispositivo menos 15º

Tabla 3. Configuración del campo de visión virtual de la aplicación

Como podemos observar en la anterior tabla, fue necesario reducir el ángulo de visión para calibrar el campo de visión virtual de la aplicación. Cada dispositivo móvil puede poseer un ángulo de visión distinto en la cámara trasera, y, por lo tanto, es necesario obtener su valor de las características de la cámara.

En caso de usar la cámara trasera del dispositivo Xiaomi A2 Lite, el ángulo de visión vertical de la cámara es de 52.1º.

El problema de usar directamente el ángulo vertical de la cámara del dispositivo radica a que el Módulo de Cámara realiza una transformación de las imágenes reales para que quepan dentro de la pantalla, si aplicamos el ángulo de la cámara directamente, el campo de visión será bastante superior al que podemos observar realmente en la pantalla del dispositivo.

Por ese motivo, mediante pruebas de calibrado usando objetos reales junto a instancias de imágenes de dimensiones aproximadas, se calculó usando trigonometría, un ángulo para calibrar aproximadamente el campo de visión del dispositivo. El ángulo que dio mejores resultados fue el resultante de restar 15 grados al ángulo vertical de la cámara del dispositivo.

Esta aplicación no asegura que todos los dispositivos posean una calibración del campo de visión correcta por la cantidad de variaciones tanto de Sensores de Imagen y dimensiones de pantalla en el mercado móvil actual.

Mejoras de rendimiento aplicadas en la implementación de OpenGL

Una mala configuración de OpenGL ES puede causar que la visión virtual no se vea de manera fluida o, que ocupe más memoria de la necesaria. En esta aplicación se implementan una manera de reducir el coste en memoria del guardado de texturas y otra manera de reducir el tiempo de cómputo al mostrar el entorno virtual por pantalla.

Reducción del número de texturas cargadas

En OpenGL ES, antes de crear un objeto, es necesario precargar texturas. Cada dispositivo, dependiendo de sus recursos, posee un número limitado de texturas para cargar en memoria. Esto significa, que es necesario llevar un control de las texturas usadas y las repetidas entre diferentes instancias de imagen cercanas al usuario.

Para realizar ese control de texturas, esta aplicación usa una Tabla de Hash. El funcionamiento de este es muy similar al sistema de caché de imágenes usado en el Modelo de Datos. Este caché, dado el Path a la imagen de la Instancia de Imagen que se quiere mostrar, devuelve un identificador de la textura si ya está cargada en memoria. En caso contrario, se carga la textura de esta imagen y se guarda en esta Tabla de Hash.

Para controlar el número de imágenes en memoria, eliminamos todas las texturas cargadas cada vez que se actualiza la lista de imágenes cercanas al usuario. Este sistema podría mejorarse para filtrar texturas de imágenes que no van a eliminarse. Pero, el coste en tiempo de eliminar las texturas una por una por filtrado, es bastante parecido al coste en tiempo necesario para volver a cargar las texturas de los Bitmaps ya precargados en memoria.

Reducción del tiempo de cómputo en la carga de imágenes

Para poder mostrar las instancias de imagen fluidamente por pantalla a una buena tasa de fotogramas por segundo, es necesario reducir al máximo el tiempo de computación necesario para el pintado de las Instancias de Imagen.

En el Modelo de Datos, ya se redujo el coste de obtención de las Instancias de Imagen cercanas al usuario, de tal forma que la Vista pueda en cualquier momento obtener una lista de las Instancias de Imagen actualizada. También se añadió una variable booleana que permitía identificar si se había modificado esta lista en algún momento.

Para poder pintar una imagen por pantalla en OpenGL ES, se crea un objeto o clase que posee todas las características y variables necesarias para pintar una Instancia de Imagen por pantalla.

El problema de crear estos objetos, radica en que, si creáramos tantos objetos como Instancias de Imagen cercanas al usuario cada vez que repintemos la pantalla, los fotogramas por segundo que puede mostrar la aplicación se reducirán drásticamente.

Como ya se mencionó en el Modelo de Datos, una Instancia de Imagen es estática. El único valor que puede cambiar de una Instancia de Imagen es la distancia relativa entre ella y el usuario, de tal forma que si este se aleja de la Instancia de Imagen la distancia relativa entre estos aumenta.

De esta forma, aprovechando que las Instancias de Imagen son estáticas, la aplicación solo creará estos objetos cuando se actualice la lista de imágenes próximas al usuario. También, en el momento de repintar las imágenes, solo será necesario crear un método en esta clase capaz de pintar una Instancia de Imagen dada una distancia relativa.

Aplicando esta forma de repintado de imágenes, utilizando el dispositivo Xiamo Mi A2, la aplicación fue capaz de mostrar más de 300 imágenes por pantalla de forma fluida y sin bajadas notables en los fotogramas por segundo. Al actualizar la lista de imágenes, esta implementación tardaba un poco menos de medio segundo en volver a cargar el mismo número de imágenes.

Implementación del Módulo de Realidad Aumentada

Para usar OpenGL ES en su versión 2 o 2.0 en Android, es necesario que el dispositivo sea compatible con sus clases. La implementación de esta clase en Android se corresponde a la clase *GLES20*. Esta clase usa métodos estáticos para crear el entorno virtual de la aplicación. La aplicación también usa la clase *GLUtils*. La clase *GLUtils* complementa a la clase *GLES20* para el cargado de texturas.

Para implementar el Módulo de Realidad Aumentada fue necesaria la creación de tres clases distintas:

1. **Controlador de Virtualización** o *VirtualCameraView***:** esta clase era la encargada de conectar, iniciar y parar la visualización virtual.
2. **Gestor de Refresco** o *GLRenderer***:** clase encargada del repintado de todo el entorno o las Instancias de Imagen en OpenGL.
3. **Instancia de Imagen en OpenGL** o *GLPicture*: Esta clase es creada a partir de una Instancia de Imagen y posee todos los métodos y variables para poder dibujarla por pantalla utilizando OpenGL ES 2.0.

En los próximos subcapítulos se explicará de manera resumida, la implementación de cada uno de estos apartados.

Implementación del Controlador de Virtualización

Llamaremos Controlador de Virtualización a la implementación de la clase *VirtualCameraView* en la aplicación.

Esta clase es la encargada de conectar el Módulo de Realidad Aumentada junto al resto de la aplicación. Permite tanto inicializar la visualización de las Instancias de Imagen como pararla. También incluye múltiples métodos para calcular distancias y rotaciones, necesarias para todo el Módulo de Realidad Aumentada para OpenGL ES.

Al crearse una instancia de esta clase, se le pasará una *GLSurfaceView* por parámetro de entrada. Una *GLSurfaceView* es un tipo de vista capaz de mostrar imágenes creadas usando OpenGL. En el método constructor de esta clase, se enlazará esta *GLSurfaceView* con el Gestor de Refresco. Este Gestor de Refresco se encargará de refrescar o actualizar la imagen de la vista cada vez que haya renderizado o pintado un nuevo fotograma.

En este constructor también se especifica la transparencia del fondo en OpenGL usando múltiples métodos, siendo el más importante, el método de la *GLSurfaceView, setZOrderOnTop*. Este método sitúa la vista virtual encima de la vista usado por el Módulo de Cámara.

También en este constructor, se especifica el tipo de renderizado de la aplicación. Existen **dos tipos de renderizados** en OpenGL: **por petición** y **continuo**. El **renderizado por petición** hace que la aplicación no repinte por pantalla la *GLSurfaceView* hasta que la aplicación se lo notifique al Gestor de Refresco. Al contrario, usando el **renderizado continuo**, la aplicación estará siempre actualizando de manera constante los fotogramas de la *GLSurfaceView* sin necesidad de informar al Gestor de Refresco.

Debido a que esta aplicación por cualquier cambio en la rotación, localización o en la lista de imágenes próximas al usuario necesita actualizar el renderizado, al existir tantos cambios en un corto margen de tiempo, **la aplicación usará el renderizado continuo** sin ningún sistema de notificaciones.

Implementación del gestor de refresco

Nombraremos Gestor de Refresco a la implementación de la clase *GLRenderer*. Esta clase es la encargada de pintar automáticamente por pantalla, las Instancias de Imagen cercanas al usuario.

Esta clase implementa la interface *GLSurfaceView.Renderer*. Esta interface, permite conectar el Gestor de Refresco a una *GLSurfaceView*. Esta interface implementa los tres métodos principales para el funcionamiento de una aplicación en OpenGL:

1. ***onSurfaceCreated***: este método se ejecuta al conectarse el Gestor de Refresco a la *GLSurfaceView* y se encarga de inicializar los campos necesarios para el funcionamiento de OpenGL. Dentro de este método se habilita el uso de texturas y el uso de doble búfer.
2. ***onSurfaceChanged***: este método se ejecuta al conectarse el Gestor de Refresco con la *GLSurfaceView* y también se ejecuta al cambiar el tamaño de la *GLSurfaceView*. Es el encargado de crear, la matriz de la Proyección en Perspectiva. Esta matriz se crea con el método de la clase *Matrix*, *perspectiveM* usando la configuración especificada en la Tabla 3.
3. ***onDrawFrame***: este método se ejecuta continuamente y es el encargado de pintar todo el escenario y las distintas Instancias de Imagen en OpenGL.

Al repintar la imagen en el método *onDrawFrame*, se siguen múltiples pasos.

El primer paso consiste en crear la matriz de la rotación actual del dispositivo. Al multiplicar esta matriz por la matriz de Proyección en Perspectiva obtendremos una matriz capaz de simular la visión real de la cámara del dispositivo. A esta matriz la llamaremos Matriz de Visualización.

En segundo lugar, en caso de que se haya actualizado la lista de instancias de imágenes próximas al usuario, se eliminaran las texturas antiguas guardadas en OpenGL y se crearán, a partir de estas Instancias de Imagen nuevos objetos o instancias de la clase *GLPicture*.

En tercer y último lugar, la aplicación se encargará de pintar cada una de las instancias de *GLPicture* por pantalla. Antes de pintarlas, se recalculará para cada *GLPicture*, la distancia relativa de cada una de ellas hasta el dispositivo. También se les pasará como parámetro de entrada la Matriz de Visualización.

Implementación de una Instancia de Imagen en OpenGL

Para poder guardar o pintar objetos en OpenGL se recomienda crear una clase para cada uno de ellos. En el caso de esta aplicación, solo poseemos un tipo de objeto para pintar que se corresponde a una Instancia de Imagen en OpenGL. Esta clase recibirá el nombre de *GLPicture*. Este nombre será usado en todo este subcapítulo.

La clase *GLPicture* permite dibujar la Instancia de Imagen a la que representa en OpenGL ES. Esta clase posee un método llamado *draw* que al pasarle por parámetros de entrada una posición y una Matriz de Visualización dibuja la Instancia de Imagen por pantalla.

Esta clase también implementa una Tabla de Hash de texturas que sirve como caché para guardar las texturas de las imágenes en memoria. En este subcapítulo llamaremos a este hash, **Caché de Texturas**.

Para poder crear una *GLPicture* es necesario en el **constructor** pasarle estos parámetros obtenidos de la Instancia de Imagen a la que van a dibujar:

1. **El Path a la imagen**: Este Path sirve para guardar la textura de la Instancia de Imagen en el **Caché de Texturas**. En caso de que este Path este usado, significa que esta textura ya está cargada en memoria.
2. **Rotación de la** Instancia de Imagen: La matriz de rotación de la Instancia de Imagen con la que la creó el usuario.
3. **Bitmap de su imagen**: imagen cargada en memoria a la que apunta el Path de la imagen del punto **1**. OpenGL, no puede pintar directamente Bitmaps, y por lo tanto, en caso de que la textura no esté ya pre-cacheada, se creará una nueva textura de OpenGL a partir de este Bitmap.
4. **Ratio de pixeles por centímetro**: Esta ratio se asigna a la Instancia de Imagen al crearse e indica el tamaño que va a tener cada pixel en 0.01 unidades en OpenGL. Recordemos que la correspondencia en metros de unidades en OpenGL es de uno a uno.

Usando todos los anteriores parámetros se preconfigurará todos los parámetros que necesita la *GLPicture*. Para poder dibujar estas imágenes en la vista usada por OpenGL, será necesario crear un rectángulo, con las dimensiones de la imagen. Asignando la propia textura de la imagen a este rectángulo conseguiremos pintar en la vista una imagen 2D en OpenGL.

En el constructor de la clase *GLPicture*, se configurarán todos los campos necesarios para el pintado de este rectángulo junto a la textura de la imagen.

El primer paso para hacerlo, consiste en **crear un búfer de memoria con los vértices que van a componer este rectángulo**. La aplicación siempre crea las imágenes en el centro de OpenGL. Estas imágenes serán posteriormente trasladadas en el método *draw* de la misma clase.

De esta forma, vamos a crear un array de los cuatro vértices formados por cada una de las esquinas de este rectángulo. Cada uno de estos vértices es representado por un punto en tres dimensiones. Por ejemplo, el vértice situado en la esquina superior derecha de una imagen se corresponde a:

$$Vértice = \left(-\left(\frac{\text{ancho de la imagen en pixeles}}{2}\right)x, \left(\frac{\text{altura de la imagen en pixeles}}{2}\right)y, 0\ \text{z}\right)$$

Ecuación 8. Cómputo del vértice superior izquierdo de una GLPicture

Al conseguir todos los cuatro vértices de una imagen, multiplicaremos cada una de las tres dimensiones de estos vértices por el **ratio de pixeles por centímetro** de la Instancia de Imagen. Aplicando estos pasos, obtendremos los vértices de un rectángulo centrado al origen de coordenadas, con las proporciones de la imagen y con un tamaño proporcional al ratio de píxeles por centímetro. Finalmente, guardaremos este array en un búfer de memoria, para que OpenGL pueda consultarlo.

En segundo lugar, OpenGL ES necesita **crear otro búfer con la información del orden de dibujado de estos vértices**. No basta con crear solamente los vértices, es necesario especificar como van a conectarse entre ellos y en qué orden. Para eso, es necesario crear un array de números con el orden de pintado a seguir dentro del array de vértices. En OpenGL ES, solo se pueden unir vértices mediante estructuras triangulares. Esto significa que tenemos que crear este rectángulo a partir de dos triángulos:

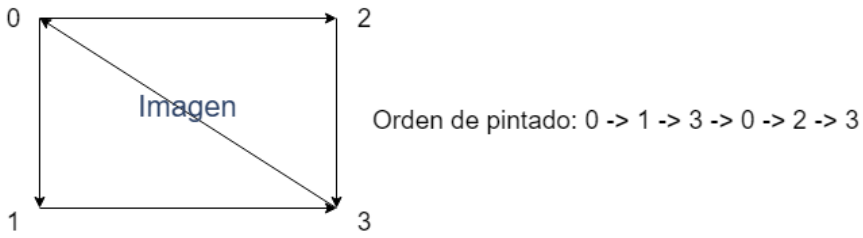

Figura 32. Representación del orden de pintado de un rectángulo usando dos triángulos en OpenGL ES

Como podemos observar en la Figura 32, los vértices de estos dos triángulos son los mismos cuatro vértices que forman un rectángulo normal.

En tercer lugar, es necesario **crear la textura de la Instancia de Imagen u obtenerla si ya existe dentro del Caché de Texturas**. Para crear una textura en OpenGL ES, es necesario reservar un identificador para la textura. Estos identificadores servirán para que las *GLPicture* puedan acceder y usar la textura cargada en memoria.

Antes de cargar el Bitmap dentro de estos identificadores, es necesario especificar ciertas características de la textura. La primera, es capacitar a la textura para que pueda adaptarse a cualquier tamaño. La segunda característica, consiste en deshabilitar el degradado o reducción de la resolución de la textura en distancias lejanas. Aunque la segunda característica puede aumentar el rendimiento de la aplicación debido al

procesado de menos píxeles, en esta aplicación se desea que la calidad o realismo de la imagen virtual sea la más precisa a cualquier distancia.

Para acabar de configurar esta textura, vamos a cargarla en OpenGL utilizando el Bitmap de la Instancia de Imagen. Finalmente, también va a ser necesario crear un búfer con un array del orden de dibujado de la textura. Cambiar o modificar el orden de dibujado de la textura erróneamente puede invertir o rotar el dibujo de la imagen cargada.

Finalmente, como **cuarto paso** es necesario **crear un Programa**.

Un Programa es una clase interna usada por la clase *GLES20* para especificar que transformaciones van a aplicarse a los vértices y a la textura antes de pintarse por pantalla. Por ejemplo, en nuestra aplicación necesitamos que a cada vértice se le aplique la Matriz de Visualización, la rotación de la propia Instancia de Imagen y la traslación a la posición relativa de la imagen respecto al usuario.

Para poder realizar estas operaciones dentro del Programa, es necesario enlazarlo a un Shader de Vértices y a un Shader de Fragmentos. A grandes rasgos, un Shader de Vértices es el subprograma que se ejecutará para calcular la posición final de un vértice antes de pintarlo. Un Shader de Fragmentos es otro subprograma que calcula el color de cada pixel dentro del rectángulo de la imagen. Estos subprogramas están escritos en el lenguaje de programación C.

En ambos códigos en C se definen múltiples variables. Estas variables no se inicializan dentro de estos dos códigos. Estas variables de ambos subprogramas, se enlazan con las matrices creadas en Java de la aplicación para modificar la posición y la rotación de la imagen cada vez que se repinta por pantalla.

Para calcular la posición y rotación final de un vértice por pantalla, la aplicación usa esta fórmula:

$$posVertFinal = MV * (posVert * MR * MT)$$

Ecuación 9. Cómputo de la posición final de un vértice dentro del Shader de Vértices

Variable	Descripción
posVertFinal	**Posición final del vértice** al aplicar todas las transformaciones matriciales
MV	**Matriz de Visualización** calculada en el Gestor de Refresco
posVert	**Posición inicial del vértice.** Uno de los vértices guardados en el búfer de vértices de la *GLPicture*)
MR	**Matriz de Rotación** de la Instancia de Imagen
MT	**Matriz de Translación** creada a partir de la distancia relativa entre el usuario y la Instancia de Imagen

Tabla 4. Descripción del cómputo de un vértice dentro del Shader de Vértices

En caso del Shader de Fragmentos, el propio subprograma asignará a cada pixel del rectángulo, el color del pixel de la textura de la imagen que le corresponde. En caso de que quisiéramos cambiar el color de las imágenes o aplicarles sombras, tendríamos que modificar el cómputo de color de cada pixel en el código de este subprograma.

Finalmente, cuando todas estas características ya se hayan configurado dentro de la *GLPicture*, el Gestor de Refresco ya estará capacitado para usar o ejecutar el método de dibujado *draw*.

El método *draw*, es un método público de la clase *GLPicture* que se encarga de dibujar la Instancia de Imagen a la que representa el *GLPicture* por pantalla. Para poder pintar la imagen por pantalla, este método necesita tanto la Matriz de Visualización como la distancia relativa actual de la Instancia de Imagen al usuario. Estos dos valores los pasará por parámetro de entrada el Gestor de Refresco.

Al iniciarse el método, seleccionaremos el Programa que hemos creado anteriormente en el constructor de la clase.

Seguidamente, calcularemos la matriz de traslación del objeto usando la distancia relativa de la Instancia de Imagen respecto al usuario.

En tercer lugar, seleccionaremos mediante el identificador de textura de la propia *GLPicture*, la textura ya cargada para OpenGL que queremos pintar por pantalla.

En cuarto lugar, enlazaremos todas las matrices necesarias para el cómputo de la posición final de los vértices del rectángulo con las variables del subprograma del Shader de Vértices.

Finalmente, ejecutando el método de *GLES20*, *glDrawElements*, dibujaremos por pantalla la Instancia de Imagen usando los subprogramas asignados al Programa de la instancia de *GLPicture*.

Ciclo de Vida del Módulo de Realidad Aumentada

El Módulo de Realidad Aumentada se inicializa en el método *onResume* de la actividad principal de la aplicación y se pausa, en el método *onPause* de la misma actividad. También, debido al problema de solapamiento de las imágenes virtuales encima de la vista del menú deslizante, este módulo se deshabilita al abrir el menú deslizante y se vuelve a iniciar al cerrarse.

El usuario no puede deshabilitar manualmente el Módulo de Realidad Aumentada de la aplicación.

Capítulo 9: Manual de Usuario

Instalación

1. Descargue la APK de la aplicación desde este enlace.
2. Guarde el archivo APK en el dispositivo móvil donde se quiere instalar la aplicación.
3. Una vez guardara la aplicación en el dispositivo abra el APK para iniciar la instalación de la aplicación. Para poder instalar esta aplicación, es necesario habilitar la instalación de aplicaciones de origen desconocido en el dispositivo.

Si la aplicación no puede instalarse en su dispositivo, significa que el dispositivo no cumple con los requisitos mínimos necesarios para usarla. Si desea instalar la aplicación a través de Android Studio, puede descargarse el código de la aplicación desde este enlace: https://github.com/CientQuely/TFG.

Uso de la aplicación

Al iniciarse la aplicación se mostrará por pantalla esta imagen:

Figura 33. Pantalla de inicio de la aplicación

Para poder iniciar la aplicación es necesario pulsar el botón **Start** y aceptar todos los permisos que le pida la aplicación. Una vez hecho esto, la aplicación va a redirigirlo a la actividad principal de la aplicación:

Figura 34. Actividad de realidad aumentada de la aplicación

Para **abrir el menú deslizante** de la aplicación deslice su dedo desde el límite izquierdo de la aplicación hacia la derecha. Al hacerlo, se abrirá este menú deslizante:

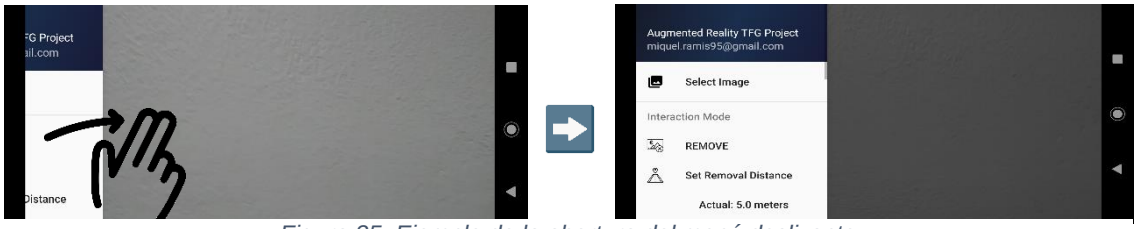

Figura 35. Ejemplo de la abertura del menú deslizante

En este menú existen múltiples funcionalidades con las que puede interactuar el usuario. La primera de ellas consiste en poder **seleccionar una imagen** desde la galería de imágenes:

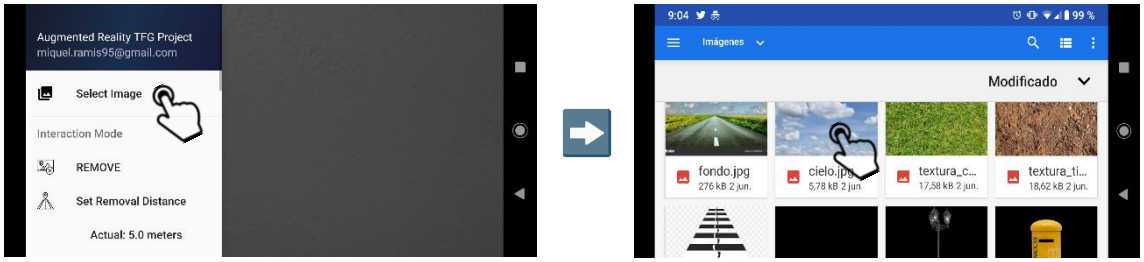

Figura 36. Ejemplo de selección de una imagen.

La aplicación posee **dos modos de interacción**: ADD para añadir imágenes y REMOVE para eliminarlas.

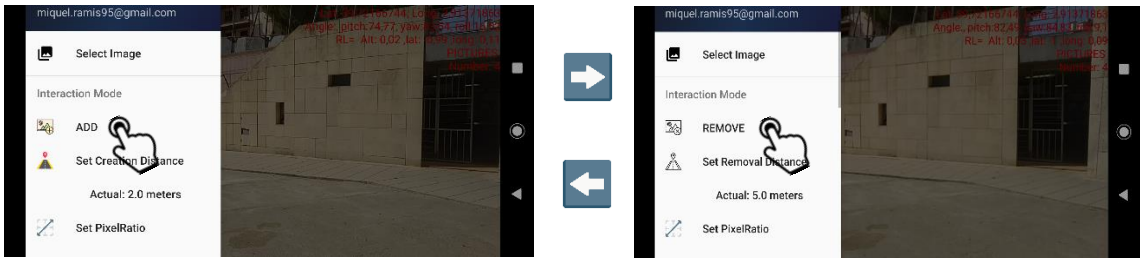

Figura 37. Cambio entre del modo ADD y REMOVE y vice versa

En el modo **ADD** podemos **crear la imagen seleccionada** desde la galería y también podemos **modificar la distancia** a la que se quiere crearla:

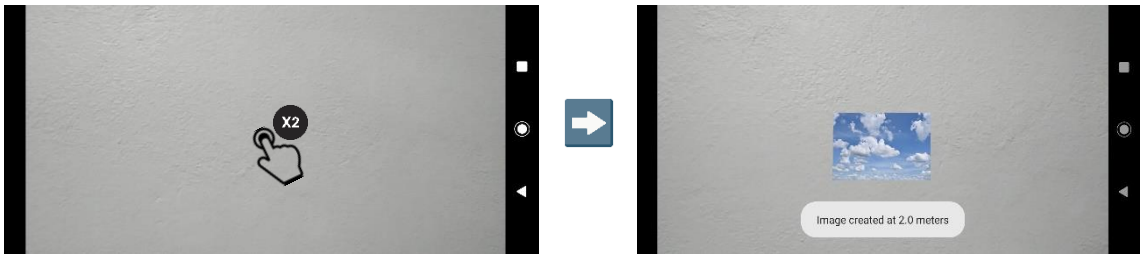

Figura 38. Ejemplo de la creación de una Instancia de Imagen de una imagen del cielo.

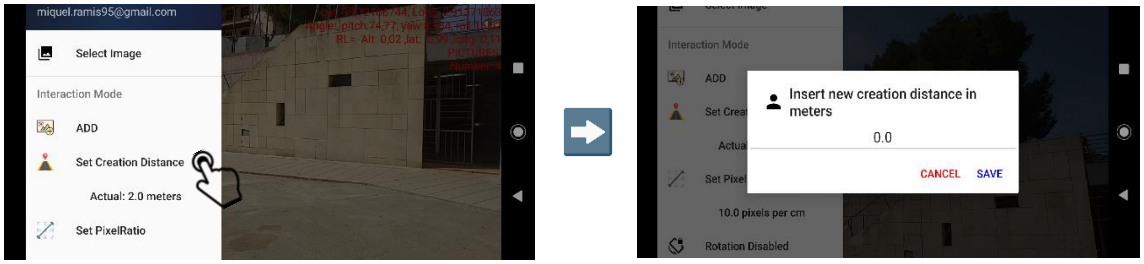

Figura 39. Ejemplo de cómo modificar la distancia de creación de las Instancias de Imagen

En el modo **REMOVE** podemos **eliminar las imágenes en un radio** de **M** metros respecto al usuario y podemos **modificar la distancia M** de eliminado:

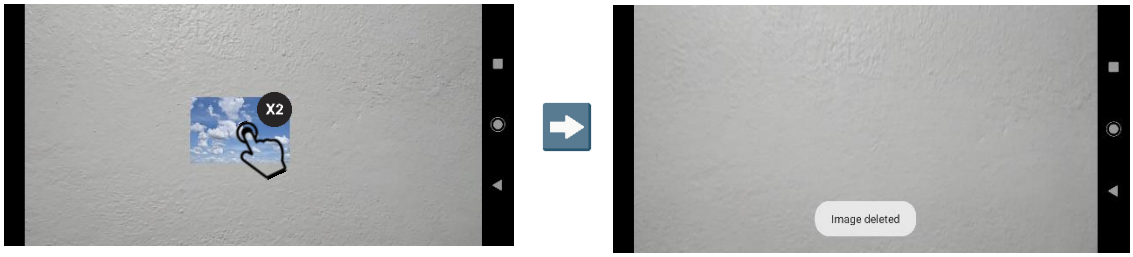

Figura 40. Ejemplo de eliminación de una Instancias de Imagen

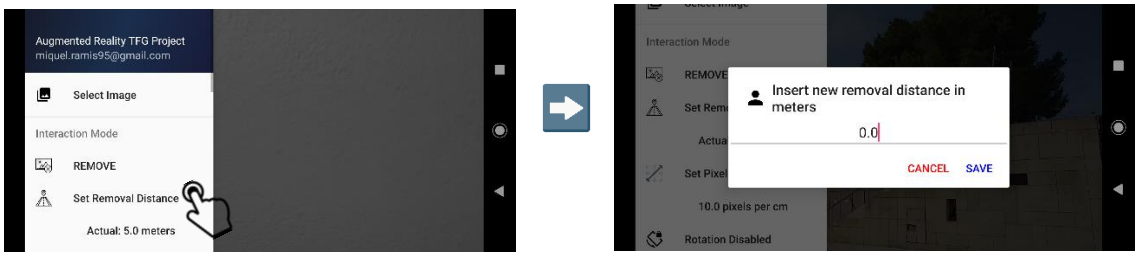

Figura 41. Ejemplo de cómo modificar la distancia de eliminación de las Instancias de Imagen

En ambos modos podemos **modificar el ratio de pixeles por centímetro** con el que van a crearse las imágenes:

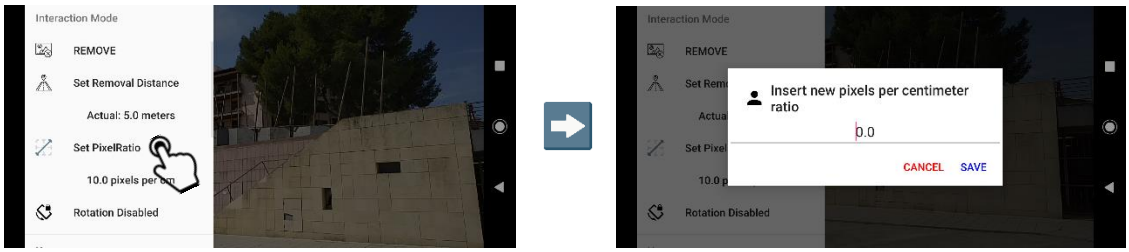

Figura 42. Ejemplo de cómo modificar el ratio de píxeles por centímetro.

En esta aplicación también se puede **habilitar o deshabilitar la rotación del eje de la cámara del dispositivo**:

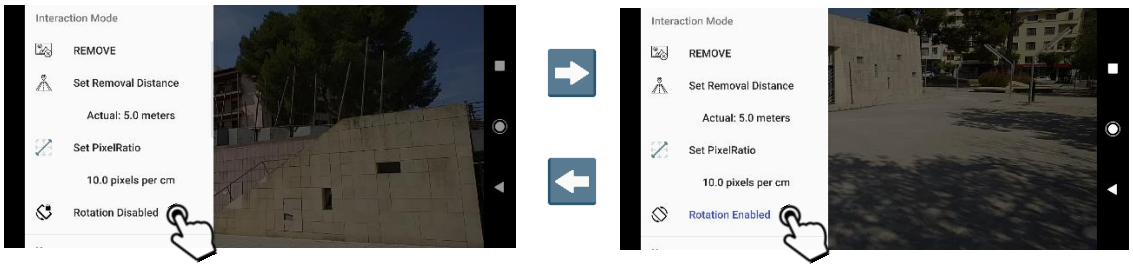

Figura 43. Ejemplo de cómo habilitar y deshabilitar la rotación sobre el eje de la cámara del dispositivo

Esta funcionalidad **al habilitarse, permite no rotar las imágenes** al rotar el dispositivo:

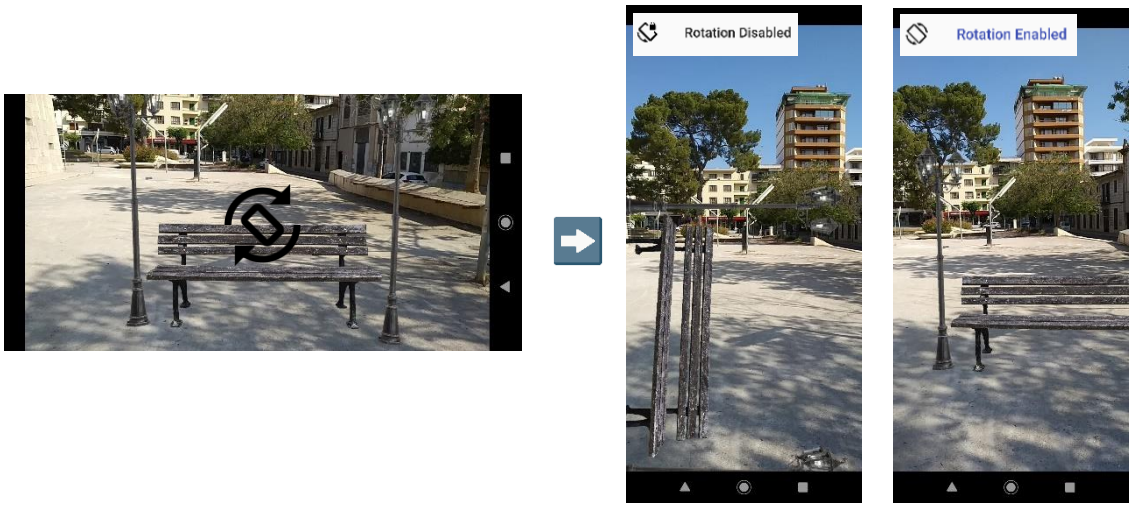

Figura 44. Ejemplo de la rotación sobre el eje de la cámara del dispositivo

Este sistema **por defecto esta deshabilitado** por problemas debido a que usa Ángulos de Euler para calcular la rotación en el eje de la cámara. Debido al Bloque de Cardán, existen ángulos o direcciones donde esta rotación no funciona correctamente.

Debido a que **la altitud** de la geolocalización del dispositivo es demasiado imprecisa como para usarse, el usuario debe **ingresar la altura manualmente**. Para cambiar la altura a la que está situado el usuario:

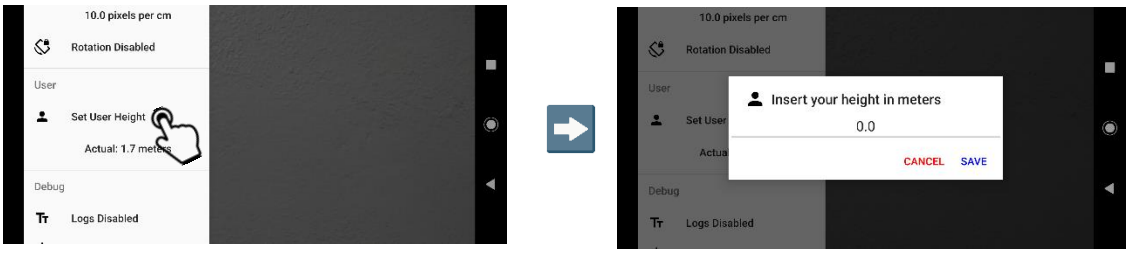

Figura 45. Ejemplo de cómo cambiar la altura actual del usuario

Esto permite que, al situarnos en una plataforma a cierta altura, puedan visualizarse las imágenes añadidas abajo de esta:

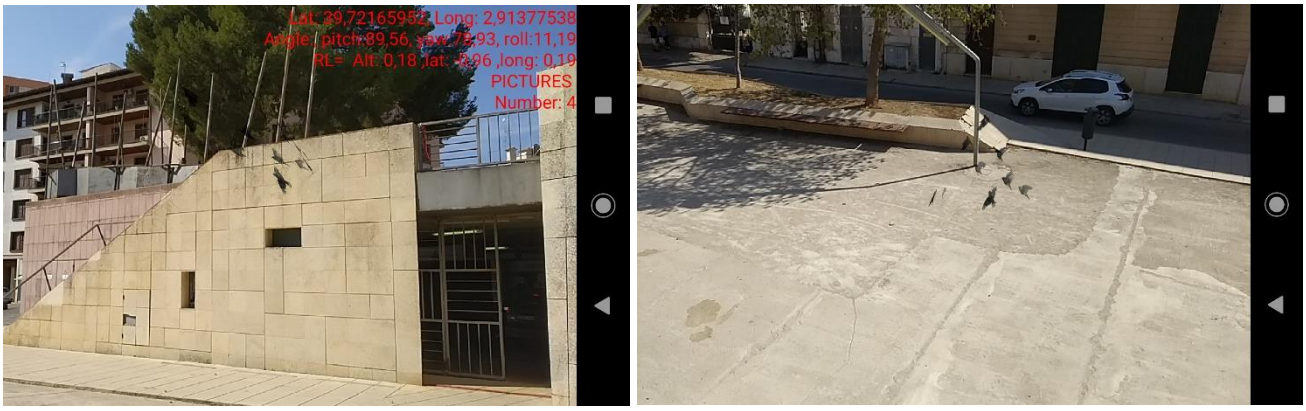

Figura 46. Ejemplo de la visualización de dos imágenes de pájaros a diferentes alturas

Para el depurado de la aplicación, existen otras tres funcionalidades.

La primera de ellas permite **habilitar y deshabilitar** los **Logs**. Los **Logs** son información de las características en tiempo real del dispositivo que pueden mostrarse por pantalla. Estos **muestran la latitud y longitud** actual del dispositivo, **los Ángulos de Euler** de la cámara del dispositivo, el punto que junto al origen de coordenadas forma el vector director a donde apunta la cámara del dispositivo y, finalmente, el número de Instancias de Imagen próximas al usuario:

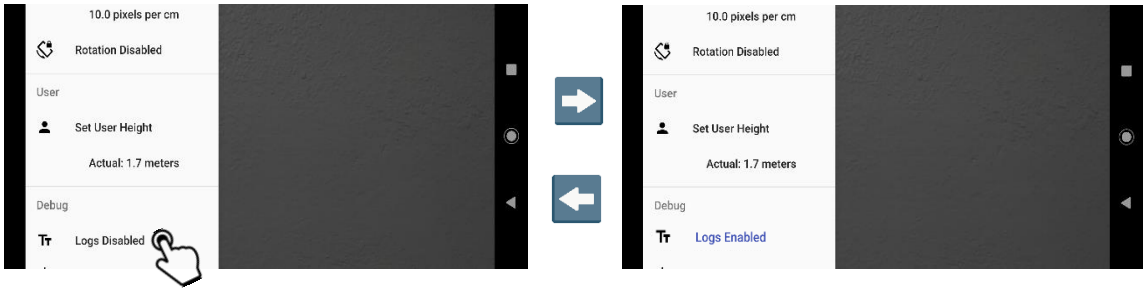

Figura 47. Ejemplo de cómo habilitar y deshabilitar los logs del dispositivo

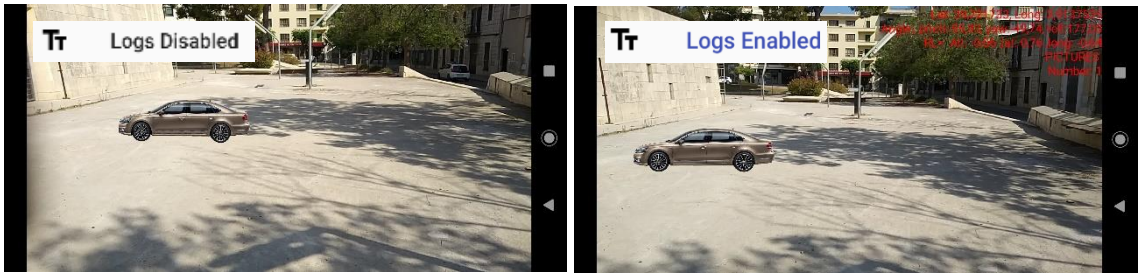

Figura 48. Comparación entre Logs deshabilitados y habilitados

También se puede **habilitar y deshabilitar la geolocalización del dispositivo**. Esto permite dejar de movernos en el espacio, para poder maquetar el entorno sin que se modifique la **geolocalización** del dispositivo:

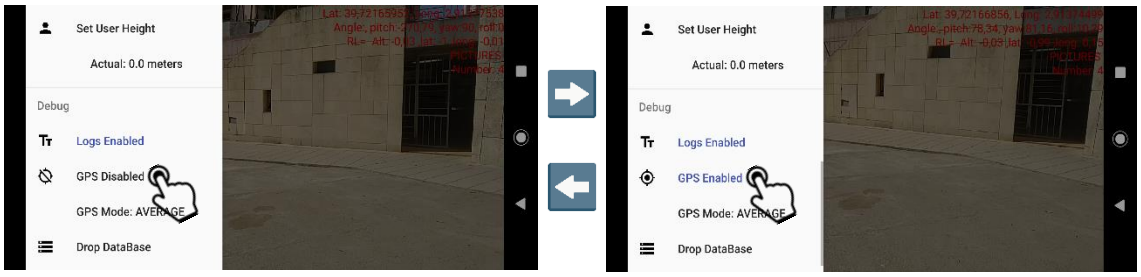

Figura 49. Ejemplo de cómo habilitar y deshabilitar la geolocalización del dispositivo

También podemos cambiar que tipo de cómputo usado para obtener la **geolocalización** del dispositivo. Los diferentes tipos de cómputo para la localización real del dispositivo son:

1. **LAST**: última localización obtenida. Es la más rápida y la más imprecisa.
2. **AVERAGE**: media aritmética de las últimas 20 localizaciones.
3. **PONDERED**(Recomendada): media aritmética ponderada de las últimas 20 localizaciones.

Para modificar el cómputo de la localización del dispositivo:

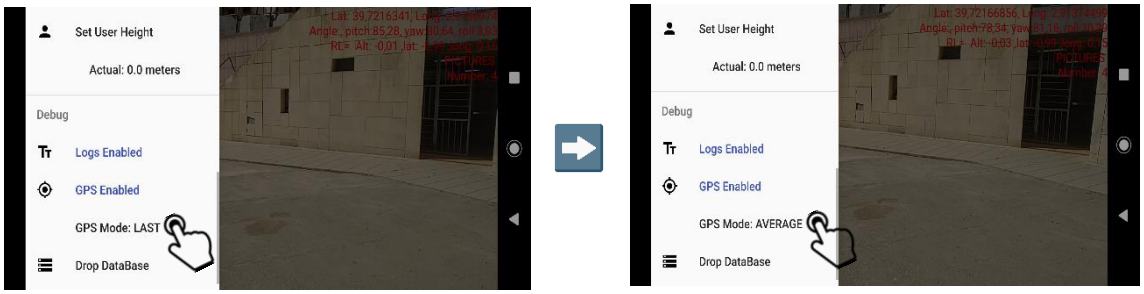

Figura 50. Ejemplo de cómo modificar el tipo de cómputo de la localización real del dispositivo

Finalmente, también se pueden eliminar todas las Instancias de Imagen guardadas en la base de datos:

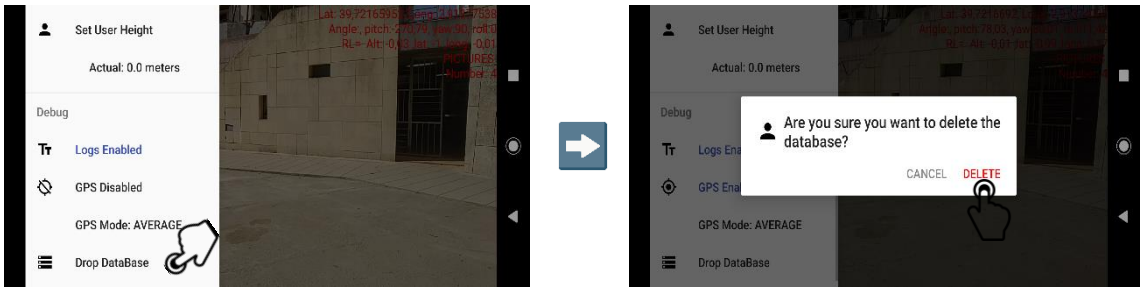

Figura 51. Ejemplo del borrado de todas las Instancia de Imagen guardadas en la BDD

Ejemplos

Figura 52. Imagen de dos marcos virtuales situados a la misma distancia

La única diferencia entre estos dos marcos es el ratio de píxeles por centímetro. El marco más grande posee la mitad del ratio de píxeles por centímetro que el mas pequeño.

Figura 53. Imagen de la maquetación en la Figura 34 en otra perspectiva.

Figura 54. Maquetación en un fondo pixelado para crear un escenario virtual.

Figura 55. Ejemplo de seis imágenes de pájaros con diferentes localizaciones y rotaciones.

Capítulo 10: Conclusiones

Gracias a este proyecto he sido capaz de comprender la magnitud y las dificultades de aplicar la realidad aumentada en dispositivos móviles. Crear una aplicación de RA me ha permitido adquirir conocimientos en muchos ámbitos de la programación. He aprendido a programar para dispositivos Android, he mejorada mis capacidades de integración y modulación de código significativamente, he aprendido a usar e implementar casi todos los sensores más importantes de un dispositivo móvil y finalmente, he aprendido a gestionar y a documentar proyectos de mayor envergadura.

Quiero remarcar la importancia del uso de la metodología ágil Scrum que me ha simplificado mucho, la gestión del volumen de trabajo en el proyecto.

El uso del lenguaje de programación Java en este proyecto ha sido una buena elección. He encontrado bastante información y documentación sobre cada tipo de sensor junto a ejemplos y maneras de inicializarlos. También, el uso de XML para maquetar la Vista en la pantalla del dispositivo, me ha parecido muy intuitiva.

A cualquier lector que quiera programar en dispositivos Android, le recomiendo el uso de Android Studio como entorno de desarrollo de su aplicación. Android Studio ofrece muchas ventajas, tanto para compilar y emular una aplicación, como para depurarla. El único problema que tuve con este entorno de desarrollo fue que, en algunas actualizaciones, el compilado de la aplicación dejaba de funcionar por un fallo en el compilador de Android y tenía que volver a reconfigurarlo.

La creación de una aplicación de RA, es un proyecto muy grande donde se pueden mejorar e implementar nuevas características indefinidamente. Por ejemplo, me hubiera gustado introducir o implementar modelados en tres dimensiones en el proyecto o implementar un servidor de imágenes para poder compartirlas en múltiples dispositivos.

Para realizar este proyecto he creado y usado más de 30 clases distintas. Si no hubiera usado el patrón MVP para controlar y gestionar correctamente las conexiones entre las distintas clases, me hubiera sido mucho más difícil crear esta aplicación. Implementarlo, me ha demostrado la importancia de aplicar un patrón arquitectónico adecuado para una aplicación de un tamaño mínimamente elevado.

Este proyecto me ha fascinado, tanto por todos los apartados que he programado como por la necesidad de aplicar múltiples soluciones a cada uno de los problemas y dificultades que se planteaban al usar la realidad aumentada en el proyecto.

Por ejemplo, el apartado que más trabajo me ha llevado ha sido encontrar una forma de aplicar la rotación real del dispositivo a la rotación virtual en OpenGL.

La optimización del Modelo de Datos para reducir la cantidad de recursos usados en la aplicación también la he encontrado muy entretenida de realizar. Siendo la parte de la que estoy mas orgulloso la Cuadrícula de Imágenes.

Estoy bastante satisfecho con el resultado final de la aplicación. No es un sistema perfecto, y aun puede considerarse un prototipo a mejorar en futuras versiones. Pero, mi objetivo era crear una aplicación capaz de usarse y que cumpliera los requisitos del proyecto de manera satisfactoria, y creo, que he cumplido con los requisitos que demandaba este proyecto en todos sus aspectos.

En mi opinión, estamos aún en las etapas tempranas de la implementación de la RA en dispositivos móviles. Actualmente, la imprecisión del GPS y el tiempo que tarda en obtener coordenadas precisas en entornos al aire libre, no permite crear un sistema de realidad aumentada fluido que lo use. Debido a ello, mientras no se mejoren los sensores de localización en dispositivos móviles, no vamos a ser capaces de implementar un sistema de realidad aumentada perfecto para maquetación de exteriores. También, en el caso del Sensor de Rotación, al sacudir repetidamente el dispositivo, puede causar un desplazamiento o error inevitable en el cálculo de la rotación del dispositivo.

Si tuviera que volver a elegir este proyecto, aunque conociera estos problemas en el hardware del dispositivo, lo haría encarecidamente. Y le recomiendo al lector, si está interesado en crear una aplicación parecida, que lo intente.

Finalmente, lo que más me enorgullece de haber realizado este proyecto, no es solo la experiencia obtenida en todos los ámbitos que ya he mencionado, sino, el demostrarme a mí mismo, que soy capaz de aplicar todos los conocimientos obtenidos a lo largo de mis estudios en retos y proyectos completamente nuevos, desconocidos y de mayor envergadura.

Bibliografía

[1] Alexander Pacha. GitHub "*Sensor fusion for robust outdoor Augmente Reality tracking on mobile devices*" [2013]. [En línea] Disponible: Enlace [Último acceso: 30-06-2019]

[2] Alexander Pacha. Tesis "*Sensor fusion for robust outdoor Augmented Reality tracking on mobile devices*" [2013] [En línea] Disponible: Enlace [Último acceso: 30-06-2019]

[3] Wiley Brand. "*Android App Development for Dummies 3rd Edition*" [2015]

[4] Jorge Martinez Ladrón de Guevara. "*Fundamentos de Programación en Java*" [2015] [En línea] Disponible: Enlace [Último acceso: 30-06-2019]

[5] Iubaris Info 4 Media SL. "*Guia de Formación de Scrum Manager*" [2016] [En línea] Disponible: Enlace [Último acceso: 30-06-2019]

[6] Irving Alberto Cruz Matías. Capítulo 3 de la "*Tesis de Rotaciones Multidimensionales Generales*" [2015] [En línea] Disponible: Enlace [Último acceso: 30-06-2019]

[7] Wikipedia. "*Quaternions and Spatial Rotation*" [2017] [En línea] Disponible: Enlace [Último acceso: 30-06-2019]

[8] Kevin Brothaler. "*OpenGL ES 2 for Android: A Quick-Start Guide*" [2014]

[9] Chandan Singh. "*Observable and Obersever Java implementation example*" [2014] [En línea] Disponible: Enlace [Último acceso: 30-06-2019]

[10] Rodrigo Santamaría. "*Hilos en Java*" [2015] [En línea] Disponible: Enlace [Último acceso: 30-06-2019]

[11] Android Developer Website. "*User Location Guide*" [En línea] Disponible: Enlace [Último acceso: 30-06-2019]

[12] Android Developer Website. "*Camera2 package provider documentation*" [En línea] Disponible: Enlace [Último acceso: 30-06-2019]

[13] Kevin Bonsor, Nathan Chandler. "How Augmented Reality Works" [En línea] Disponible: Enlace [Último acceso: 30-06-2019]