Universidad Zaragoza
1542

# Trabajo Fin de Grado

## Aplicación de Realidad Aumentada para el Museo de Informática Histórica (MIH) de la Universidad de Zaragoza


Autor/es

Roberto Clemente Salvador

Director/es

Eduardo Mena Nieto

Departamento de Informática e Ingeniería de Sistemas
Escuela de Ingeniería y Arquitectura

Septiembre de 2018

TRABAJOS DE FIN DE GRADO / FIN DE MÁSTER

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

# DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD

(Este documento debe acompañar al Trabajo Fin de Grado (TFG)/Trabajo Fin de Máster (TFM) cuando sea depositado para su evaluación).

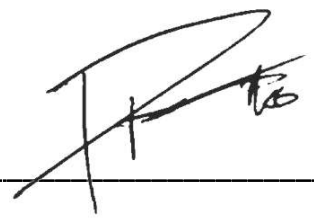
D./Dª. Roberto Clemente Salvador,

con nº de DNI 18174418W en aplicación de lo dispuesto en el art. 14 (Derechos de autor) del Acuerdo de 11 de septiembre de 2014, del Consejo de Gobierno, por el que se aprueba el Reglamento de los TFG y TFM de la Universidad de Zaragoza,

Declaro que el presente Trabajo de Fin de (Grado/Máster) Grado de Ingeniería Informática, (Título del Trabajo) Aplicación de Realidad Aumentada para el Museo de Informática Histórica (MIH) de la Universidad de Zaragoza

es de mi autoría y es original, no habiéndose utilizado fuente sin ser citada debidamente.

Zaragoza, 18 de Septiembre de 2018

Fdo: 

# Aplicación de Realidad Aumentada para el Museo de Informática Histórica (MIH) de la Universidad de Zaragoza


**RESUMEN**

El MIH (Museo de Informática Histórica) cuenta con un gran número de ítems en su colección, al mismo tiempo que un número limitado de vitrinas para exponerlos. Por ello, estos objetos se suelen exponer juntos en un entorno reducido, lo cual no deja espacio para colocar sus respectivas etiquetas. El propósito de este proyecto es evitar esta limitación, llevando la información de dichas etiquetas a una plataforma móvil. A través de la cámara del dispositivo el usuario podrá reconocer los objetos que se encuentren ante él y, acto seguido, el sistema dibujará una etiqueta virtual sobre el objeto reconocido mediante técnicas de Realidad Aumentada. Además, el usuario podrá seleccionar los objetos que reconozca para visualizar información más detallada de los mismos, incluyendo una descripción, una imagen y un modelo 3D con el que poder interactuar. Además, el sistema permitirá visualizar estanterías y objetos virtuales como si realmente figurasen en el museo.

Para que los datos de la aplicación sean consistentes con los objetos que figuran actualmente en el museo se implantará una base de datos en un servidor externo, desde la que será posible actualizar los datos de la aplicación en caso de modificación. Para gestionar dicha BD se creará una herramienta con interfaz gráfica, facilitando así su mantenimiento por parte del administrador.

*Dedicado a mis abuelos Nicolás y Avelina,*
*sé que estaríais orgullosos.*

# Agradecimientos

En primer lugar, quiero agradecer a mi tutor Eduardo Mena, por embarcarme en un proyecto tan masoquista como apasionante. Cuanto más grande era el problema, el bloqueo, el tirón de pelos o la dificultad, más grande era la satisfacción al superarlos, haciéndome enfrentar con motivación el siguiente reto. También agradecer a Juan Domingo Tardos por aconsejarme en el reconocimiento de objetos.

Gracias a mi hermano Miguel y mi primo Germán, por todas sus ayudas, por aguantarme y por las indispensables partidas de *Smash Bros.* en los descansos del trabajo.

Mil gracias a mis padres Miguel y Carmen, por sus correcciones y recomendaciones. Pero eso no es nada comparado con simplemente estar ahí.

También quiero agradecer a mis abuelos José y Encarna por sus velas y rezos. Creo que el reconocimiento de objetos ha sido posible gracias a ellos.

Gracias a mis tíos Lourdes y Ángel de Barcelona, segundos padres, por educarme con Rock desde pequeño y por los grandes momentos.

Agradecer a mi tía Pili, por sus correcciones en los textos en inglés y por los "lamines" de por medio.

Muchas gracias a mi tio Ramón por introducirme en el mundo de la tecnología desde pequeño, cuando empecé a llenarme la boca diciendo que quería ser "Ingeniero de Telecomunicaciones" sin siquiera saber que significaba.

También gracias de antemano a todos mis amigos por ayudarme a olvidar pronto todo lo que en este trabajo he aprendido.

Por último, quiero agradecer especialmente a la persona que más me ha tenido que aguantar, que le ha tocado ejercer el papel de "Pepito Grillo" en incontables ocasiones y que ha convertido los últimos 4 años en los mejores de mi vida. Gracias, Isabel.

# Índice

# Índice de figuras

# Índice de tablas

# Capítulo 1

# Introducción

Este trabajo comprende el desarrollo de una aplicación móvil de Realidad Aumentada para el Museo de Informática Histórica de la Universidad de Zaragoza, aportando también todas las herramientas necesarias para su gestión. Adicionalmente se ha considerado tarea del estudiante analizar las metodologías existentes de la Realidad Aumentada y el reconocimiento de objetos.

## 1.1. El Museo de Informática Histórica

El Museo de Informática Histórica (MIH)[1] surgió en 2003 de la iniciativa de un grupo de personas del Centro Politécnico Superior de la Universidad de Zaragoza. Su objetivo es recuperar parte de los precursores de la breve y trepidante era informática, completando así unos fondos representativos del avance tecnológico en nuestra sociedad.

En estos momentos los fondos del museo incluyen varios centenares de artículos tecnológicos (desde sistemas completos a dispositivos de almacenamiento, microprocesadores, periféricos...) y sus correspondientes fuentes de documentación (manuales originales, guías de usuario, fotografías, documentos digitales, etc).

La parte más importante de estos fondos se encuentra expuesta en las vitrinas que se distribuyen a lo largo del hall del edificio Ada Byron del Campus Río Ebro. Como consecuencia de la remodelación realizada en Octubre de 2008, se decidió dividir la exposición permanente en cinco áreas temáticas, cubriendo cada una de ellas una familia de ordenadores: ordenadores domésticos, ordenadores Apple, compatibles PC, estaciones de trabajo y servidores y terminales. Además de los ordenadores expuestos en los expositores centrales, cada vitrina de pared ofrece una extensa muestra de los distintos componentes hardware, parte del software y documentación más significativa de cada familia de ordenadores, incluyendo la evolución de los microprocesadores y de los sistemas de almacenamiento. En la figura 1.1 puede apreciarse un ejemplo de las vitrinas del museo.

A pesar de que el MIH (Museo de Informática Histórica) cuenta con un gran número de dispositivos en su colección el espacio que se le ha dedicado para exponerlos es limitado. Esto hace que muchos de ellos (sobre todo los de menor tamaño) se tengan que disponer en una superficie muy reducida, lo cual no deja espacio para colocar sus respectivas etiquetas. Además son precisamente los objetos pequeños los más costosos de identificar por visitantes no expertos.

[1] *Texto extraído del panel 1 del Museo de Informática Histórica (MIH), accesible a través de la siguiente dirección web (último acceso 17/09/2018): http://mih.unizar.es/*


Figura 1.1: Ejemplo de vitrina del MIH.

## 1.2. Realidad Aumentada

La Realidad Aumentada[3] es la fusión del mundo real y el virtual, de forma que los elementos virtuales se superponen sobre el escenario real creando la ilusión de que verderamente están ahí (ver ejemplo en figura 1.2). Para ello, son necesarios tres elementos: una cámara, una pantalla y un *software* que defina cómo han de mostrarse los elementos virtuales. Los dispositivos móviles, presentes ya en la mayor parte de la población, poseen dichas características que, junto al hecho de ser portátiles, suponen uno de los mejores medios en la actualidad para hacer uso de la Realidad Aumentada. Es por esto que esta tecnología se encuentra en auge, llevando a muchas empresas a explorar sus posibles aplicaciones.

En julio de 2016 apareció el juego Pokémon GO, que gracias a su popularidad masiva dio a conocer el concepto de la Realidad Aumentada a nivel global. Sin embargo, a parte de servir como atractivo en productos de entretenimiento, la RA tiene multitud de aplicaciones, como pueden ser pedagógicos, médicos o turísticos.

## 1.3. Objetivo

El propósito de este proyecto es llevar la información de dichas etiquetas a una plataforma móvil para evitar la sobrecarga de las vitrinas del museo y facilitar la identificación de los objetos menos distinguibles. El usuario podrá ver unas etiquetas virtuales junto a los objetos detectados por la cámara de la aplicación gracias a tecnologías de Realidad Aumentada. También podrá acceder a información más detallada pulsando sobre el objeto en la pantalla. Aprovechando la información introducida en la aplicación el sistema incluirá un catálogo desde el que se podrá consultar los objetos.

La aplicación funcionará también para aquellos objetos de la colección que cuentan con etiquetas reales, puesto que no pretende sustituir el sistema tradicional de etiquetas, sino

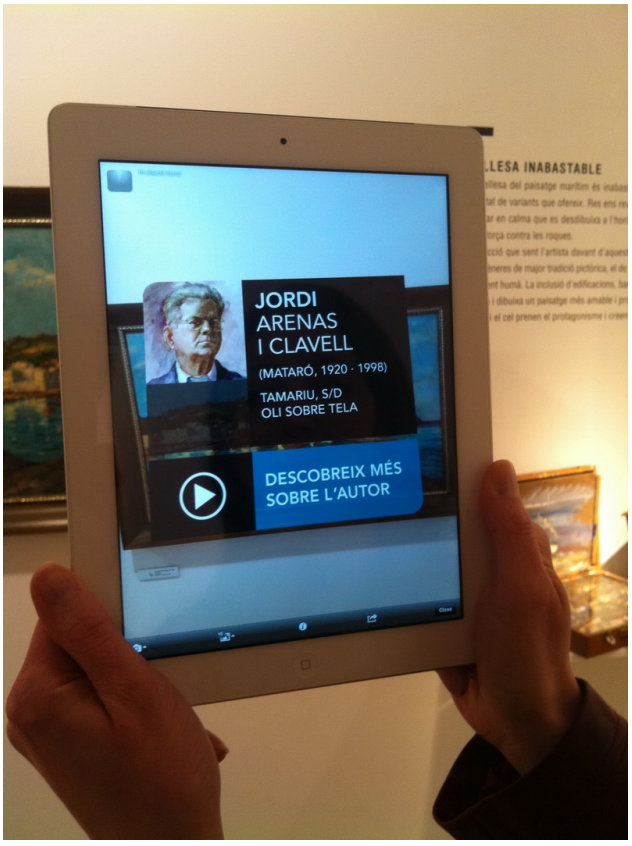



Figura 1.2: Ejemplo de aplicación de RA.

complementarlo. De este modo, el sistema mantiene también un registro de todos los objetos actuales del museo.

Además, se deberá tener en cuenta que el museo puede experimentar modificaciones. Está prevista una nueva remodelación para otoño de 2018, que implicará muchos cambios en la organización actual de los elementos del museo. Por tanto, para que el sistema sea mantenible, deberá ser capaz de actualizar las modificaciones sin importunios para el usuario, suponiendo un proceso lo más transparente posible. Del mismo modo, el administrador de los datos deberá poder aplicar las modificaciones de forma sencilla.

Por último, se aprovechará el motor de Realidad Aumentada implementado para mostrar estanterías virtuales como si realmente existiesen en el museo, que incluirán a su vez objetos virtuales. A modo de ejemplo, podría colocarse, al lado de una vitrina de pared existente, una estantería virtual que expusiera distintos modelos de consolas antiguas, o bien una estantería virtual que mostrase objetos existentes pero en otra disposición para previsualizar una posible reestructuración del museo.

# Capítulo 2

# Requisitos

Siendo el objetivo último elaborar la aplicación, se han definido los requisitos a satisfacer únicamente para esta aplicación. El resto de elementos del sistema global (herramientas, bases de datos, etc.) son considerados medios para satisfacer estos requisitos y no estarán sujetos a requisitos propios.

## 2.1. Primera iteración

Los primeros requisitos de la aplicación a desarrollar se especifican de manera resumida en las tablas 2.1 y 2.2.

| ID | Descripción |
|---|---|
| RF1 | El usuario podrá seleccionar los objetos encontrados en la imagen para ver su información detallada pulsando sobre ellos en la pantalla táctil. |
| RF2 | El usuario podrá desplegar el menú de opciones deslizando el dedo por la pantalla táctil hacia la derecha. |
| RF3 | El sistema permitirá al usuario descargar la última versión de la base de datos de la colección (o posponer) al iniciar la aplicación o a través del menú. |
| RF4 | El sistema será capaz de reconocer objetos en una imagen. |
| RF5 | El sistema será capaz de localizar los objetos reconocidos dentro de una imagen. |
| RF6 | El sistema será capaz de posicionar la cámara respecto a una imagen tomada. |
| RF7 | El sistema será capaz de mostrar información de un objeto en la misma persperctiva que la imagen (Realidad Aumentada). |
| RF8 | El sistema podrá mostrar una estantería virtual. |
| RF9 | El sistema podrá mostrar reconstrucciones 3D de los objetos. |

Tabla 2.1: Tabla de requisitos funcionales

| ID | Descripción |
|---|---|
| RF10 | El sistema será capaz de visualizar la información de los objetos desde un catálogo. |
| RF11 | El sistema será capaz de reconocer baldas de estanterías reales, identificando los objetos que en ellas se encuentren. |
| RF12 | El usuario podrá seleccionar objetos virtuales situados en estanterías virtuales para visualizar su información detallada. |
| RF13 | El sistema mostrará un catálogo con imágenes de los objetos. |
| RF14 | El sistema mostrará la localización de los objetos en el museo. |

Tabla 2.3: Tabla de nuevos requisitos funcionales

| Id | Descripción |
|---|---|
| RNF6 | El algoritmo de reconocimiento utilizará el algoritmo de Bolsa de Palabras. |
| RNF7 | El sistema generará la colección por primera vez desde un fichero. |

Tabla 2.4: Tabla de nuevos requisitos no funcionales

| Id | Descripción |
|---|---|
| RNF1 | El sistema se implementará con herramientas gratuitas. |
| RNF2 | El sistema funcionará a tiempo real. |
| RNF3 | El sistema funcionará sobre dispositivos Android. |
| RNF4 | El sistema estará terminado para la primera semana de septiembre de 2017. |
| RNF5 | El sistema requiere acceso a internet. |

Tabla 2.2: Tabla de requisitos no funcionales

## 2.2. Segunda iteración

A la vista de los resultados obtenidos en la primera versión de la aplicación se decidieron añadir los siguientes nuevos requisitos, representados en las tablas 2.3 y 2.4.

# Capítulo 3

# Diseño y arquitectura del sistema

Durante esta sección se describirán las decisiones tomadas relativas a la estructura del sistema y diseño, visto desde el punto de vista de la ingeniería del software.

## 3.1. Descripción del sistema

A la hora de diseñar el sistema se prestó especial importancia a los datos del museo que la aplicación iba a utilizar, así como a la concordancia de los mismos con el estado del museo actual. Como se ha comentado anteriormente, los objetos del museo son susceptibles de cambios, como pueden ser: retiro de la exposición, cambio de estantería, inclusión en estanterías virtuales... Todo ello requiere que, para que el estado de la aplicación concuerde con el del museo, esta sea capaz de adquirir los cambios de alguna forma.
Por ello, se decidió elaborar un sistema basado en tres partes o componentes:

- **MIHex** (MIH explorer). Aplicación de Realidad Aumentada, producto final de este TFG.
- **Servidor**. Almacenará los datos del museo necesarios para la aplicación en una base de datos[7], que será accesible desde la aplicación MIHex. Su objetivo es mantener la aplicación actualizada con la información del estado del museo actual.
- **MIHDatabaseManager** (**MIH Da**ta**Ba**se**Ma**nager). Herramienta gestora que facilitará la introducción de datos en la base de datos del servidor.

En la figura 3.1 se refleja un esquema del sistema a implantar:

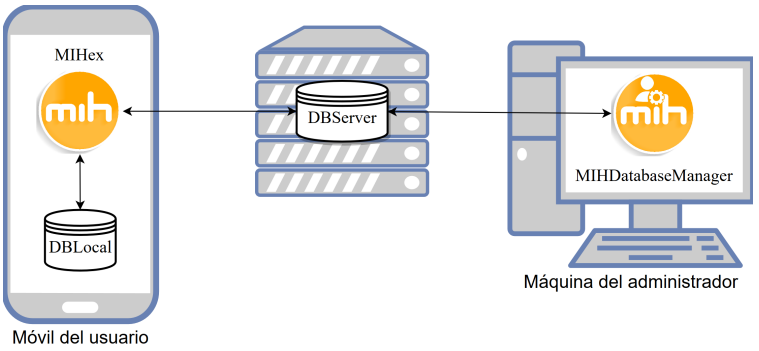



Figura 3.1: Arquitectura del sistema

### 3.1.1. MIHex

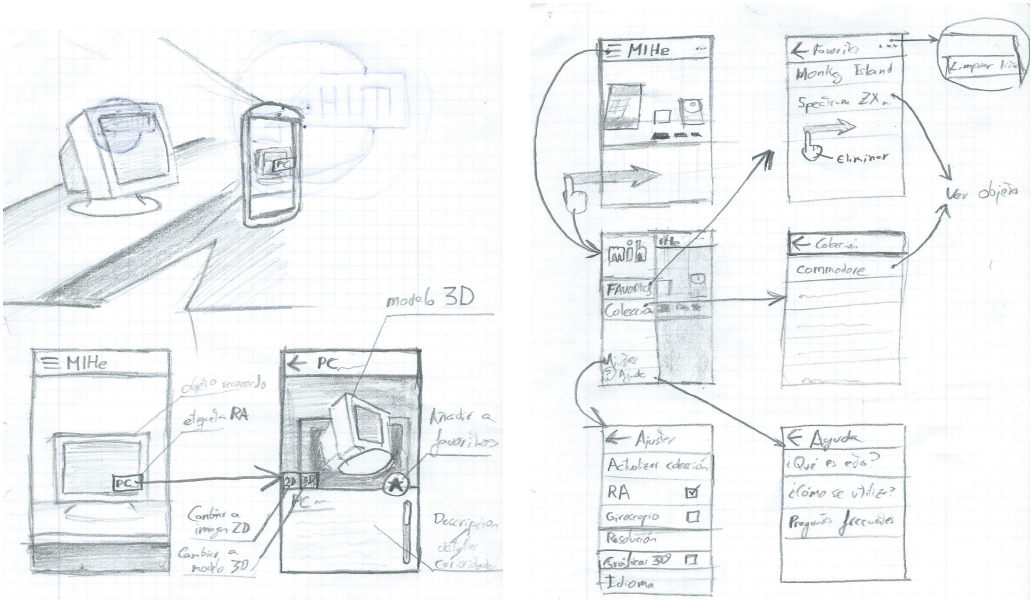
La elaboración de la aplicación MIHex es el objetivo último de este proyecto. Además de las características mencionadas anteriormente, el diseño de la aplicación se enfocará bajo la premisa de que el producto final será utilizado por los visitantes del museo. Por ello, se deberá prestar especial atención a una serie de características:

- Fluidez. El usuario no quiere demoras, por lo que el sistema deberá funcionar lo más rápido posible, lo que implicará prestar especial atención a la implementación, optimizando todo aquello que pueda ocasionar esperas.

- Experiencia de uso agradable. En el caso de acciones inevitablemente costosas (actualizaciones, reconocimiento, etc.) el sistema mostrará animaciones de carga o procesado para entretener al usuario y amenizar el tiempo de espera.

- Interfaz cómoda y familiar. Se desea que la interfaz resulte agradable al usuario, por lo que se optó por utilizar componentes propios de las aplicaciones de Android (menús desplegables, barra de navegación, uso de botones flotantes...), sujetos además a las guías de estilo del *Material Design*[1].

En la figura 3.2 se muestra el prototipo en papel en el que se basa la aplicación MIHex. Incluye además el mapa de navegación. El resultado final, junto con una guía para el usuario, quedan reflejados en el anexo D.




(a) Pantalla principal y vista de objeto (b) Navegación por menús

Figura 3.2: Prototipo y mapa de navegación

[1] Normativa de diseño definida por Google.

### 3.1.2. MIHDatabaseManager

El objetivo de esta herramienta es facilitarle al administrador la tarea de introducción de datos. Una vez finalizado este proyecto, realizar un mantenimiento del sistema (introducir nuevo objetos, actualizarlos, enfrentar la futura remodelación del museo) ya no será competencia del alumno, por lo que la herramienta debe ser fácil de utilizar por un nuevo administrador sin demasiada curva de aprendizaje. Por ello, se incluye al final de este trabajo el anexo E, que compone un manual de usuario de esta aplicación.

La aplicación contará con una interfaz gráfica, pues algunos elementos son de caracter visual (como estanterías y objetos virtuales) y, de no ser por ella, sería muy complicado configurarlos. Además, otorgándole una interfaz de usuario se consigue un nuevo nivel de abstracción, pues el administrador no requerirá amplios conocimientos de cómo funciona el sistema por debajo. Esta capa intermedia garantiza también una correcta introducción de datos, minimizando así la posibilidad de errores por parte del administrador.

## 3.2. Arquitectura

Tanto la *app* MIHex como la herramienta MIHDatabaseManager contendrán al menos los siguientes 4 módulos principales:

- ***vision***. Incluirá todos los componentes relativos a la visión por computador, utilizada durante el reconocimiento y seguimiento de objetos.
- ***graphics***. Incluirá todos los componentes relativos a los gráficos 3D, utilizados durante la realidad aumentada y vista en detalle de un objeto.
- ***database***. Incluirá todos los componentes relativos a la base de datos.
- ***util***. Incluirá herramientas comunes en ambas aplicaciones, como lectura/escritura de ficheros.

Con el fin de reutilizar código y mejorar la consistencia entre las aplicaciones, se elaborará un paquete compartido *commons*, en el que se introducirán, para cada uno de los módulos, todas las clases de uso común. Ambas aplicaciones contendrán sus propios paquetes para los módulos, cuyas clases podrán extender a las existentes en *commons* para añadir funcionalidades propias. También contendrán otros módulos totalmente dependientes de la plataforma, como el de interfaz gráfica (UI) y el código en C++ de MIHex. La representación de esta estructura queda reflejada en el diagrama de paquetes de la figura 3.3.

En el diagrama de despliegue de la figura 3.4 supone una representación del sistema completo, incluyendo también la BD del servidor. También se muestran los paquetes mencionados anteriormente que serán incluidos en cada aplicación.

## 3.3. Gestión de datos

Se consideró que las bases de datos utilizadas iban a resultar uno de los pilares más importantes para el sistema. Es por ello que se puso especial énfasis en su elaboración, de forma que no hubiera que aplicar demasiados rediseños, ya que afectan directamente al

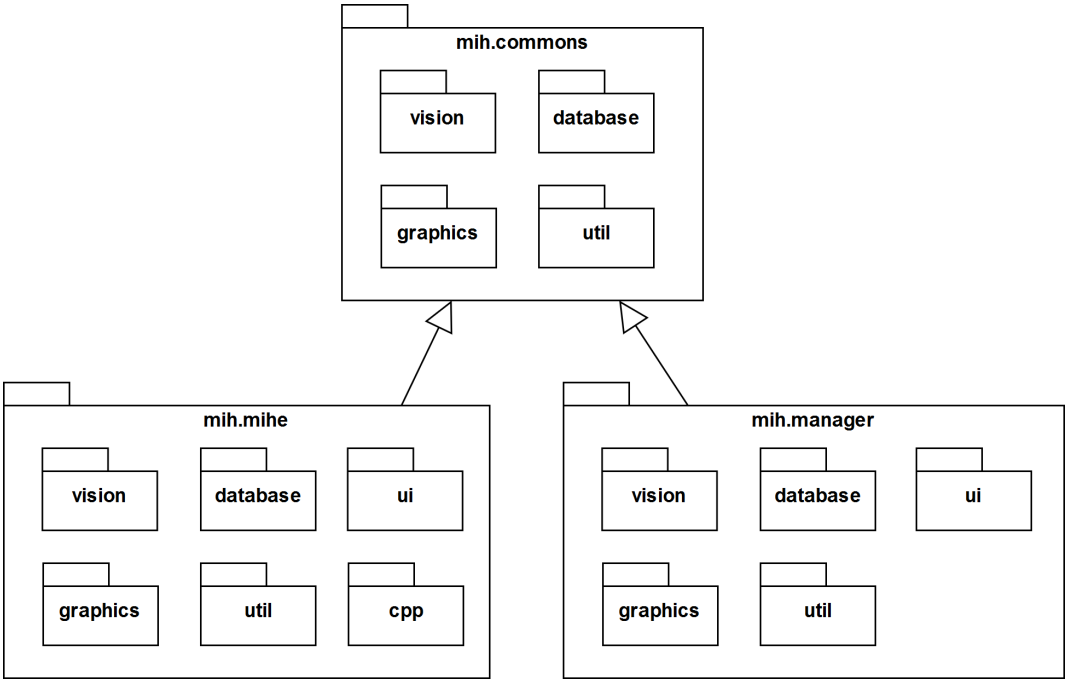



Figura 3.3: Diagrama de paquetes de la aplicación

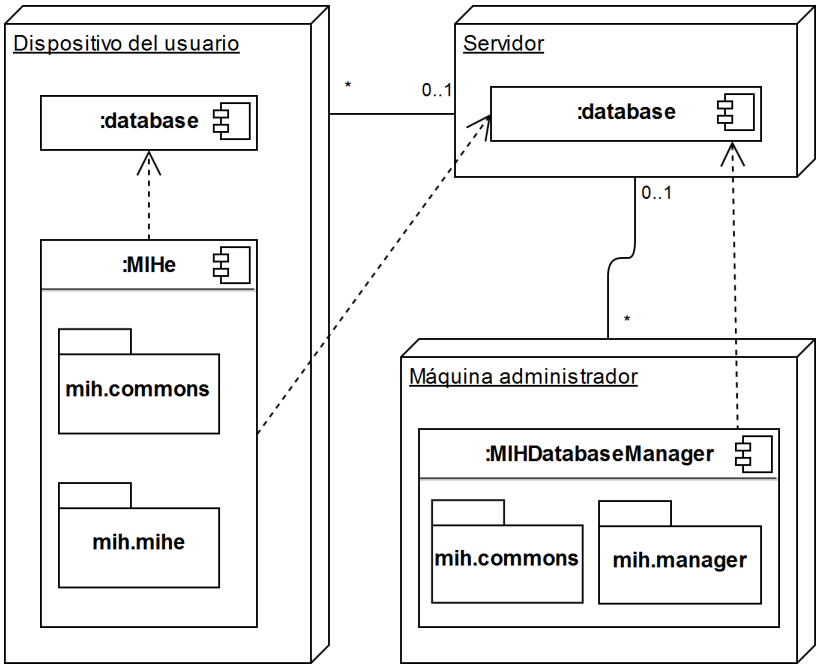



Figura 3.4: Diagrama de despliegue de la aplicación

resto del sistema. La base de datos está pensada con el fin de dar capacidad para todas las funcionalidades previstas, aunque en la versión final de la aplicación no se llevaran a cabo por motivos de tiempo. En la figura 3.5 se muestra el modelo básico Entidad-Relación diseñado para dar soporte a la estructura básica de elementos que se pretenden almacenar. El modelo completo se encuentra detallado en el anexo B, donde se muestra también todo el diseño y

pasos realizados sobre esta base de datos. La base de datos interna de la *app* cuenta con una estructura muy similar para mantener la mayor consistencia posible, salvo por algunos detalles especificados también en el anexo anteriormente mencionado que la simplifican.

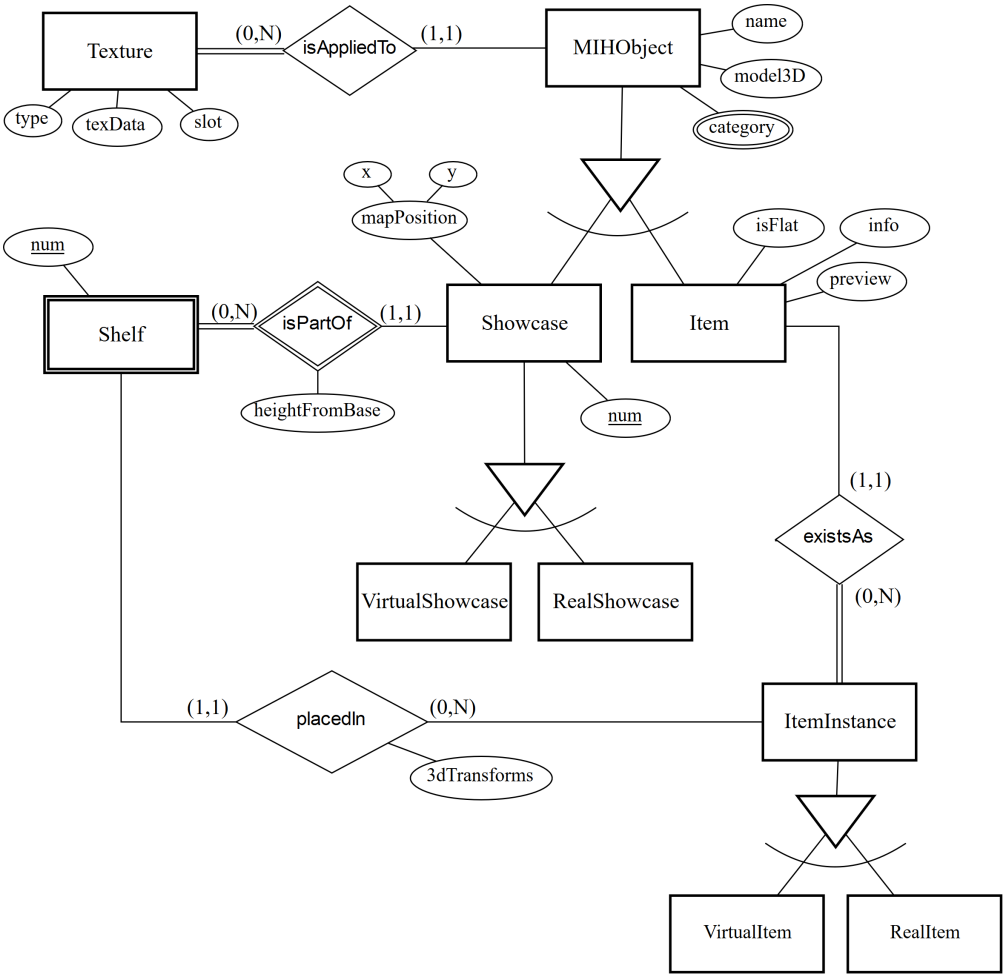



Figura 3.5: Diagrama ER de los elementos del museo a almacenar. No incluye elementos específicos del reconocimiento ni de actualización.

Además, se decidió que las imágenes utilizadas para la detección de objetos no se almacenarían en la base de datos, ya que desde la *app* únicamente es necesario el cómputo que se realiza sobre ellas. Será este resultado el que se guarde en la base de datos, ya que ocupa menos espacio de almacenamiento. Esto también implicará un tiempo de actualización más rápido y un cálculo menos que realizar a la hora de detectar los objetos. Por contraparte, al no estar almacenadas en la base de datos, las imágenes deberán encontrarse en la máquina de la aplicación gestora, concretamente en directorios generados automáticamente para cada clase de reconocimiento.

### 3.3.1. Diagramas de clases

Para representar en la aplicación los componentes del museo que se desean digitalizar, se elaboró el diagrama de clases de la figura 3.6, donde se muestran las interfaces y relaciones de dichos componentes y de los utilizados durante el reconocimiento: *Recognizable*,

representando los objetos que pueden ser reconocidos (items, estanterías y baldas) y *Detectable*, representando a los que pueden aparecer en una imagen (reconocibles + instancias). Estas interfaces se incluirán en el paquete *database* de *commons*, mientras la implementación vendrá dada en cada una de las aplicaciones, ya que trabajarán de forma distinta por debajo.

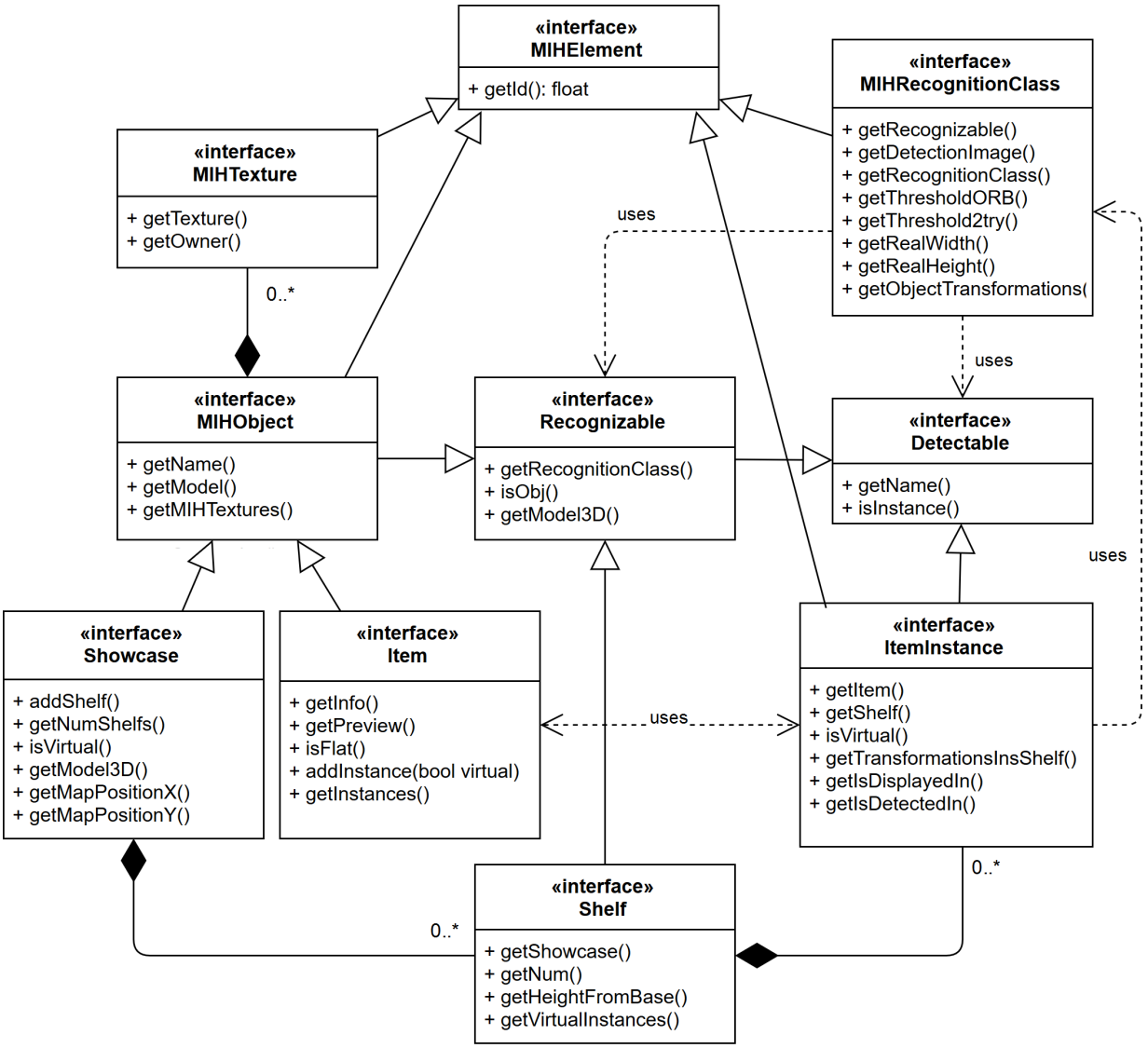



Figura 3.6: Diagrama de clases de los componentes del museo

Tanto en MIHDatabaseManager como en MIHex se decidió utilizar DAOs[2] para abstraer a las clases de los componentes de su almacenamiento en base de datos y archivos. Como desde MIHDatabaseManager se pueden lanzar consultas CRUD sobre la base de datos, los DAOs deberán implementar cada una de las operaciones, tal y como se muestra en la figura 3.7.

La *app* MIHex tiene acceso a las 2 bases de datos, la del servidor y la local. Sin embargo, se decidió crear únicamente un DAO para cada elemento, ya que van a contar con la misma

[2] Data Access Object. Interfaz que facilita el almacenamiento de un objeto de negocio.

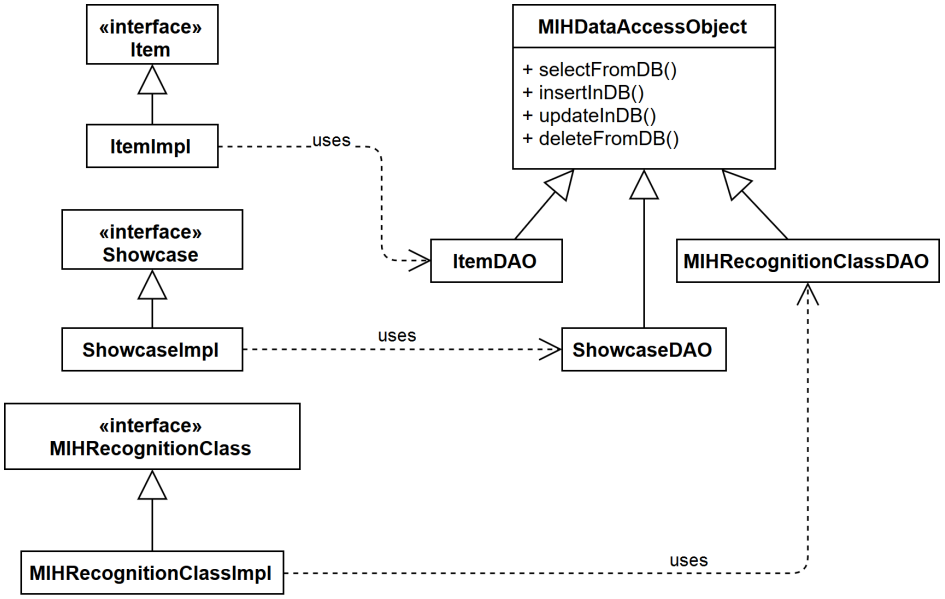



Figura 3.7: Ejemplos de clases de los componentes del museo y sus respectivos DAOs en MIHDatabaseManager.

estructura. Además, en este caso, las operaciones de inserción, actualización y borrado sobre la base de datos local serán muy similares entre sí, ya que siempre se realizarán desde fichero o desde la base de datos del servidor. Para facilitar este proceso, se ha diseñado un elemento extra que se encargará de centralizar todos estos procesos, llamado *TableUpdater*. De esta forma, cada DAO trabajará a través de un *TableUpdater* especificándole la tabla a la que corresponde junto con sus atributos, encapsulados en la clase *TableField*. Estas modificaciones se ven reflejadas en el diagrama de clases de la figura 3.8.

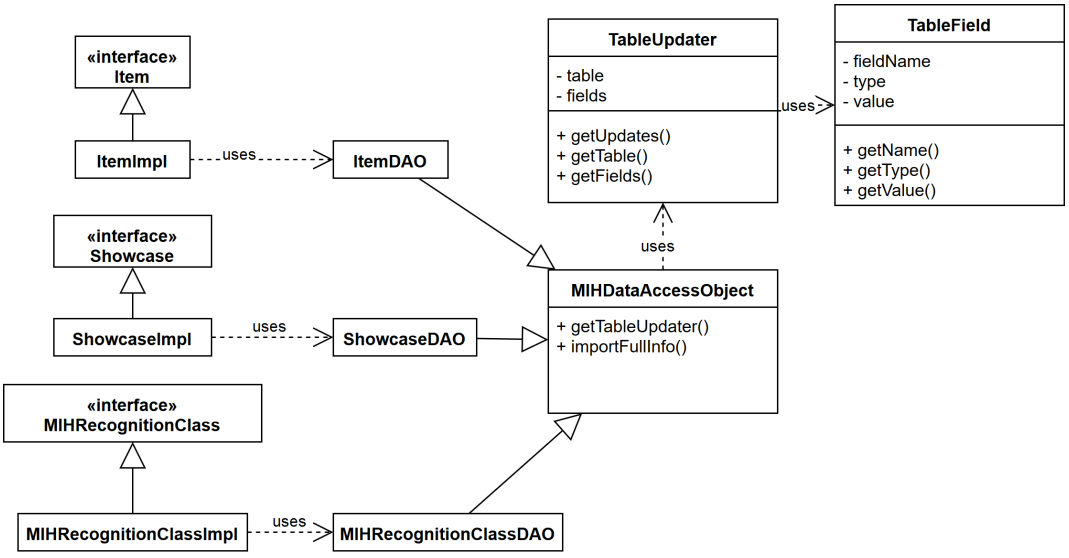



Figura 3.8: Ejemplos de clases de los componentes del museo y sus respectivos DAOs en MIHex.

### 3.3.2. Actualizaciones

En la figura 3.9 se muestran las interacciones entre la aplicación MIHex y la base de datos del servidor producidas durante el proceso de actualización.

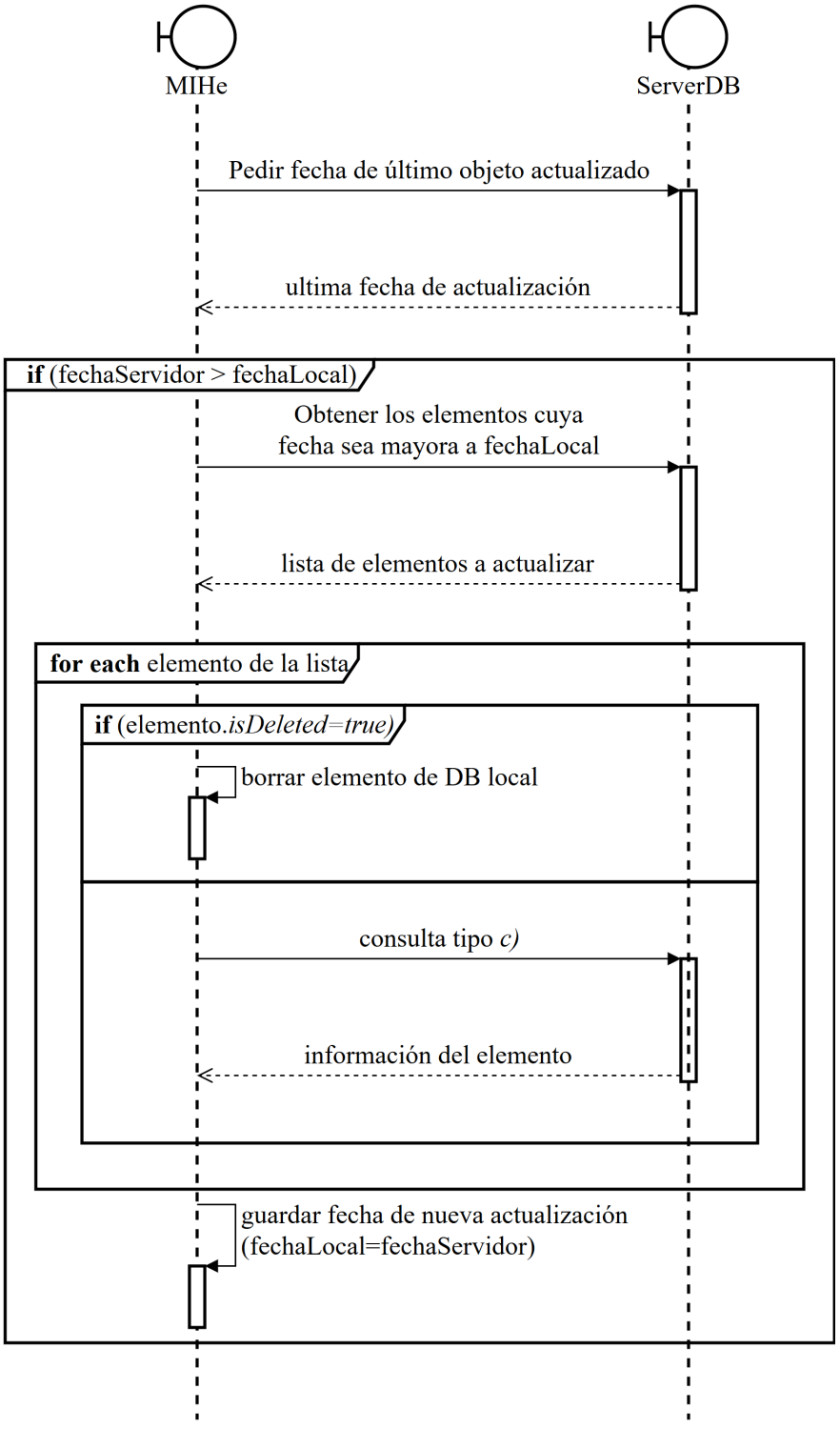
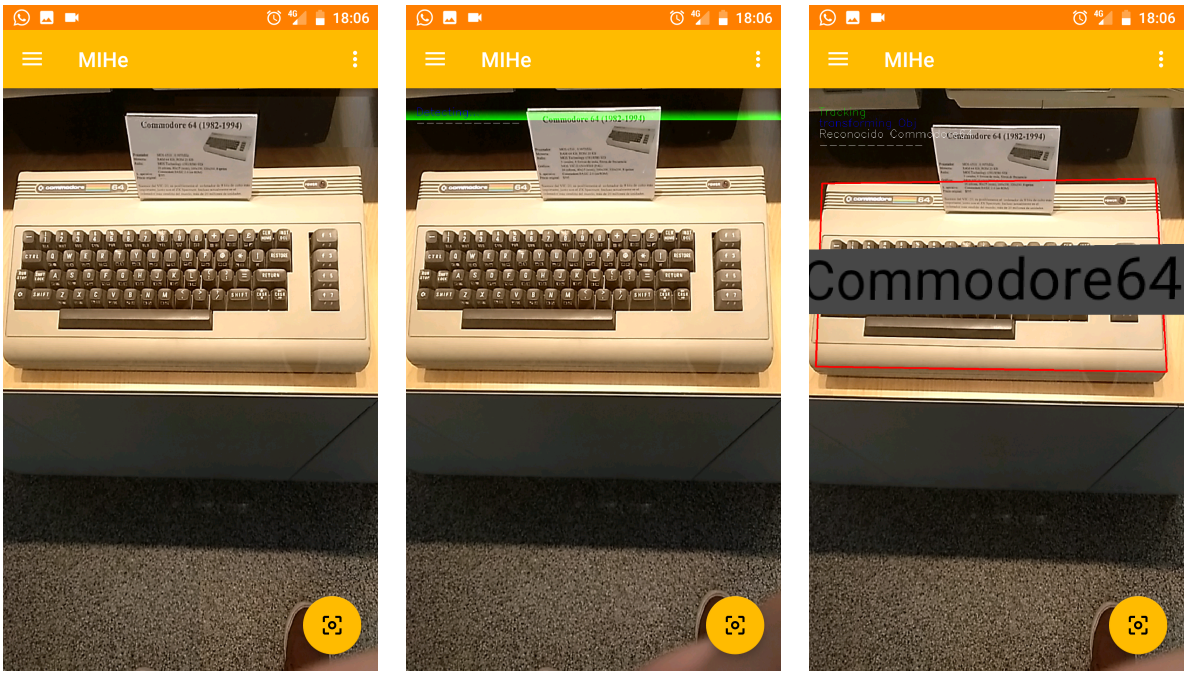



Figura 3.9: Diagrama de secuencia utilizado durante la actualización de MIHex.

# Capítulo 4

# Detección de objetos

El primer objetivo y más importante de la aplicación es la detección de objetos del museo para poder seleccionarlos y mostrar información detallada de los mismos. Es posible realizar este reconocimiento a través de la cámara del dispositivo, la cual proporciona imágenes a tiempo real. Sin embargo, tras realizar unas primeras tentativas de la tecnología que se iba a utilizar y analizar algunas aplicaciones existentes de reconocimiento de objetos, se concluyó que resultaría inviable realizar un reconocimiento para cada imagen de la cámara (ver anexo C). Por ello se adoptó la siguiente estrategia: permitir al usuario iniciar el reconocimiento al pulsar un botón. Este proceso se realizará en segundo plano, mientras que en la pantalla se muestra una animación para que el usuario tenga una experiencia más cómoda, tal y como se refleja en la figura 4.1. Una vez reconocidos los objetos y determinada su localización en la imagen tomada al momento de pulsar el botón, se realizará un seguimiento o *tracking* de los mismos en las sucesivas imágenes.




(a) Objeto antes de reconocerlo (b) Reconociendo (c) Objeto reconocido

Figura 4.1: Proceso de reconocimiento apreciado por el usuario.

## 4.1. Reconocimiento

Para afrontar la problemática del reconocimiento de objetos se decidió utilizar un modelo de bolsa de palabras visuales (BoVW, del inglés *Bag of Visual Words*)[1][2], que consiste en una variante del modelo de bolsa de palabras (BoW[1]) utilizado en el ámbito de la recuperación de información. Esta variante puede compararse con el modelo BoW realizando una analogía entre imágenes y documentos, así como entre fragmentos de la imagen y palabras. También se realizaron algunas modificaciones para intentar mejorar la eficiencia del modelo, que serán comentadas posteriormente. El algoritmo de reconocimiento se compone de dos partes:

- **Entrenamiento y creación de la bolsa de palabras**. En primer lugar es necesario determinar los objetos que queremos que el sistema sea capaz de reconocer, ya sean ítems del museo, estanterías o baldas. Para cada uno de ellos se definen:

  - Varias imágenes de reconocimiento. De cada una de ellas se obtendrán unos descriptores, que posteriormente supondrán las palabras de la imagen.
  - Una única imagen de detección, que servirá para validar geometricamente un objeto reconocido durante el proceso de reconocimiento.

  Cuando todos los objetos a reconocer tengan estos dos tipos de imagen se procederá a elaborar la BoVW. Para ello se agruparán todos los descriptores de todas las imágenes de todos los objetos mediante técnicas de aprendizaje automático. Posteriormente se obtendrá un descriptor núcleo para cada uno de los grupos, que supondrá una palabra visual. El conjunto de todas ellas compondrá el vocabulario de la mencionada bolsa de palabras.

  Por último, se generará un vector TF-IDF[2] para cada imagen de reconocimiento, contando las palabras que aparecen en cada una de ellas.

- **Reconocimiento o consulta a través de BoVW**. Cuando un usuario tome una instantánea con el objetivo de identificar los objetos que aparecen se llevará a cabo un proceso similar al de las imágenes de reconocimiento. Se extraerán los descriptores de la imagen y se obtendrán de la BoVW las palabras visuales que más se asemejen a cada uno de ellos. Con este conjunto de palabras se elaborará un vector TF-IDF, que se comparará con los de los objetos para determinar cúal es el más semejante. Si el vector de consulta es suficientemente similar al del objeto candidato se considerará este objeto como reconocido. En caso contrario, se llevará a cabo una validación geométrica para considerarlo y, si la validación no es correcta, se desechará este candidato y se continuará con el siguiente más parecido.

Estas partes se ven reflejadas en la figura 4.2 y serán descritas con mayor profundidad en las secciones siguientes.

[1] *Bag of Words*: Modelo que representa **documentos** de texto como conjuntos de **palabras** sin importar su orden.

[2] Vector cuyos componentes son el valor TF (*Term Frequency*, es decir, número de veces que aparece el término o palabra en la consulta) de la palabra multiplicado por su valor IDF (*Inverse Document Frequency*, es decir, el inverso del número de apariciones del término en la colección).

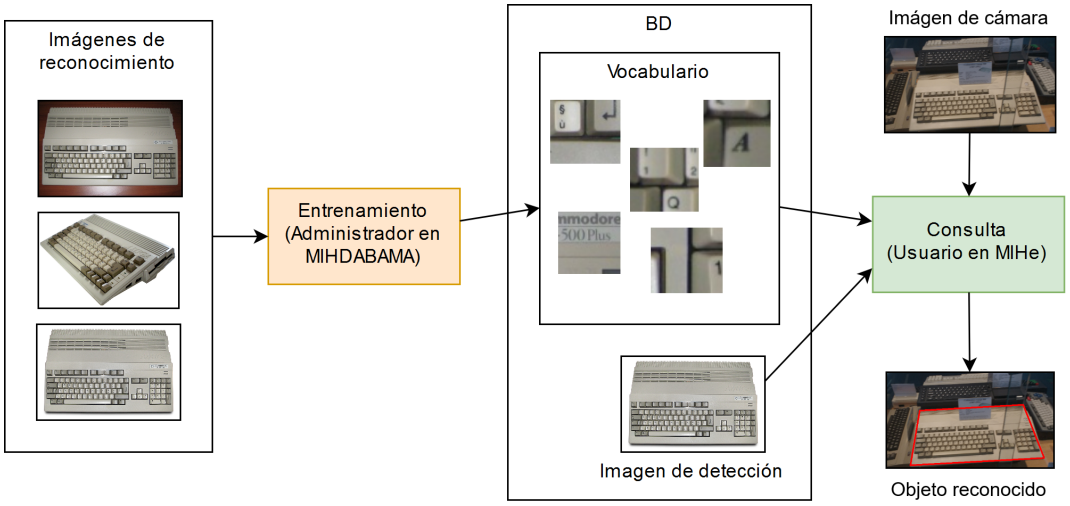



Figura 4.2: Esquema general del sistema de reconocimiento

Los descriptores mencionados anteriormente se refieren a descriptores de puntos de interés o *features*[3]. De entre todos los tipos existentes se consideraron los siguientes:

- Puntos SIFT: los emparejamientos realizados con estos puntos son los más precisos. Sin embargo, son bastante costosos computacionalmente y, además, están sujeto a licencia.
- Puntos ORB: son menos precisos, pero son más rápidos de calcular. No están sujetos a ninguna licencia.
- Puntos SURF: suponen una alternativa con una relación de precisión-eficiencia más equilibrada que las anteriores. Sin embargo, también es necesaria una licencia.
- Puntos AKAZE: suponen una segunda alternativa intermedia y, además, libre de licencia.
- Puntos BRISK: tercera alternativa intermedia y libre de licencia.

A priori se consideró que la mejor opción era utilizar los puntos ORB[4], ya que los restantes eran inviables en una ejecución a tiempo real. Tras realizar pruebas de los 4 puntos con un programa sencillo de reconocimiento en un ordenador personal, se concluyó definitivamente que se utilizarían los puntos ORB, ya que los resultados reflejaban un buen rendimiento y generalmente resultados satisfactorios. Sin embargo, con algunos objetos con poca información visual, los resultados no eran tan satisfactorios. Por ello, una vez se decidió utilizar la estrategia de realizar el reconocimiento en segundo plano se consideró la posibilidad de incluir el reconocimiento con AKAZE o BRISK, ya que el tiempo de ejecución dejó de ser tan crucial. Tras realizar pruebas con estos puntos se obtuvieron mejores resultados pero en mucho más tiempo, algo que supondría un grave impacto en una colección de mayor tamaño. Finalmente se adoptó la siguiente decisión: trabajar siempre con puntos ORB y utilizar puntos AKAZE o BRISK únicamente para aquellos objetos cuyos resultados con ORB no sean satifactorios y resulte crucial.

[3] Puntos localizados en una imagen con cierta información visual (p.e. esquinas). Los descriptores son conjuntos de información extraida de los píxeles circundantes al punto de interés asociado.

### 4.1.1. Entrenamiento y bolsa de palabras visuales

El entrenamiento se lleva a cabo desde la aplicación gestora MIHDatabaseManager. Lo ideal sería realizarlo una única vez, pues es un proceso costoso que requiere ser precomputado antes de realizar el reconocimiento, pero es posible repetirlo en caso de que el vocabulario de palabras actual no se adapte a determinadas circunstancias. Por ejemplo, si se ha añadido un número considerable de objetos nuevos (y por tanto de imágenes de reconocimiento) o si con nuevos objetos no se obtienen resultados satisfactorios.

Para cada objeto el administrador escogerá las imágenes de reconocimiento que considere convenientes. El sistema ofrece la posibilidad de mostrar los puntos clave de estas imágenes para comprobar las zonas que se utilizarán para reconocer la imagen. Durante la importación de las imágenes, si estas son demasiado grandes, se lleva a cabo un reescalado para limitar la resolución máxima a un cuadro de 1000x1000 píxeles.

Del mismo modo, el administrador añadirá la imagen de detección (ejemplo en figura 4.3d), y podrá validar su eficacia con las existentes imágenes de reconocimiento (ejemplos en la figura 4.3). En caso de que con puntos ORB no se obtengan resultados favorables, el administrador podrá seleccionar la opción de realizar la comprobación gométrica con AKAZE o BRISK. Durante el proceso de reconocimiento posterior, esta imagen de detección también es utilizada para definir los límites del objeto dentro de la imagen de la cámara, que en la pantalla se representarán con un cuadro rojo (ejemplo en la figura 4.1c). En ocasiones, la imagen puede contener zonas que no pertenecen al objeto, como el fondo. Para contemplar estos casos, es posible seleccionar manualmente los límites del objeto en la imagen de detección. Estos límites son los que enmarcarán al objeto en la imagen de la cámara tomada por el usuario, ofreciendo unos resultados más precisos. La misma zona que se define para enmarcar al objeto se utilizará también para filtrar los puntos de interés que estén fuera, pues no pertenecen realmente al objeto.


(a) Imagen de reconocimiento 1 (b) Imagen de reconocimiento 2 (c) Imagen de reconocimiento 3 (d) Imagen de detección

Figura 4.3: Ejemplo de imágenes reconocimiento y detección.

Una vez definidos las dos tipos de imágenes para todos los objetos que se pretenden reconocer, se puede proceder a la elaboración del vocabulario de la BoVW. Para ello se extraerán todos los descriptores de los puntos de interés de todas las imágenes de reconocimiento existentes. La cantidad de descriptores puede ser muy elevada ($500\,desc./imagen \times \sim$

$7\,imagenes/objeto \times \sim 2000\,objetos \simeq 7{,}000{,}000\,desc.$). Tal cantidad de descriptores resulta inviable, por lo que se agrupan en un número de *clusters* o grupos comprendido entre 5.000 y 10.000. Para ello se utiliza el algoritmo de aprendizaje automático *k-means*.

Sin embargo, los descriptores de ORB son binarios[4], por lo que el centroide de los grupos no puede ser calculado a través de una media de los descriptores de cada grupo. Como solución, la secuencia de bits del centroide se creará como una "votación" bit a bit de todos los integrantes del grupo, estableciendo el valor de cada bit como el más común entre los situados en la misma posición (ver figura 4.4). Esto supone una variante del *k-means* original, que se denomina *k-majority*. Tras converger el algoritmo, los centroides finales serán considerados como las palabras del vocabulario deseado.

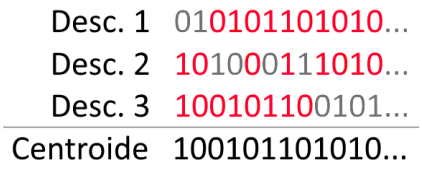



Figura 4.4: Ejemplo con 3 descriptores de obtención del centroide en el algoritmo *k-majority*.

Una vez creada la bolsa de palabras, será necesario describir cada objeto con las palabras del vocabulario. Para ello se creará un vector TF-IDF para cada imagen de reconocimiento, en el que cada elemento corresponderá a una palabra del vocabulario, y su valor será el valor TF-IDF de la imagen. A diferencia del algoritmo original, el sistema permite añadir varias imágenes por objeto, por lo que serían necesarios tantos vectores como imágenes. Como esto incrementa el número de comparaciones durante el proceso de reconocimiento, se decidió unificar los vectores de cada objeto en uno solo. Esto es posible realizando la media elemento a elemento. Se consideró que realizar esta acción no tenía repercusiones negativas, ya que las palabras más importantes (y por tanto comunes en casi todas las imágenes) seguirían teniendo un valor elevado, mientras que las menos importantes lo mantendrían bajo. Además, esto puede reducir valores espúreos, ya que una palabra considerada importante en una de las imágenes puede ser irrelevante para el resto (o viceversa), resultando en un valor TF-IDF final más aproximado.
En el la figura 4.5 se muestra a modo de resumen un esquema del proceso de entrenamiento.

Una vez obtenido el vocabulario de palabras y los vectores TF-IDF, es posible probar el sistema de reconocimiento en la herramienta MIHDatabaseManager antes de guardar los cambios en la base de datos. Para más información, ver el anexo E.

### 4.1.2. Reconocimiento a través de BoVW

Tras pulsar el botón de reconocer en la pantalla de MIHex, se procederá a identificar el objeto que aparezca en la imagen de la cámara en ese mismo instante. El primer paso es determinar qué palabras del vocabulario anteriormente entrenado aparecen en la imagen y su frecuencia. Para ello, se extraen los puntos y descriptores ORB tal y como se realizó

[4] Son descriptores definidos por una secuencia de bits y no por una serie de números, lo que hace que sean más pequeños y eficientes.

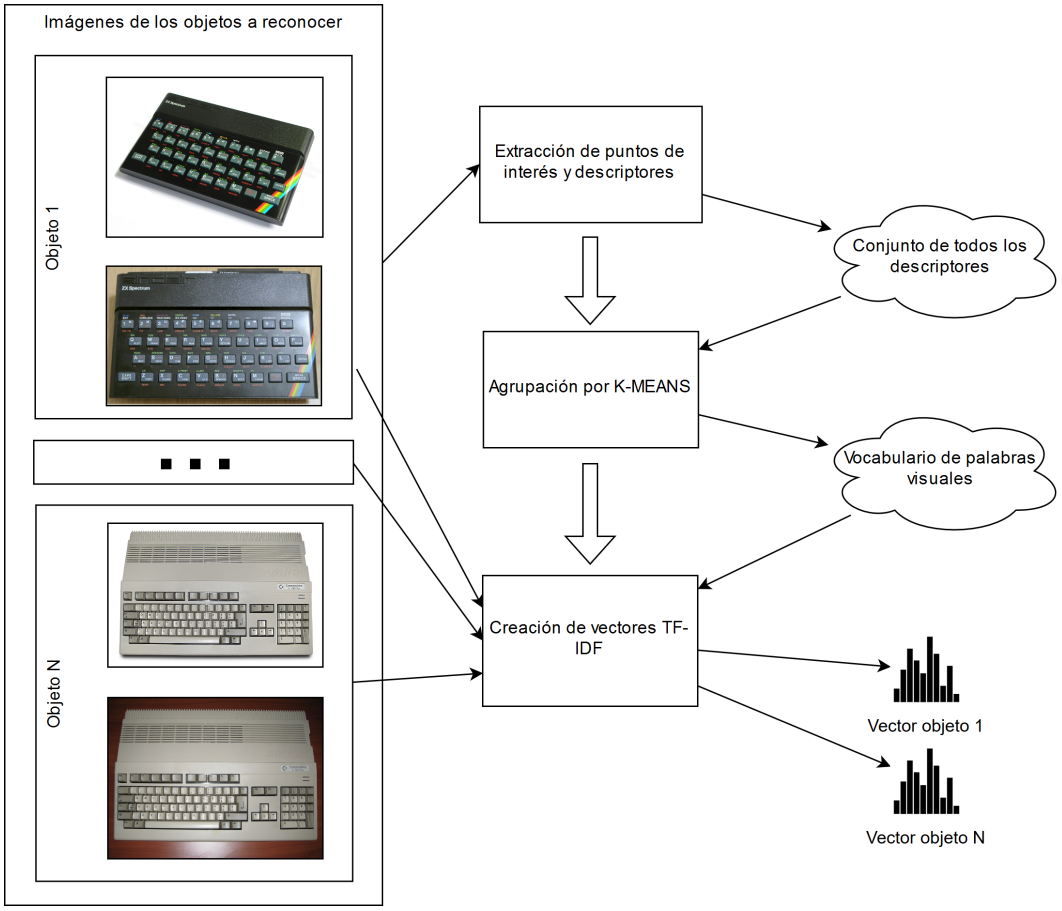



Figura 4.5: Esquema del entrenamiento de la bolsa de palabras

durante el entrenamiento. Cada uno de ellos se compara con todas las palabras visuales del vocabulario para seleccionar la más parecida (operación medida mediante la distancia de Hamming[5]), que se considerará incluida en la imagen y, por tanto, se incrementará su frecuencia. Como este proceso resulta muy costoso y repercute directamente en la experiencia del usuario, se ha puesto especial énfasis en su optimización, tal y como se describe en el anexo C.

Una vez determinadas la frecuencia de las palabras, se elaborará un vector TF-IDF de las mismas de igual manera que se realizaron en el punto anterior. Este vector $\vec{q}$ será comparado con los vectores $\vec{d}$ de todos los objetos del museo para obtener el más parecido. Para realizar esta comparación se utilizó el cálculo del coseno, pues el ángulo que forman ambos vectores determina lo parecidos que son. De esta forma, los objetos se ordenarán decrecientemente en función del resultado de la operación $\cos(\vec{q}, \vec{d})$ para su vector TF-IDF, lo que permite obtener los objetos del museo más parecidos a la imagen tomada.

Idealmente, el objeto más parecido debería ser realmente el reconocido en la pantalla. Esto se ha considerado así en el caso de que el valor del coseno obtenido con el objeto más parecido sea considerablemente superior al del segundo o supere un cierto umbral que lo garantice. En caso contrario, es necesario un segundo proceso para discriminar. El sistema de BoVW

[5] Dadas dos cadenas de bits, la distancia de Hamming se define como el número de bits que hay que cambiar para transformar una en otra, o lo que es lo mismo, el número de bits diferentes.

únicamente tiene en cuenta la aparición y frecuencia de palabras, pero no su localización en la imagen. En este momento entran en juego las imágenes de detección definidas para todos los objetos. Los puntos de interés y descriptores precomputados de las mismas se emparejarán con los de la imagen de la cámara, aplicando también una comparación al segundo más próximo para descartar los que no supongan un emparejamiento claro. Sin embargo, esto no garantiza que los emparejamientos sean correctos, por lo que se realizará un filtrado mediante el algoritmo de RANSAC[6] para validar geométricamente si la imagen de detección aparece realmente en la imagen de la cámara, es decir, si son homográficamente compatibles. El resultado de esta operación será un listado de emparejamientos correctos, denominados *inliers*. Si el número de *inliers* supera un cierto umbral, el sistema considerará el objeto como reconocido. En caso contrario, se procederá a repetir el proceso con los siguientes candidatos y, si en los 5 primeros no se obtiene un resultado concluyente, se declarará que no hay ningun objeto reconocido en la imagen.

Por último, solo resta mostrar el objeto reconocido al usuario. Para ello, a parte de la información mostrada mediante RA, se enmarcará el objeto reconocido en la imagen de la cámara. Al mismo tiempo que se hace un filtrado RANSAC se puede obtener una matriz de homografía que transforme los puntos de la imagen de detección en puntos sobre la imagen tomada. Gracias a esta matriz, las esquinas definidas manualmente en las imágenes de detección durante la fase de aprendizaje se pueden proyectar sobre la imagen de la cámara, lo que da lugar a las esquinas del marco que se pretende obtener. Con esto, el sistema de reconocimiento ha concluido, ya que las esquinas proyectadas son capaces de permanecer en las sucesivas imágenes de la cámara gracias al algoritmo de *tracking*.

## 4.2. Seguimiento o *tracking*

Una vez localizados los objetos en el *frame* inicial es necesario mantener su localización en los sucesivos *frames* de la cámara en tiempo real. Para efectuar esta acción se ha utilizado la misma tecnología que para el reconocimiento de objetos: el emparejamiento de *features*. En este caso se realizan emparejamientos entre los *frames* recibidas de la cámara, siempre desde un *frame* que contenga objetos detectados. Al establecer esta correspondencia se pueden localizar los mismos objetos contenidos en el *frame* antiguo en el nuevo, logrando así un seguimiento o *tracking* de los objetos.
Se han implementado dos tipos de *tracking*:

- Estático: Se realiza desde la imagen inicial, es decir, sobre la que se han detectado los objetos.

- Dinámico: Se realiza sucesivamente sobre las imágenes entrantes de la cámara, partiendo de una imagen inicial sobre la que se haya aplicado previamente *tracking* estático. Esta opción permite desplazar o rotar la cámara en mayor medida que el estático sin perder el *tracking*, pero puede producir más error.

[6] Random sample consensus. Modelo iterativo utilizado para obtener los parámetros de un modelo matemático, así como para distinguir los *inliers* (datos que encajan con el modelo y los parámetros obtenidos) de los *outliers* (valores atípicos) en los datos de entrada.

De este modo, una vez se ha realizado el *tracking* estático se continua con el dinámico, logrando mantener la posición del objeto en los *frames* sucesivos. Si no es posible continuar con el *tracking* dinámico, siempre se puede recuperar desde el *tracking* estático o desde el último *frame* procesado mediante el *tracking* dinámico. Del mismo modo, se puede intentar realizar el *tracking* estático siempre que se pueda para corregir las posibles deformaciones producidas por el *tracking* dinámico. En la figura 4.6 se muestra la máquina de estados que define este proceso.

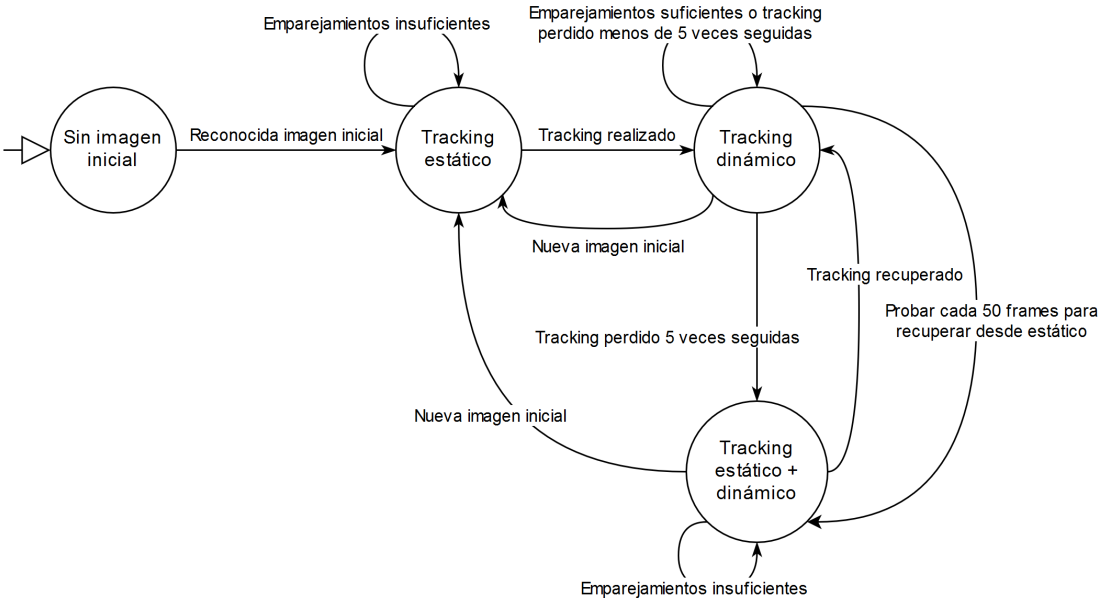



Figura 4.6: Máquina de estados implementada en el *tracking*

Realizar un *tracking* completamente estático consistente a escala requiere varios niveles de pirámide en la obtención de puntos ORB, lo que significa procesar la misma imagen a distintas escalas, pues el *frame* actual de la cámara puede variar bastante en escala respecto al *frame* sobre el que se realizó el reconocimiento. Esto supone un impacto significativo en el rendimiento a tiempo real. Sin embargo, para el *tracking* dinámico se parte de la premisa de que la diferencia de escala entre sucesivos *frames* de la cámara no ha de ser muy grande, es decir, los objetos que se vean en un *frame* no van a verse muy diferentes en el *frame* posterior. Por tanto, en este *tracking* se pueden reducir los niveles de la pirámide de los puntos ORB, lo que disminuye considerablemente el coste computacional.

# Capítulo 5

# Implementación

A continuación se expone cómo se ha llevado a cabo la implementación de la aplicación, detallando aquellas decisiones que se han considerado de importancia.

## 5.1. Tecnología

Para el desarrollo de la aplicación MIHex se dispone de un móvil con procesador Qualcomm Snapdragon 626 Octa Core de 2,2 GHz y sistema operativo Android 7.1.1. La herramienta MIHDatabaseManager se programará enteramente en Java, por lo que será multiplataforma. Por comodidad, durante el desarrollo del trabajo tanto MIHDatabaseManager como el servidor externo estarán instalados en un ordenador personal, con procesdor Intel Core i5 de 2,5 GHz. Para probar la RA en MIHDatabaseManager, también se llegó a utilizar una *webcam* externa.

El IDE escogido para elaborar la aplicación ha sido Android Studio 2.3.3, mientras que para la herramienta MIHDatabaseManager se hará uso del IDE Eclipse.
En cuanto a las bases de datos se han utilizado los siguientes SGBD[1]:

- MySQL en la base de datos del servidor, por ser gratuita y haber trabajado con ella previamente.
- SQLite en la base de datos local, pues Android incluye por defecto las librerías necesarias.

Por último, las librerías existentes que se utilizarán en el sistema son:

- OpenCV, librería que facilita operaciones para la visión por computador.
- OpenGL, librería que facilita el uso de gráficos a través de la GPU. Permite programar *shaders*[2] a través del lenguaje GLSL. Concretamente se utilizarán la versión GLES2.0 en Android y GL2 junto con JOGL y GLUEGEN en MIHDatabaseManager.
- JDBC, librería que facilita el acceso a las bases de datos desde Java.

## 5.2. Reconocimiento de objetos

Para realizar el algoritmo de BoVW, la librería openCV ofrece la clase *BOWKMeansTrainer*. Sin embargo, esta clase solo permite clusterizar mediante el algoritmo *k-means*, lo

[1] Sistemas de gestión de Bases de Datos

[2] Tipo de programa encargado de definir algunas propiedades de cómo se llevará a cabo el renderizado de un modelo tridimiensional en GPU.

cual no permite agrupar correctamente los puntos ORB por ser descriptores binarios. Es por esto que se optó por programar personalmente todo el proceso, incluyendo el algoritmo *k-majority*. Gracias a esto se puedieron aplicar modificaciones para experimentar con vías alternativas que pudieran favorecer el reconocimiento, como el uso de vectores TF-IDF en lugar de histogramas.

Pese a utilizarse el mismo algoritmo tanto en MIHDatabaseManager como en MIHe, ambas implementaciones difieren en varios aspectos:

- Lenguaje de programación. En MIHDatabaseManager únicamente se ha utilizado la librería openCV a través de la interfaz que ofrece para Java, lo que facilitó un desarrollo de la primera versión del algoritmo. Sin embargo, al llevar el mismo código a la aplicación móvil, el tiempo de ejecución aumentó de forma drástica. Es por ello que se optó por traducir el código implementado a C++ y utilizarlo desde la *app* a través de JNI.

- Cálculo de la distancia de Hamming. A la hora de determinar cuál es la palabra del vocabulario más similar a un descriptor, se usa como medida la distancia de Hamming. En MIHDatabaseManager se utilizó la función *norm* de openCV, suficiente para esta primera versión. Sin embargo, en la *app* MIHe, su uso tuvo mucho impacto en el tiempo de ejecución, ya que este cálculo es repetido aproximadamente 8 millones de veces (1000 descriptores x 8000 palabras). Con el fin de optimizar al máximo y evitar el uso de llamadas adicionales, se programó una versión optimizada de este cálculo. En el anexo C se explica detalladamente el proceso llevado a cabo y la optimización que supuso.

- Validación localización de objetos. Mientras que en la herramienta MIHDatabaseManager se realizan de forma secuencial, en MIHex se calculan en paralelo, lanzando un hilo por cada candidato. De esta forma, mientras todavía perdura la animación de reconocimiento, se irán agregando a la escena aquellos objetos que realmente aparezcan en la imagen. De esta acción también se ve beneficiado el tiempo de ejecución, pues al aplicarse en paralelo se aprovecha más el hardware del dispositivo.

## 5.3. Gráficos 3D

En un principio se pensó en utilizar los gráficos 3D únicamente en la vista detallada de los objetos, mostrando representaciones 3D de los mismos. Además, sólo se iban a incluir si no suponía ningún impacto en el tiempo de ejecución. Sin embargo, gracias a la librería de openGL, todos los cálculos de gráficos se pueden desviar a la GPU, lo que supone un impacto nulo en la CPU. Por este motivo, se decidió utilizar también los gráficos 3D para representar los objetos 3D de la Realidad Aumentada, ya que el CPU queda libre para dedicarse exclusivamente a la detección de objetos.

En los siguientes apartados se explica cómo se ha llevado a cabo cada parte dedicada a los gráficos.

### 5.3.1. *Renderer*

El *renderer*, implementado en la clase *MyGLRenderer*, es quien se encarga de coordinar todos los elementos gráficos que serán renderizados. Contiene una lista con todos los objetos

3D que figuran en la escena y que se dibujarán en la pantalla, así como un *shader* por defecto, que usarán los objetos de la escena si no cuentan con uno propio. En esta clase se activan además ciertos parámetros de openGL, necesarios para un correcto funcionamiento. Estos son:

- *Culling*: Establece que se dibuje una sola cara de los triángulos a renderizar. La cara ha de ser la que ofrezcan los modelos en sentido contrario a las agujas del reloj para que se renderice la cara correcta. Esto hace que se eviten cálculos innecesarios obteniendo los mismos resultados.

- *Depth Test*: Es necesario que esté activado para que, durante la rasterización, se evalúen qué *fragments* están delante de otros y por tanto determinar su visibilidad.

### 5.3.2. Matriz de proyección

La matriz de proyección o *frustum* determina cómo será la proyección de los objetos de la escena en la pantalla. Puesto que se desea que estos objetos virtuales se proyecten de la misma manera que se proyectan los objetos reales vistos desde la cámara, dicha matriz se ha de definir utilizando algunos de los parámetros intrínsecos de la cámara[3]. Se ha utilizado la siguiente implementación:

$$M_{proj} = \begin{bmatrix} \alpha/c_x & 0 & 0 & 0 \\ 0 & \beta/c_y & 0 & 0 \\ 0 & 0 & \frac{-(Z_f+Z_n)}{Z_f-Z_n} & \frac{-2*Z_f*Z_n}{Z_f-Z_n} \\ 0 & 0 & -1 & 0 \end{bmatrix},$$

donde:

- $\alpha$ y $\beta$ ambas representan la distancia focal de la cámara. Pertenecen a los parámetros intrínsecos de la cámara y se pueden aproximar a la anchura o altura de la imagen;

- $c_x$ y $c_y$ es el centro de la imagen de la cámara (en píxeles).

- $Z_n$ y $Z_f$ son los planos cercano y lejano que definen el *frustum* de openGL. Se le han dado unos valores arbitrarios de 10 y 10000, pues es recomendable que el valor de $Z_f$ sea 1000 veces mayor que el de $Z_n$ para mantener una precisión del z-buffer adecuada. Valores $Z_f < 10000$ hacían que, cuando la cámara se aleja de los objetos físicos durante el modo de RA, su representación en 3D o etiquetas se salieran del *frustum* y, por tanto, desaparecieran de la escena.

Esta matriz únicamente es necesario inicializarla cada vez que cambia la orientación de la cámara, pues esto afecta a los parámetros intrínsecos de la cámara y, por tanto, a la proyección a utilizar. En el código se ha definido esta matriz pero en su versión transpuesta, pues openGL realiza la lectura por columnas.

[3] La relación entre la cámara de openCV y la matriz de proyección de openGL se ha obtenido de la siguiente dirección (último acceso 18/09/2018):
http://kgeorge.github.io/2014/03/08/calculating-opengl-perspective-matrix-from-opencv-intrinsic-matrix

### 5.3.3. Matriz de vista de la cámara

La matriz de vista de la cámara determina la posición y rotaciones de la misma respecto a las coordenadas definidas en openGL. Se crea a partir de la posición de la cámara, un punto hacia el cual estará orientada y un vector que determina la inclinación.

Puesto que la posición y rotación de los objetos son obtenidas respecto a la cámara, conviene situar esta en el eje de coordenadas, evitando así realizar más operaciones. Se coloca además orientada hacia el eje Z en sentido positivo, y con el vector de inclinación apuntando al eje Y en sentido positivo (hacia arriba, lo que significa que no hay inclinación). Por tanto, la matriz de vista de la cámara queda definida de la siguiente manera:

$$M_{view} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} \hat{P} \\ \hat{D} \\ \hat{I} \end{bmatrix} ,$$

donde $\hat{P}$ es la posición de la cámara, $\hat{D}$ es su dirección e $\hat{I}$ es su inclinación.

Esta matriz únicamente es necesario iniciarla una sóla vez. Esto se debe a que la cámara permanecerá estática y serán los objetos de la escena los que se muevan o roten en torno a ella. Esto está explicado con mayor detalle en el apartado de Realidad Aumentada (Ver sección 5.4).

### 5.3.4. Modelos 3D

Son los objetos tridimensionales que se dibujan a través del *renderer* en la pantalla. Se generar en 3 pasos:

1. Creación: A través del constructor se les pasa la ruta del fichero OBJ, la textura ya importada que utilizará y el *shader* que utilizará (por defecto el del propio *renderer*). El sistema solo admite una textura por objeto.

2. Carga del fichero OBJ: Se importan las coordenadas, las normales y las UVs almacenadas en el fichero OBJ indicado. Para más detalles ver la sección 5.3.5

3. Generar buffers y enviarlos a la GPU: Una vez importadas las coordenadas, las normales y las UVs se introducen en los *buffers* para cargarlos en la tarjeta gráfica. En este proceso también se envían a la GPU la textura y los *shaders* que va a utilizar el modelo (Ver sección 5.3.7).

Una vez completados estos pasos, el modelo está listo para ser renderizado a través del *renderer*.

### 5.3.5. Importar archivos OBJ

Se ha creado un importador de archivos OBJ propio. De esta forma no se requieren librerías de terceros y, además, se cargarán únicamente los valores que interesen. El sistema da soporte para almacenar varias texturas por modelo, pero solo se llegó a aplicar funcionalmente una única textura sobre los modelos, la que define el albedo[4].
El procesamiento se hace por líneas:

[4]Color básico de un material

- Si comienza por 'v' los siguientes 3 valores se cargarán como las coordenadas de un vértice. Se almacenarán en un vector de ternas (ver Coords en figura 5.1). Al mismo tiempo que se leen las coordenadas de los vértices se calcula también cuál es el máximo y mínimo en los ejes X e Y, para así poder determinar el centro de coordenadas como el eje relativo del objeto.
- Si comienza por 'vn' los siguientes 3 valores se cargarán como las normales de un vértice. Se almacenarán en un vector de ternas (ver Normales en figura 5.1).
- Si comienza por 'vt' los siguientes 2 valores se cargarán como las UVs[5] de un vértice. Se almacenarán en un vector de pares (ver UVs en figura 5.1).
- Si comienza por 'f' los siguientes valores representan los 3 vértices que componen una cara. Cada vértice está representado en una terna de 3 indices: el que apunta a las coordenadas de su vértice, el que apunta a sus normales y el que apunta a sus UVs. El orden en el que están indicados los vértices será también el orden que tome openGL para dibujarlos. Se almacenarán tanto el orden de las coordenadas de los vértices como el de sus normales y UVs, tal y como se muestra en la figura 5.1.

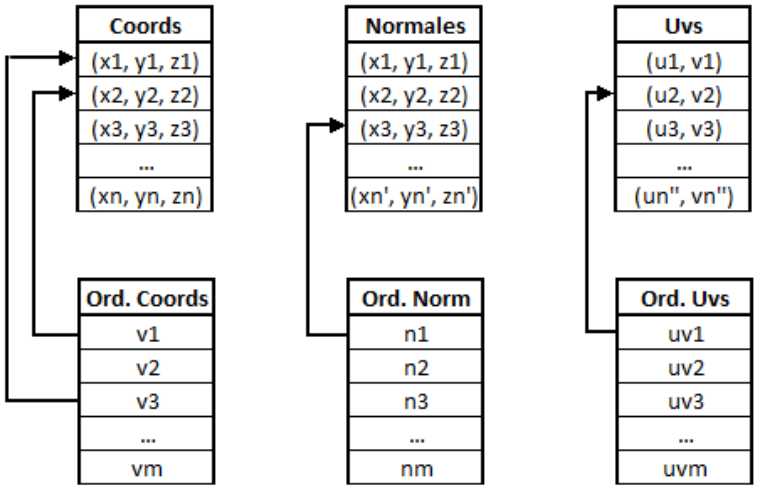



Figura 5.1: Ejemplo de *arrays* utilizados durante la carga del modelo 3D.

Una vez leído el fichero OBJ y obtenidos los vectores de valores y de orden, se obtiene la escala del objeto para que al aplicarla el objeto quepa dentro de un cubo 1x1x1 y pueda transformarse más tarde a cualquier otra escala. Posteriormente se construyen los vectores que serán enviados a la GPU:

- El *vertexData* incluye la información de cada vértice de forma consecutiva: las 3 coordenadas del vértice, los 3 valores de la normal y los 2 valores de las coordenadas UV. El orden en el que se introducen es el mismo que llevan las coordenadas del vértice en el *array* de valores. De esta forma, el indexado coincide con el del *array* de orden de coordenadas. En la figura 5.2 se muestra cómo se genera este vector.

[5] Coordenadas sobre una textura que tomará un vértice. De esta forma, durante el proceso de renderizado, una cara de tres vértices se coloreará con el triángulo que definan sus 3 vértices sobre la textura.

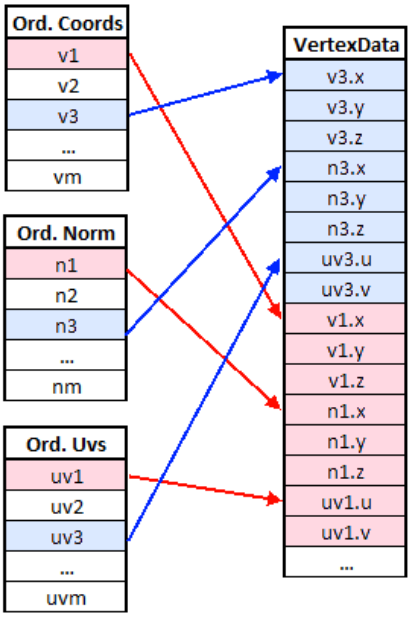



Figura 5.2: Ejemplo de cómo se construye el *vertexData*, siguiendo con el ejemplo de vectores de valores y de orden de la figura 5.1

- El *drawOrder* incluye el orden en el que se van a renderizar los vértices. Se obtiene únicamente a partir del *array* de orden de coordenadas.

### 5.3.6. Etiquetas

Las etiquetas se han definido como un subtipo de los modelos 3D, y por tanto se pueden renderizar a través de openGL como tales. Poseen un modelo 3D propio con forma de plano, que una vez definida la textura se escala para que posean la mismas dimensiones y el texto no se vea deformado. La textura es creada dinámicamente al inicializar la etiqueta a partir del nombre del objeto reconocido o de la descripción que se le defina. En el caso de que la descripción ocupe más de 2 líneas, se mantendrá el tamaño de la etiqueta, pero dicha descripción se desplazará a modo de *scroll* gracias a un desplazamiento de las coordenadas UV de la textura. Con esto es posible mostrar mucha más información en un espacio reducido.

Puesto que las etiquetas aparecen y desaparecen con frecuencia de la pantalla, únicamente se inicializan una sóla vez. Cuando desaparecen de la pantalla, en vez de destruirse se dejan en un estado invisible por si vuelven a aparecer.

### 5.3.7. *Shaders*

A través de los *shaders* se indica a la GPU cómo ha de procesar la información enviada a través de los *buffers* para crear los gráficos. Puesto que su procesado se lleva a cabo en la GPU, conviene enviarle la mayor cantidad de información posible (como la multiplicación de matrices de transformación, cámara, etc.) para librar al CPU de estas tareas. Como en la mayoría de *pipelines* gráficos, están definidos dos tipos de *shaders*:

- El *vertex shader*: se aplica sobre cada vértice. Recibe como atributos los valores especificados a través del *buffer vertexData*, es decir, las 3 coordenadas de posición del vértice, las 3 coordenadas del vector normal y las 2 coordenadas UVs. A través de las variables uniformes se le añaden además las matrices de transformación que afectarán tanto a las posiciones de los vértices como a las normales. Primero se aplican las transformaciones definidas explícitamente para el modelo[6] y, posteriormente, el escalado, rotación y traslación obtenidos para el posicionamiento en RA, seguido de la vista-proyección. Una vez aplicadas las transformaciones sobre la posición del vértice y su normal se envían junto con las coordenadas UVs sin tratar al *fragment shader* a través de las variables intermedias. El *shader* produce además una salida propia con la posición transformada del vértice que se envía internamente al *pipeline* gráfico.

- El *fragment shader*: se aplica sobre cada *fragment* producido durante la rasterización. Recibe como entrada las variables intermedias producidas por el *vertex shader*. Además, recibe a través de las varibles uniformes la posición y color de un único punto de luz y la textura que se aplicará. En este *shader* se lleva a cabo el grueso del proceso de *shading*, encargándose de obtener para cada fragmento su color (o albedo) y su iluminación: el primero lo obtiene interpolando las coordenadas UV de los vértices a los que pertenece ese fragmento; la segunda viene definida por la siguiente BRDF lambertiana, la cual es similar a la BRDF de Phong pero sin el componente especular:

  $$I_p = k_a i_a + k_d(\hat{L} \cdot \hat{N})$$

  De esta forma, la iluminación viene dada por un componente constante ($k_a$, que multiplicado con la luz ambiente $i_a$, simula la luz indirecta) y por un componente difuso, $k_d$, que es multiplicado por el coseno[7] entre la normal y el vector que va desde las coordenadas interpoladas del fragmento hasta el punto de luz definido. El color calculado para el fragmento a través de dicha función es enviado a la salida propia del *shader*.

En la figura 5.3 se muestra la interacción que se lleva a cabo entre los *shaders*. A la izquierda se muestran los atributos, que son los especificados en el *vertexData*. En la parte superior se muestran las variables uniformes. Entre los *shaders* se muestran la información que se envía del *vertex shader* al *fragment shader*. En la parte inferior se muestran las salidas propias de cada *shader*.

## 5.4. Realidad Aumentada

El proceso final para lograr la Realidad Aumentada es combinar los resultados de la visión por computador realizada por openCV y los gráficos producidos a través de openGL. Esta combinación debe de responder a dos preguntas:

[6] Conjunto de transformaciones de traslación, rotación y escala que se aplican antes que el resto de transformaciones. Se utiliza para definir dónde se va a considerar el origen del modelo y su rotación y escala iniciales. Podrán ser absolutas o, en el caso de corresponder a modelos de objetos virtuales, relativas al modelo de la estantería en la que se encuentren.

[7] Puesto que ambos componentes $\hat{L}$ y $\hat{N}$ están normalizados: $\hat{L} \cdot \hat{N} = |\hat{L}||\hat{N}| \cos\alpha \equiv \cos\alpha$

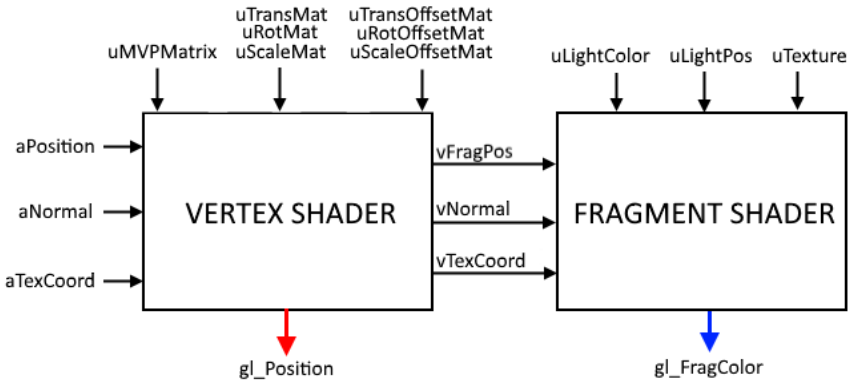



Figura 5.3: Parámetros e intercambio de información de los *shaders*.

- ¿Qué mostrar? Una vez determinados qué objetos aparecen en la imagen a través del reconocimiento de openCV se crean las etiquetas 3D correspondientes a dichos objetos (ver sección 5.3.6) y se añaden a la lista de objetos en escena del *renderer*. Existen determinados objetos de los que interesa añadir a esa lista el modelo 3D del objeto en vez de la etiqueta, como la estantería virtual.

- ¿Cómo mostrarlo? El objetivo de la RA es que parezca que los objetos ficticios están realmente en la imagen, y para ello se deben colocar en la misma perspectiva que la imagen, lo cual implica determinar dónde está la cámara respecto a la imagen. Este problema es conocido como *Perspective-n-Point*.

A continuación se describen las respuestas que se les ha dado a estas dos preguntas.

### 5.4.1. El problema *Perspective-n-Point*

Una vez obtenida la matriz de homografía y calculadas las esquinas del objeto en la imagen, es posible determinar la localización de la cámara respecto a ese objeto. Obtener estos valores se conoce como el problema de *Prespective-n-Point*, y openCV aporta la función *solvePnP* para facilitar esta tarea. A esta función es necesario pasarle los parámetros intrínsecos de la cámara, y ésta devuelve los parámetros extrínsecos[8], es decir, el vector de traslación y el vector de rotación que determinan la posición y la rotación de la cámara. Invirtiendo el sentido de estos vectores podemos obtener la traslación y rotación de un objeto respecto a la cámara.

Por último, es necesario enviar estos vectores a openGL para que aplique las transformaciones de las etiquetas. Respecto al vector de rotación se pensó inicialmente en obtener los ángulos de Euler para, posteriormente, construir la matriz de rotación, pero en la librería de openCV existe la función *Rodrigues*, que transforma directamente el vector de rotación en una matriz de rotación, lo que nos ahorra este paso intermedio. Gracias a la matriz de proyección, las distancias obtenidas de openCV concuerdan con las que se utilizarán en openGL,

[8] Se ha adaptado la configuración de la siguiente dirección (último acceso el 18/09/2018) https://www.learnopencv.com/head-pose-estimation-using-opencv-and-dlib/

por lo que no es necesario aplicarles ningún otro cambio. El vector de traslación y la matriz de rotación se pasan al objeto, que los utilizará dentro del *shader* en la fase de renderizado.

### 5.4.2. Vistas

Antes de finalizar la sección de Realidad Aumentada es necesario especificar cómo se visualizan los resultados conjuntos en la pantalla.

Por un lado, el previo de la imagen se muestra en la vista implementada de openCV. Por otro lado, los gráficos producidos por openGL también han de ser mostrados en una vista de Android. Para ello se ha creado una clase que extiende la vista que openGL proporciona para dicha labor.

Para crear el efecto de realidad aumentada se superpone la vista de openGL a la de openCV, configurando sus dimensiones para que sean iguales y, preferiblemente, ocupen el mayor espacio en pantalla posible. Estableciendo un fondo transparente para la vista de openGL se consigue que los gráficos se muestren superpuestos a la imagen de la cámara, creando el efecto de la realidad aumentada.

## 5.5. Gestión de datos

En un principio se ideó la herramienta MIHDatabaseManager con un uso minimalista: introducir y modificar los datos de los objetos en la base de datos del servidor por parte del administrador, cuando se utilizaba todavía la primera versión de la base de datos (ver anexo B). Sin embargo, al decidir incorporar las nuevas funcionalidades de la segunda iteración (ver sección 2.3), se decidió extender también las funcionalidades de esta herramienta:

- Gestionar toda la estructura del museo (objetos, baldas, estanterías...).
- Gestionar todas las imágenes requeridas durante el entrenamiento del reconocimiento (ver sección 4), así como generar todos los datos necesarios para este proceso.
- Implementar el reconocimiento. En ocasiones, programar la *app* MIHex directamente sobre el dispositivo móvil resultaba tedioso debido a la lentitud de compilación e instalación, mientras que en MIHDatabaseManager era casi instantáneo. Por ello, se realizó una primera implementación del reconocimiento en la herramienta para posteriormente llevarla a la *app*, ahorrando así mucho tiempo. Además, esto permite probar el sistema entrenado antes de llevarlo a la *app*, pudiendo apreciar si el resultado es el deseado y no se han cometido errores.
- Gestionar los elementos gráficos de la Realidad Aumentada. En la mayoría de ocasiones se desea que una estantería virtual esté en una determinada posición respecto a la imagen a reconocer. Por ello, el sistema permite modificar la posición, rotación y escala de la estantería para poder ajustarla con precisión y comodidad. En el caso de los objetos virtuales, están disponibles las mismas opciones para colocarlos en una estantería virtual a gusto del administrador.
- Exportar toda la base de datos a un fichero, con el fin de ser leído desde la *app* MIHex en un orden controlado. Obtener todos los datos de la BD externa desde el inicio

es un proceso costoso, por lo que se decidió que la aplicación poblaría su base de datos interna desde este fichero, que es incluido dentro del APK[9]. De esta forma, la aplicación accederá a la BD del servidor únicamente para las futuras actualizaciones. Para garantizar que la lectura/escritura de este fichero se realiza de forma consistente en ambas aplicaciones, las clases que implementan esta funcionalidad se han incluido dentro del paquete *commons*.

[9] ***A****ndroid a****p****plication pac****k****age*. Archivo de paquetes de Android, utilizado en este caso para instalar la aplicación.

# Capítulo 6

# Resultados

A continuación se exponen algunos de los resultados obtenidos tras la finalización del proyecto. Los principales aspectos a analizar tras el desarrollo de la aplicación son el reconocimiento y el *tracking* de los objetos, ya que son las funcionalidades principales de la aplicación. Además han resultado los más problemáticos, pues son dependientes de las condiciones expositivas del museo (posicionamiento de los objetos en las vitrinas, iluminación, sombras, reflejos, etc.). Los resultados visuales de la app MIHex y la herramienta MIHDatabaseManager pueden apreciarse en los anexos D y E como parte del manual de usuario.

## 6.1. Cumplimiento de los requisitos

Se lograron satisfacer todos los requisitos de la primera iteración, a excepción de:

- RNF4 - "El sistema estará terminado para la primera semana de septiembre de 2017", pues los resultados todavía no eran satisfactorios. Además, al elaborar los nuevos requisitos se extendió todavía más el tiempo previsto.

- También se podría considerar que el RF3 "El sistema permitirá al usuario descargar la última versión de la base de datos de la colección (o posponer) al iniciar la aplicación o a través del menú" no se llevó a cabo al 100 %, ya que únicamente se llegó a desplegar la base de datos, en un ordenador personal, donde funcionó sin problemas. Si se deseara implementarlo en otra base de datos de otro servidor, bastaría con crear las bases de datos mediante los mismos *scripts*, definir la dirección del nuevo servidor tanto en MIHDatabaseManager como en MIHex y migrar el contenido a través del fichero exportable desde MIHDatabaseManager.

En cuanto a los requisitos de la segunda operación, hubo dos que no se lograron implementar correctamente. Estos son:

- RF11 - "El sistema será capaz de reconocer baldas de estanterías reales, identificando los objetos que en ellas se encuentren". Se logró que el sistema contemplara la aparición de ítems en baldas, tanto reales como virtuales. También se implementó el reconocimiento de baldas, pero los resultados obtenidos no fueron satisfactorios, ya que originaban muchos falsos positivos. Es por esto que se decidió descartar esta opción.

- RF12 - "El usuario podrá seleccionar objetos virtuales situados en estanterías virtuales para visualizar su información detallada". Se llegó a admitir en el sistema y en la MIHDatabaseManager la posibilidad de incluir ítems en las estanterías virtuales. Sin embargo, por falta de tiempo no se pudo incluir en la versión final de la aplicación.

## 6.2. Reconocimiento

El reconocimiento supuso el mayor reto de este TFG, tanto por ser el ámbito más complicado como por las restricciones de tiempo y memoria que requería. Dentro de la aplicación es el módulo más importante, ya que su funcionamiento es el mayor condicionante del grado de satisfacción del proyecto. Por ello, son estos los resultados que más se deseaba perfeccionar, y pueden dividirse en resultados de eficiencia y resultados de eficacia.

### 6.2.1. Resultados de eficiencia

Tras aplicar todas las optimizaciones especificadas en el anexo C, para una colección de 50 objetos, se consiguió un tiempo de reconocimiento que suele rondar los 2 segundos. Este tiempo es consecuencia de una serie de operaciones, que pueden apreciarse en la figura 6.1. En esta figura se aprecian dos casos en términos de eficiencia: el caso mejor, en el que el no se reconoce el objeto y no realiza la operación de localizarlo en la imagen; y el caso peor, en el que es reconocido y por lo tanto ha de realizarse este paso. Sin embargo, en terminos de eficiencia, se produce un "caso mejor" cuando no se reconoce ningún objeto, pues se omite el paso de localizarlo en la imagen.

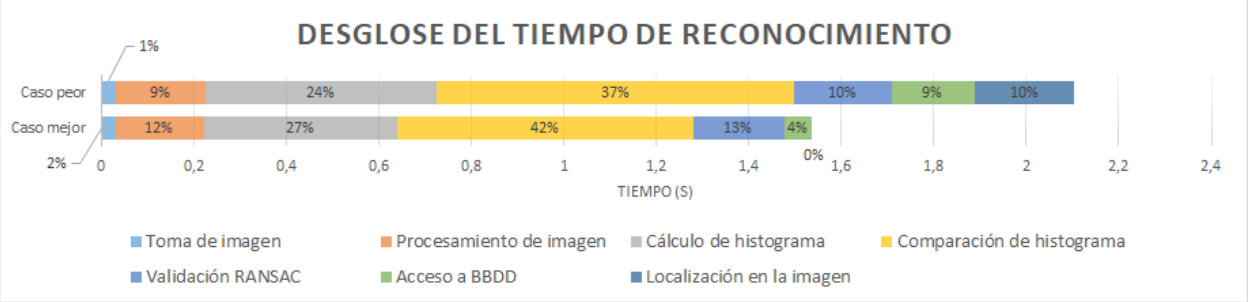



Figura 6.1: Desglose del tiempo de reconocimiento.

Entre todas ellas, la operación más costosa es la comparación de histogramas, tal y como puede apreciarse también en la figura 6.2. Esto es debido a que, por falta de tiempo, no se le llegó a aplicar una optimización exhaustiva como al resto. Las operaciones de validación por RANSAC, acceso a BD y localización en la imagen son realizadas en paralelo para cada objeto candidato, por lo que en estas figuras se muestran los tiempos de un solo elemento.

### 6.2.2. Resultados de eficacia

Tras aplicar un entrenamiento para 47 objetos, en la herramienta MIHDatabaseManager se realizó un test sobre 52 imágenes de prueba, en el que se obtuvo un acierto del 80 % (42 objetos fueron recuperados). Una vez llevados los datos a MIHex, se apreció que, sin alterar la configuración del sistema entrenado, uno de los factores más relacionados con la precisión es el espacio que ocupa el objeto en pantalla. Esto es debido a que, cuanto más se ajuste el objeto a la pantalla, más se parecerá la imagen de la cámara a las de reconocimiento/detección de dicho objeto, obteniendo mejores resultados. En la figura 6.3 se puede apreciar esta relación, mostrando, para cada ocupación en pantalla (medida en %), la puntuación o *score* obtenida al comparar el histograma de la imagen con el objeto reconocido y el número de *inliers*

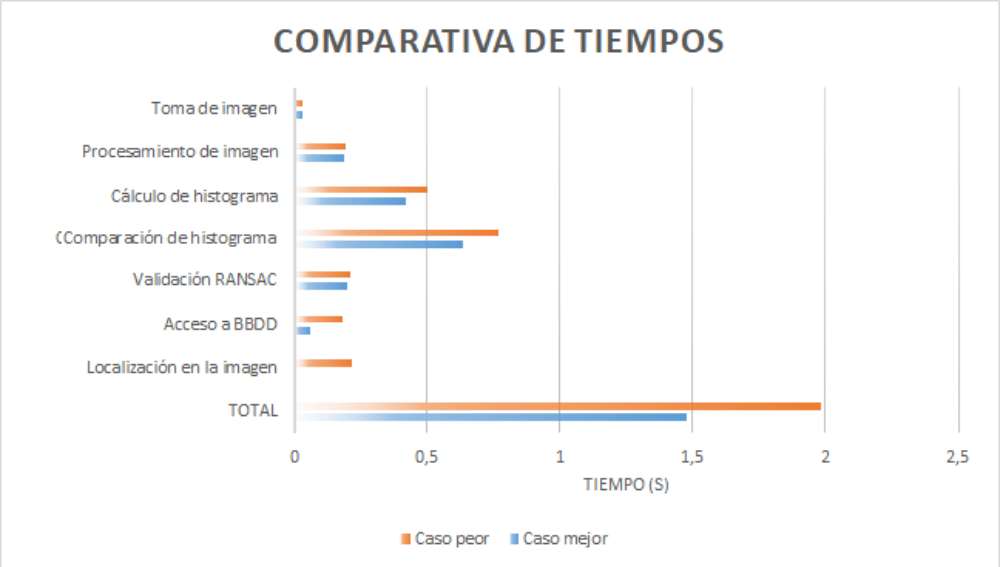



Figura 6.2: Comparativa de tiempos entre las acciones de reconocimiento.

tras aplicar la validación geométrica. De ella podemos concluir que los mejores resultados se obtienen con una ocupación en pantalla entre el 70 % y el 110 %.

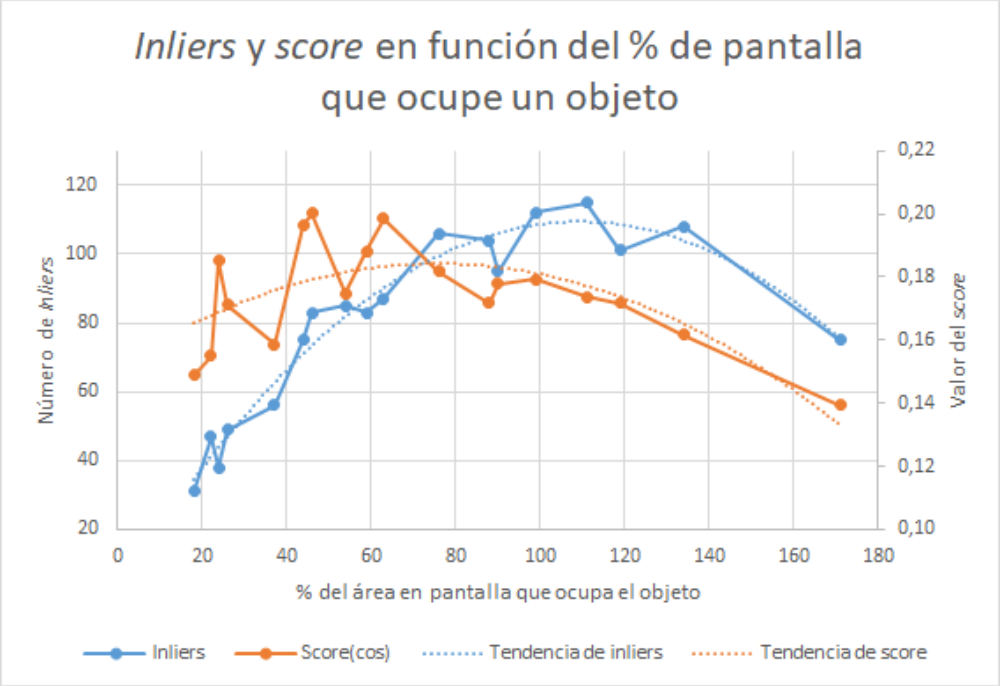
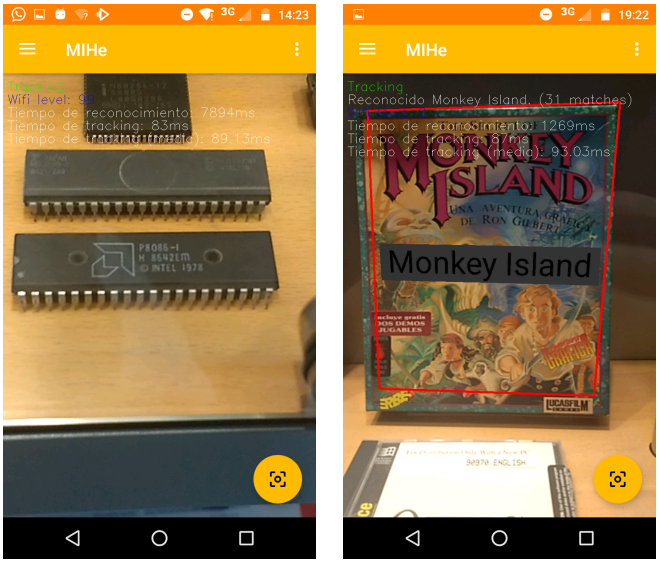



Figura 6.3: *Inliers* y *score* en función del % en pantalla que ocupe un objeto.

También es determinante la cantidad de información visual que proporciona un objeto, siendo mucho más dificiles de reconocer aquellos cuya mayor parte de la superficie no tiene detalles, tal y como se ve en la figura 6.4.

Otro factor que condiciona enormemente al reconocimiento es el fondo tras el objeto. Si este tiene mucha información visual, es posible que el sistema le dé más importancia que al

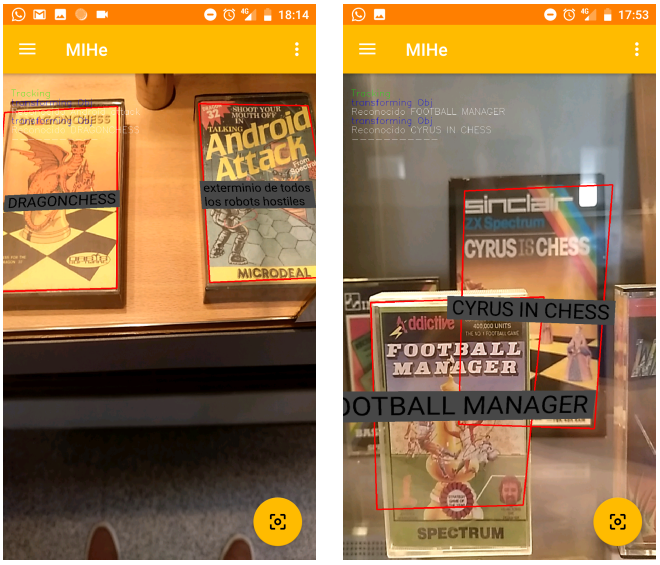



(a) Objetos con poca información visual

(b) Objeto con mucha información visual

Figura 6.4: Objetos con distinta cantidad de información visual

objeto en sí mismo, seleccionando aquellos puntos de interés que en él aparezcan. También puede ocurrir que se muestre más de un objeto reconocible en pantalla (ver figura 6.5). En este caso, el sistema puede ser capaz de reconocer a varios de ellos. Sin embargo, existen menos garantías de éxito, ya que si aparecen varios objetos en pantalla el % de ocupación en pantalla de cada uno será más reducido, situación en la que el sistema es menos preciso.



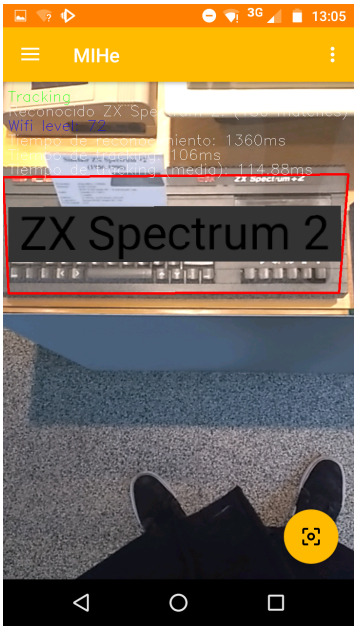

(a) Ejemplo de dos objetos contiguos encontrados en la misma imagen

(b) Ejemplo de dos objetos superpuestos encontrados en la misma imagen

Figura 6.5: Ejemplo de reconocimiento múltiple

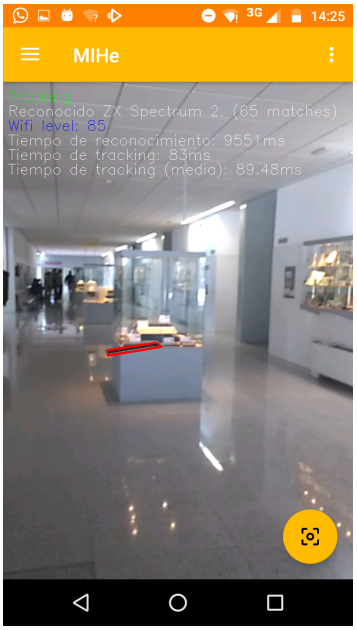
## 6.3. *Tracking*

En la figura 6.6 se muestran ejemplos de resultados obtenidos durante un ciclo habitual de *tracking*.




(a) Se reconoce el objeto y se localiza mediante el *tracking* estático


(b) Se mantiene el objeto en la imagen gracias al *tracking* dinámico



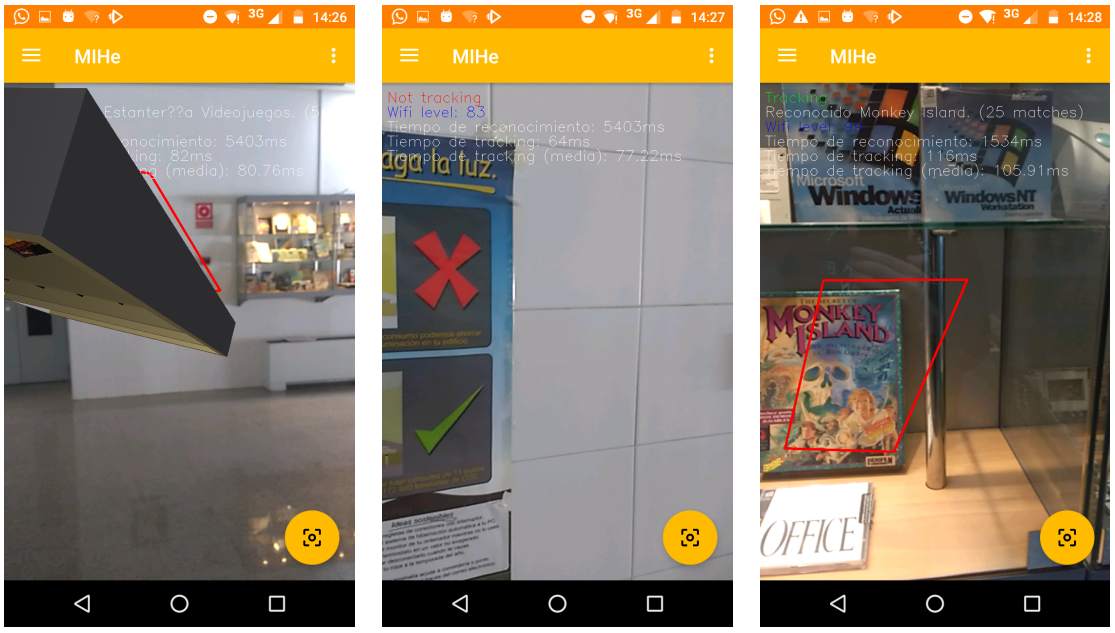

(c) Se pierde el *tracking* por realizar un movimiento brusco


(d) Se recupera la posición correcta gracias al *tracking* estático


(e) Se rota alrededor del objeto

Figura 6.6: Ejemplo de ciclo habitual de *tracking*

Además del error mencionado en la imagen 6.6c por realizar un movimiento brusco, el *tracking* también puede verse alterado debido a otras circunstancias, reflejadas en la figura 6.7.




(a) Emparejamientos de ORB erróneos

(b) Emparejamientos de ORB insuficientes (conlleva pérdida directa de tracking)

(c) Reflejos producidos sobre la vitrina

Figura 6.7: Circunstancias con *tracking* insatisfactorio

Por último, como el *tracking* se realiza a tiempo real, es necesario utilizar unos parámetros de detección y emparejamientos menos precisos que en el caso de la detección de objetos, pero que a la vez permitan seguir obteniendo buenos resultados. Por ello se realizó un estudio para decidir la configuración de ORB que mejor realizaba el *tracking* y más rápido. Para evaluar la precisión del *tracking* se han puntuado de 0 a 1 los siguientes aspectos: *tracking* realizado sin variar la distancia, variando la distancia, realizando movimientos bruscos y tratando de recuperar la imagen. Las configuraciones a considerar son las combinaciones entre el número de puntos ORB y el número de niveles de la pirámide, pues se comprobó con anterioridad que estos eran los factores que más influían tanto en la precisión como en el tiempo.

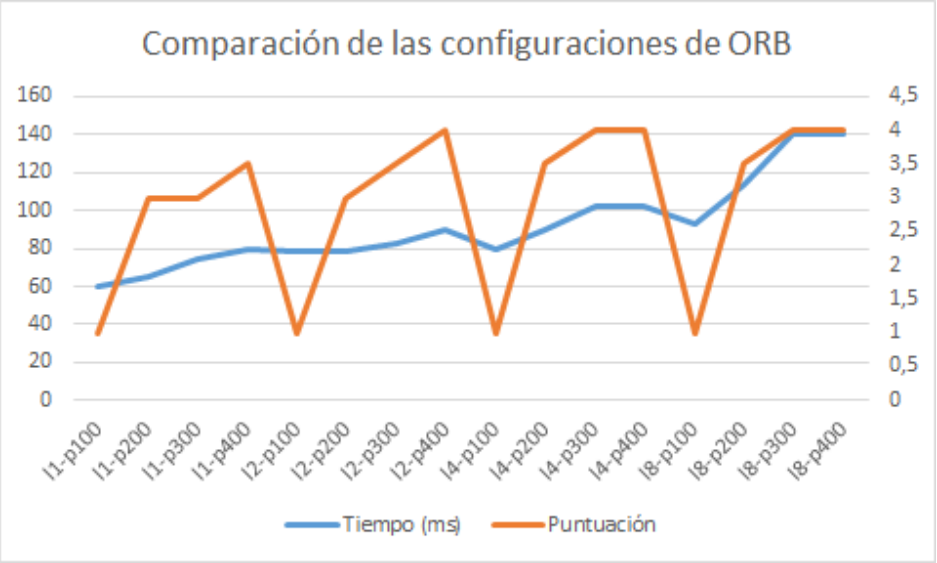



Figura 6.8: Tiempos medidos utilizando configuraciones de ORB con 100, 200, 300 y 400 puntos de interés (p) y con 1, 2, 4 y 8 niveles de pirámide (l).

A la vista de la figura 6.8, se concluyó que la mejor configuración es la de 2 niveles de pirámide y 400 puntos de interés, pues realiza buen *tracking* en un tiempo razonable. Si se diera más prioridad al rendimiento, la opción de 1 nivel de pirámide y un valor de 200 a 400 puntos de interés también podría considerarse válida.

# Capítulo 7

# Conclusiones

Además de los resultados de la aplicación mostrados en el apartado anterior, tras finalizar el proyecto se han podido extraer conclusiones a otros niveles, comentadas a continuación.

## 7.1. Tiempo empleado

El tiempo estimado para la realización del trabajo ha sido de unas 2000 horas. Esto es debido a la voluntad propia de intentar mejorar el reconocimiento de objetos (con todo el estudio, realización y pruebas que conllevaba) y elaborar un sistema completo y mantenible. A continuación se muestra el cronograma que resume el tiempo dedicado a cada uno de los aspectos más importantes durante el desarrollo del proyecto:

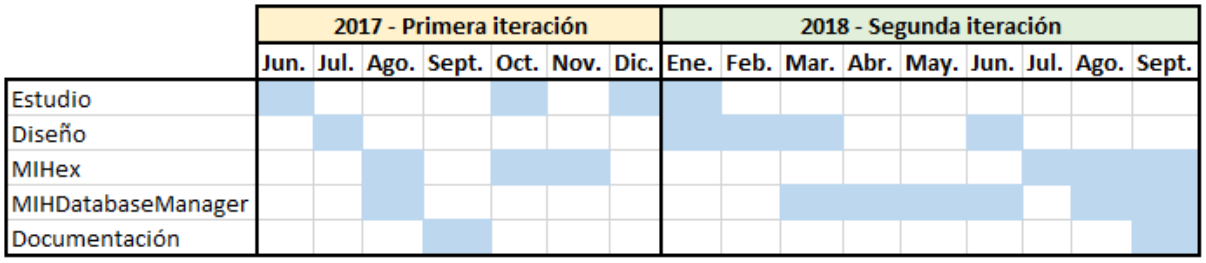



Figura 7.1: Cronograma del proyecto.

## 7.2. Posibles ampliaciones

Por falta de tiempo no se pudieron realizar todas las ideas para mejorar la aplicación. A continuación se especifican algunas de ellas, pues podrían ser de utilidad para una futura modificación de la apicación.

- Calibrar la cámara, pues los parámetros intrínsecos de la misma se han obtenido de una aproximación. Realizar una calibración podría conllevar a mejores resultados en el reconocimiento y el seguimiento.
- Añadir más texturas por modelo, pues en la versión actual solo admite una. Esto implica que los modelos 3D a utilizar deban ser bastante sencillos.
- Añadir más *shaders* para lograr otros efectos gráficos como pueden ser reflejos, transparencias, etc.

- Terminar de implementar los ítems virtuales y su selección en una estantería virtual.
- Terminar de incorporar el uso de otros descriptores para los objetos con poca información visual.
- Extraer una región de interés en la imagen de la cámara para aislar el objeto del fondo.
- Establecer un umbral de *inliers* distinto para cada objeto y ajustarlo automáticamente.
- Uso de otras técnicas de reconocimiento, como redes neuronales.

## 7.3. Opinión personal

Cuando se comenzó a plantear el proyecto se subestimaron algunos aspectos del mismo, especialmente el reconocimiento de objetos. Puesto que *a priori* parecia poca carga de trabajo, se fueron añadiendo objetivos que, finalmente, no se han podido considerar esenciales, como pueden ser la actualización por base de datos, la herramienta MIHDatabaseManager o las estanterías virtuales. Además, algunos de estos han hecho crecer paulatinamente el sistema, hasta hacerlo bastante grande y, en ocasiones, dificil de tratar. Sin embargo, a pesar de no ser esenciales, se considera que han enriquecido mucho el sistema y, a nivel personal, se ha aprendido mucho afrontando los problemas con la mentalidad de elaborar un producto que pueda ser mantenible por manos ajenas.

En cuanto al reconocimiento de objetos, a pesar de tener algunas deficiencias, a nivel personal se está bastante satisfecho con el trabajo realizado, sobre todo visto con la perspectiva del esfuerzo que se ha invertido. Tras los malos resultados de la primera iteración, todo el tiempo de búsqueda de alternativas y optimizaciones fue bastante desmoralizante, pues la mayoría de las tentativas estaban abocadas al fracaso. La principal causa de ello fue la falta de documentación de alguna de las soluciones que se pretendían abordar, dificultando así valorar si sería viable, si verdaderamente resolvería el problema o si resultaría muy complicado de configurar.

Por otro lado, la programación sobre dispositivos Android no supuso gran complicación, una vez realizados algunos tutoriales de iniciación y configuración. El mayor problema con este aspecto fue la falta de dispositivos para poder realizar pruebas y garantizar una buena compatibilidad. Además, resultó muy tedioso esperar el tiempo que Android Studio necesitaba para compilar e instalar la aplicación en el dispositivo cada vez que se realizaba una pequeña modificación y se deseaba comprobar su funcionamiento.

Por último, se está muy satisfecho con las lecciones aprendidas. La decidisión de no utilizar librerías de RA existentes y partir desde únicamente openCV y openGL para elaborar un motor de RA propio ha otorgado una buena visión sobre este paradigma, comprendiendo cuáles son sus limitaciones, cómo funciona a bajo nivel y cómo puede aplicarse de forma útil. También se han obtenido bastantes conocimientos del campo de reconocimiento de objetos, campo con muchas aplicaciones hoy en día. A su vez, cabe mencionar que, a nivel personal, la realización de este trabajo ha ayudado a fortalecer ciertos valores, como la perseverancia, el espíritu investigador y la visión de futuro.

# Bibliografía


[1] Grana, C., Borghesani, D., Manfredi, M., y Cucchiara, R, *A Fast Approach for Integrating ORB Descriptors in the Bag of Words Model*, 2015.

[2] Singh, Suriya & Choudhury, Shushman & Vishal y Kumar & V Jawahar, *Currency Recognition on Mobile Phones*, 2IIT Kharagpur, India, 2014.

[3] Azuma, y Ronald T., *A Survey of Augmented Reality*, 1997.

[4] Kim, Y. D., Park, J. T., Moon I. Y. y Oh, C. H, *Performance analysis of ORB image matching based on android*, Multimedia Content and Mobile Devices (Vol. 8667, p. 866709). International Society for Optics and Photonics. 2013.

[5] Danielsson, M. y Sievert, T, *Viability of Feature Detection on Sony Xperia Z3 using OpenCL*, 2015.

[6] Brian W. Kernighan y Dennis M. Ritchie, *C Programming Language*, segunda edición, Upper Saddle River, NJ, USA, 1988.

[7] Ramez Elmasari y Shamkant B. Navathe, *Fundamentos de Sistemas de Bases de Datos*, quinta edición, 2007

# Anexos

# Anexo A

# Glosario

En este apartado se explican aquellas palabras escogidas para referirse a los elementos del sistema.

- **Estanterías o vitrinas**. Hacen la función de expositores para los ítems del museo y pueden ser de pared o centrales.
- **Estantería virtual**. Vitrinas que no existen realmente en el museo pero podrán verse a través de la app.
- **Baldas**. Repisas que componen cada una de las estanterías.
- **Ítems u objetos lógico**. Objetos que figuran en las vitrinas.
- **Instancias u objetos físicos**. Son las ocurrencias físicas o reales de un ítem en una determinada estantería.
- **Instancias virtuales**. Son las ocurrencias virtuales de un ítem en una determinada estantería virtual.
- **Objeto del museo**. Componentes en general que conforman el museo, pudiendo referirse indistintamente a ítems o estanterías.
- **Clase de reconocimiento**. Clase abstracta que representa una entidad reconocible por el sistema de reconocimiento de objetos. Estas clases podrán ser ítems, estanterías o baldas.
- **Modelos 3D**. Conjunto de vértices y caras que componen una representación virtual de un objeto del museo.
- **Texturas**. Imágenes usadas por los modelos 3D para modificar la apariencia del modelo, principalmente el color.

# Anexo B

# Diseño de bases de datos

La información de los objetos del museo ha de almacenarse de algún modo en el dispositivo para poder ser usada por la app. Además, se decidió que dicha información podía sufrir modificaciones, por lo que es conveniente que se actualice de algún modo. Para ello se concluyó que la mejor manera para llevar esto a cabo era mediante bases de datos mySQL, una local al dispositivo y otra localizada en un servidor para poder descargar la información actualizada. Además, para introducir los datos en la BD del servidor, se decidió crear la herramienta *MIHDABAMA* (MIH Database Manager), facilitando así la tarea en la medida de lo posible. La base de esta arquitectura queda reflejada en el diagrama de la figura B.1.

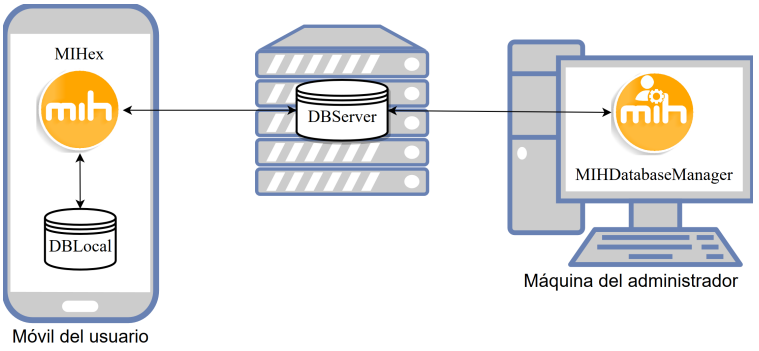



Figura B.1: Arquitectura de las bases de datos

A continuación se explica cómo se ha creado la base de datos del servidor, mientras que de la base de datos de la aplicación se comentarán únicamente las diferencias respecto a la primera. Tales diferencias serán mínimas, puesto que la información que se pretende almacenar ha de figurar en ambas BBDD.

Se ha llevado a cabo el siguiente procedimiento: diseñar un modelo Entidad-Relación, convertirlo a modelo relacional, normalizarlo y pasarlo a SQL. Posteriormente se han analizado las consultas habituales y su frecuencia para realizar un diseño físico que las optimice. Este procedimiento se llegó a realizar 3 veces hasta lograr la versión definitiva, definida a continuación. Se describirán los modelos de las versiones anteriores en el apartado B.8.

## B.1. Modelo Entidad-Relación

Debido a su crecimiento y con el fin de mostrarlo con claridad, el modelo entidad-relación definido se ha separado en varias partes[1]: elementos de la estructura del museo (ver figura B.2), elementos relativos al reconocimiento de objetos (ver B.6) y elementos actualizables (ver figura B.7). A continuación se explican en detalla cada una de estas partes.

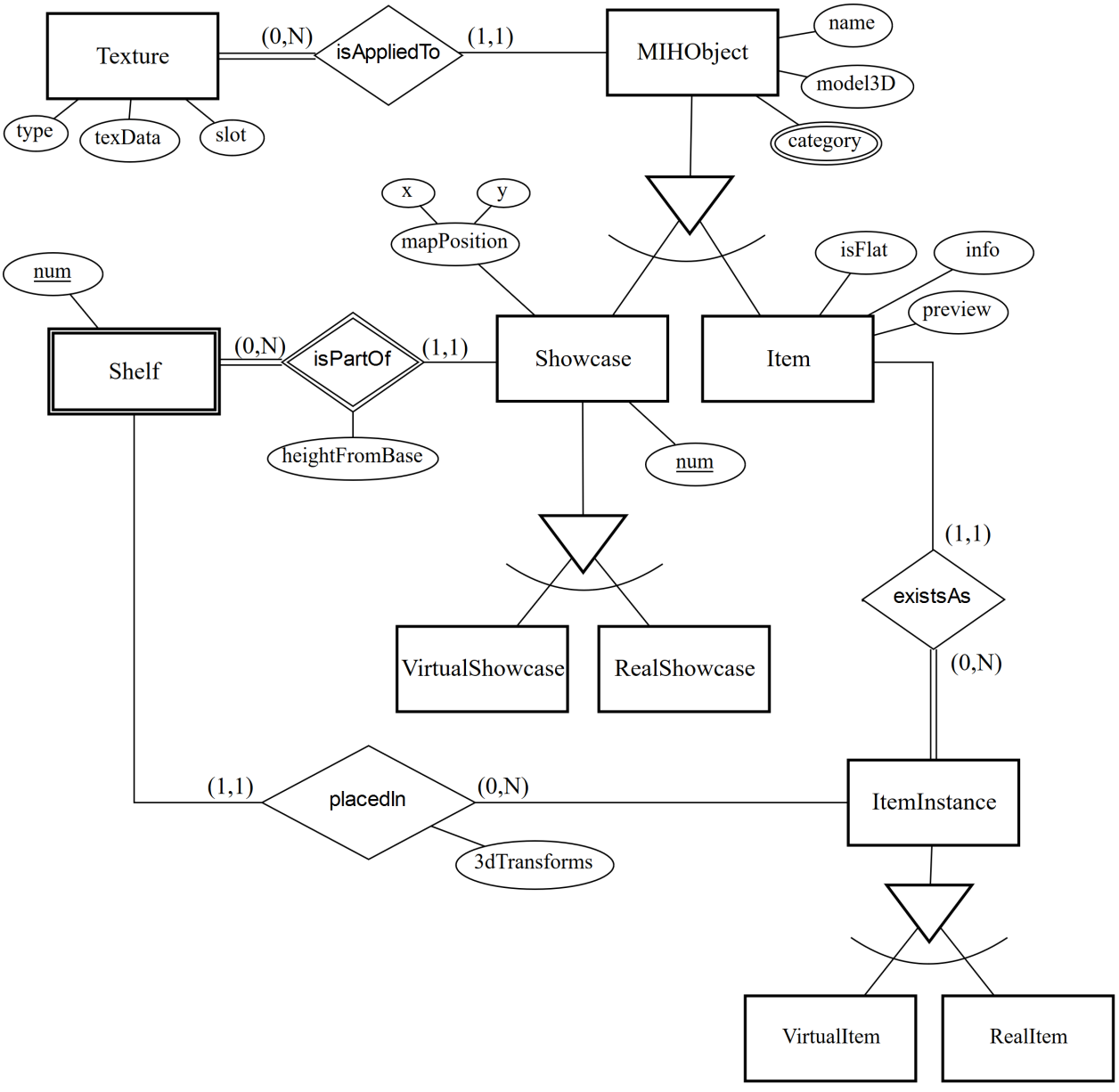



Figura B.2: Elementos que representan la arquitectura básica de los componentes del museo

A continuación se detallan las entidades y sus respectivos atributos:

- ***MIHObject***: representa todos los objetos que figuran en el museo, tanto los items como las estanerías que los contienen. Cuenta con los siguientes atributos:

[1] Puesto que la mayoría de las entidades están relacionados entre sí, algunas aparecen representadas en varias de las imágenes

- *name*: nombre del objeto
- *model3D*: modelo 3D del objeto (Ver fig. B.3)

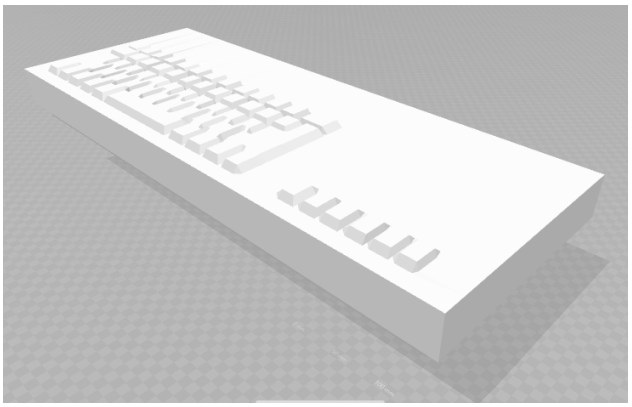
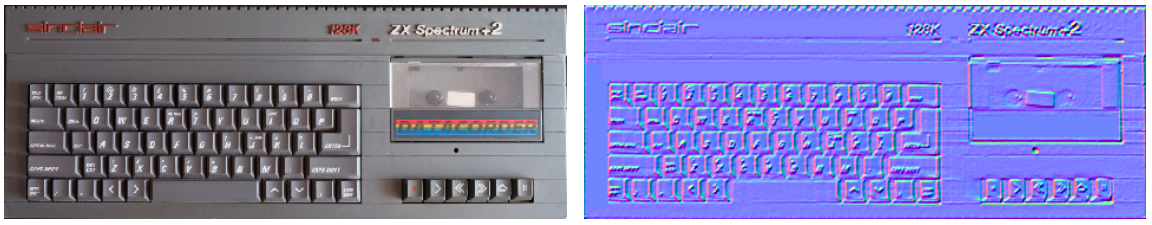


Figura B.3: Modelo del objeto en formato .obj

- ***Showcase***: representa una estantería del museo.
  - *mapPosition*: coordenadas normalizadas de la estantería en el plano del edificio
- ***VirtualShowcase***: representa una estantería virtual del museo.
- ***RealShowcase***: representa una estantería real del museo.
- ***Shelf***: representa una balda de una estantería, ya sea real o virtual.
  - *num*: número de la balda dentro de esa estantería
- ***Item***: representa un item del museo (ordenadores, chips, juegos...).
  - *info*: descripción textual del item
  - *isFlat*: indica si es un objeto aproximadamente plano (libros, chips, portadas...) o si es un objeto con volumen (PCs, servidores...)
  - *preview*: imagen del item (Ver fig. B.4)


Figura B.4: Imagen del objeto en formato .png

- ***ItemInstance***: representa la instancia de un item en el museo, ya sea un objeto físico o virtual.
- ***VirtualItem***: representa la instancia de un item virtual
- ***RealItem***: representa la instancia de un item real
- ***Texture***: representa una textura del modelo
  - *slot*: slot del modelo en el que se aplicará
  - *type*: tipo de textura (albedo=0, mapa de normales=1, mapa especular=2, etc)[2]
  - *texData*: imagen de la textura (ver Fig. B.5)



(a) Albedo (type=0) (b) Mapa de normales (type=1)

Figura B.5: Ejemplos de texturas

Del mismo modo, a continuación se especifican las relaciones representadas en el diagrama ER:

- ***isAppliedTo***: determina las texturas que se aplican sobre el modelo de un objeto
- ***isPartOf***: relaciona las baldas con la estantería a la que pertenecen
  - *heightFromBase*: altura de la balda respecto a la base inferior de la estantería
- ***placedIn***: relaciona las instancias de los items con las baldas en las que se encuentran. Además, en caso de ser un objeto virtual situado en una estantería virtual, define las transformaciones que se han de aplicar sobre el modelo para situarlo en la posición deseada:
  - *3dTransforms*: traslación, rotación y escala que se deberá aplicar al visualizarlo respecto al origen de la balda, es decir, el origen de la estantería más la altura de la balda.

[2] Más información en el manual de usuario de MIHDABAMA.

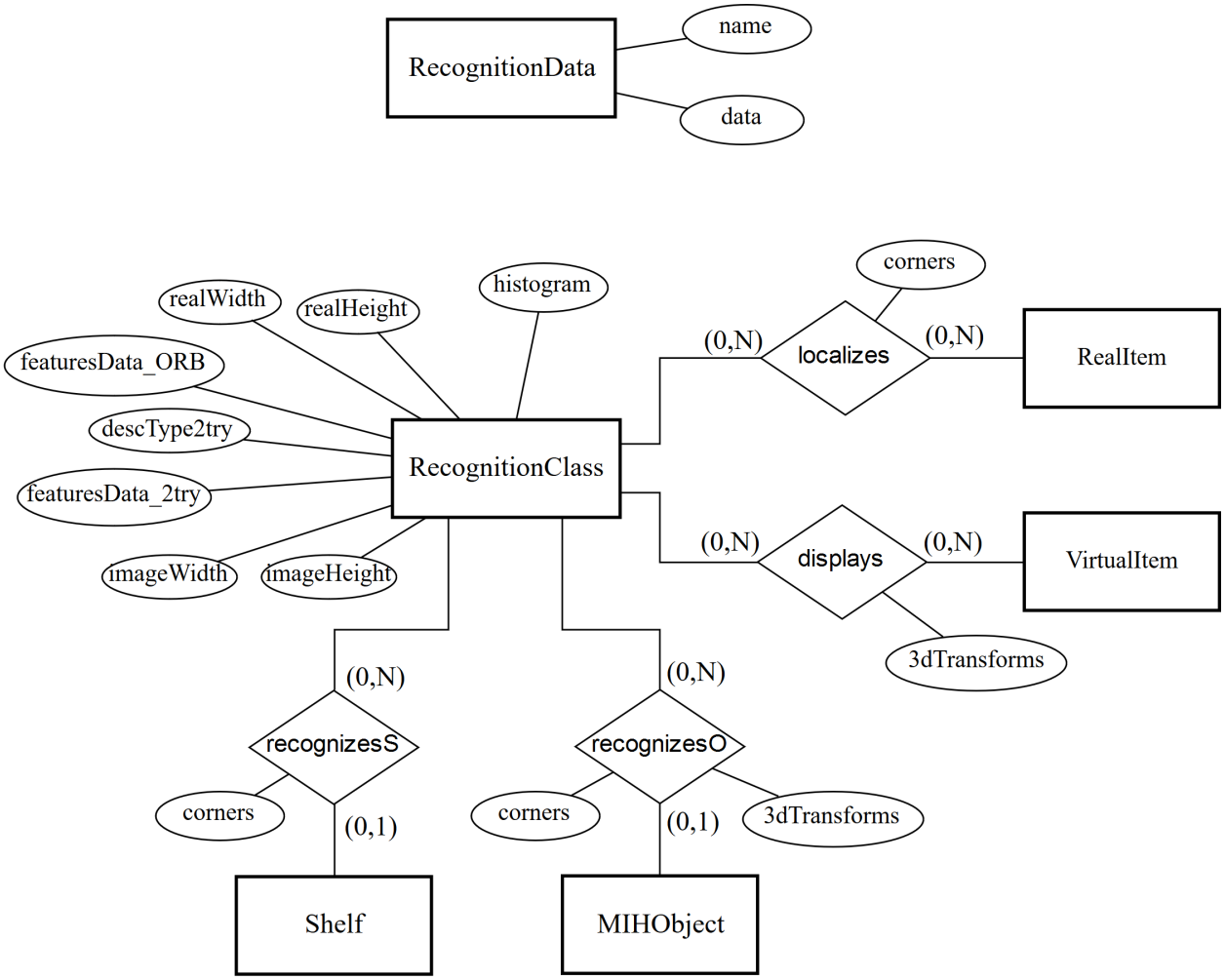



Figura B.6: Elementos que representan la arquitectura necesaria para almacenar los datos de reconocimiento asociados a los componentes del museo

A continuación se detallan las nuevas entidades y sus respectivos atributos:

- ***RecognitionClass***: representa una clase reconocible por el algoritmo de reconocimiento. Esta clase puede representar un item del museo, una estantería o una balda con instancias de items. Cuenta con los siguientes atributos:
  - *imageWidth*: ancho de la imagen utilizada en la extracción de puntos de interés en píxeles
  - *imageHeight*: alto de la imagen utilizada en la extracción de puntos de interés en píxeles
  - *realWidth*: ancho real del objeto al que representa la clase de reconocimiento (en cms)
  - *realHeight*: alto real del objeto al que representa la clase de reconocimiento (en cms)
  - *featuresData_ORB*: datos ORB extraidos de una imagen, necesarios para la fase de validación en el reconocimiento y su posterior localicación

  - *featuresData_2try*: datos extraidos de la imagen alternativos utilizando otro tipo de punto de interés (BRIEF, AKAZE). Estos datos se utilizan cuando se requiere una validación más exhaustiva durante el reconocimiento. La validación resulta más lenta que con ORB, por lo que sólo han de ser introducidos cuando se considere extricamente necesarios.
  - *descType_2try*: tipo de los puntos de interés mencionados en el punto anterior. Los valores contemplados son:
    - Ninguno=0
    - AKAZE=1
    - BRIEF=2
  - *histogram*: representa el histograma utilizado para reconocer una clase de reconocimiento

- ***RecognitionData***: almacena otros datos utilizados durante el reconocimiento (palabras, arbol de palabras, clusters...). Cuenta con los siguientes atributos:
  - *name*: nombre de los datos, para poder obtenerse de forma intuitiva. P.e.: *words*, *clusters...*
  - *data*: datos almacenados como bloque de bytes

Del mismo modo, a continuación se especifican las relaciones representadas en el diagrama ER:

- ***localizes***: determina las instancias reales que aparecen en la imagen de la clase (sólo cuando representa una balda). En caso de que varias clases de reconocimiento representen la misma balda [3], una misma instancia también podrá ser localizada en las distintas clases de reconocimiento de esa balda.
  - *corners*: coordenadas de los puntos que enmarcan la instancia en la imagen
- ***displays***: determina las instancias virtuales que se desean mostrar sobre la imagen de la clase (sólo cuando representa una balda). En caso de que varias clases de reconocimiento representen la misma balda [4], una misma instancia también podrá ser mostrada en las distintas clases de reconocimiento de esa balda.
  - *3dTransforms*: traslación, rotación y escala que se deberá aplicar al visualizarlo respecto a las coordenadas (0,0) de la imagen de la clase y su tamaño. La tercera dimensión se considera perpendicular al plano de la imagen (cuyo valor en esta dimensión es 0).
- ***recognizesS***: determina la balda que reconoce la clase de reconocimiento. En caso de reconocer un item o estantería no participará en esta relación.

[3] Esto sucede cuando una balda quiere reconocerse de distintos ángulos, como las de las estanterías centrales. Para permitir esto, se optó por utilizar varias clases de reconocimiento que apunten a la misma balda.

[4] Esto sucede cuando una balda quiere reconocerse de distintos ángulos, como las de las estanterías centrales. Para permitir esto, se optó por utilizar varias clases de reconocimiento que apunten a la misma balda.

  - *corners*: coordenadas de los puntos que enmarcan la balda en la imagen

- ***recognizesO***: determina la balda que reconoce la clase de reconocimiento. En caso de reconocer un item o estantería no participará en esta relación.

  - *corners*: coordenadas de los puntos que enmarcan el objeto en la imagen
  - *3dTransforms*: traslación, rotación y escala que se deberá aplicar al visualizarlo respecto al origen de la balda, es decir, el origen de la estantería más la altura de la balda. Puede no estar definido en caso de que solo se desee mostrar la etiqueta con el nombre.

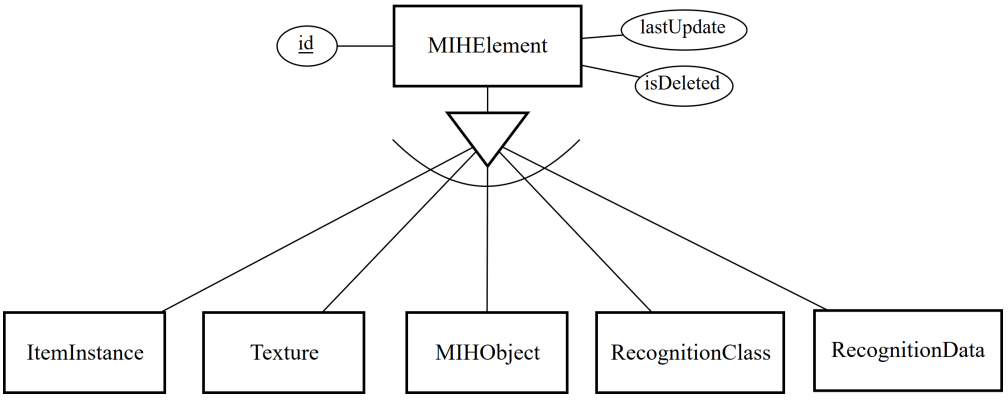



Figura B.7: Elementos actualizables

Con el objeto de favorecer las actualizaciónes, se establecieron como hijos de la entidad *MIHElement* los elementos que se desean actualizar sin afectar al resto:

- ***MIHElement***: representa cualquier elemento del que se desea comprobar su estado y descargarlo a la app.

  - *id*: identificador del elemento
  - *lastUpdate*: última modificación de inserción, borrado o actualizado realizada sobre el elemento.
  - *isDeleted*: se utiliza para marcar cuando un elemento se ha borrado. Si el elemento es borrado completamente la app necesitaría comprobar toda su base de datos contra la del servidor para saber que elementos faltan. Por ello, la solución adoptada ha sido eliminar el elemento hijo pero mantener el *MIHElement* con el atributo *isDeleted* marcado. De esta forma, la app sólo tendrá que comprobar este valor para borrarlo en su base de datos.

Por último, algunos de los atributos mencionados en los puntos anteriores son atributos compuestos, que por claridad sólo se ha mostrado el atributo raiz. En la figura B.8 se muestran todos ellos desglosados:

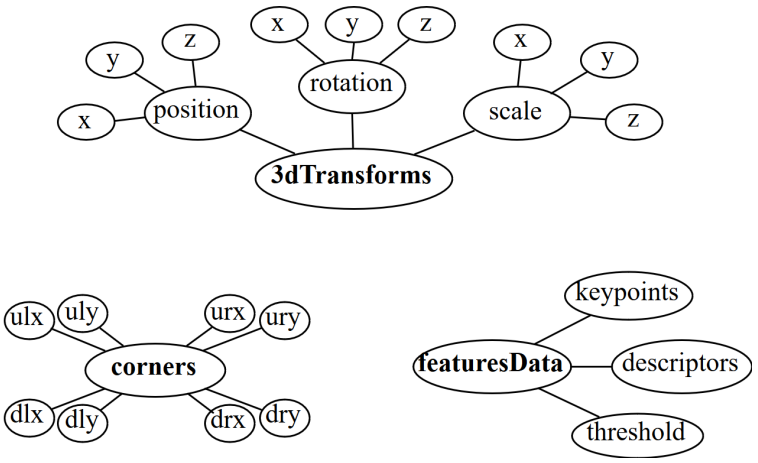



Figura B.8: Atributos compuestos

## B.2. Modelo relacional y normalización

A partir del modelo ER se pudo diseñar el primer modelo relacional. Para ello se tomaron las siguientes decisiones:

- Definir una relación para la superclase *MIHElement* para contener los atributos de actualización comunes entre las especializaciones. Se añadió también el atributo *type* para identificar el tipo de elemento desde esta relación. Este valor se estableció en orden creciente atendiendo a las depencias que originarán entre ellos: *Item*=0, *Showcase*=1, *Texture*=2, *RecognitionClass*=3, *ItemInstance*=4, y *RecognitionData*=5.

- Definir una relación para cada especialización de *MIHObject*, pues a parte de tener suficiente información como para ser representadas de forma separada, cada una de ellas participa en relaciones muy específicas. Además se decidió mantener la superclase *MIHObject* para contener los atributos comunes y poder representar correctamente las relaciones en las que participa. Se le añadió el atributo *isItem* para identificar el tipo de objeto desde la superclase.

- Para representar el atributo multivaluado *category* de *MIHObject* se ha creado la relación *ObjectCategory*, que contiene por clave la clave extranjera de *MIHObject* y la categoría.

- Agrupar las especializaciones de de *Showcase* en la superclase, pues no tienen suficiente información como para ser representadas de forma separada ni participan expresamente en ninguna relación. Para mantener la identidad de a cual entidad pertenecería se ha añadido el atributo *isVirtual*.

- Definir una relación para la superclase *ItemInstance* y eliminar las especializaciones, pues no contienen información específica. Las participaciónes en las relaciones *localizes* y *displays* pasarán a la superclase *ItemInstance*.

- La relación *existsAs*, por ser 1:N, se traduce al modelo relacional propagando la clave de *Item* a *ItemInstance* como clave extranjera.

- Por ser *Texture* una entidad débil, se establece como clave, junto con *slot* y *type*, la clave extranjera *name* de *Model3D*
- Por ser *Shelf* una entidad débil, se establece como clave, junto con *num*, la clave extranjera *name* de *Showcase*
- Las relaciones *recognizesS* y *recognizesO*, al ser ambas 1:N, se han representado en el modelo relacional propagando las claves de *MIHObject* y *Shelf* respectivamente a *RecognitionClass*, así como los atributos de las relaciones.
- Para representar las relaciones *localizes* y *displays*, al ser ambas N:M, se han creado las nuevas relaciones *localizes* y *displays*, estableciendo como claves las correspondiente claves extranjeras de las entidades partícipes.

De esta forma, el modelo relacional obtenido puede apreciarse en la figura B.9.

En este punto resultó conveniente realizar una normalización para garantizar un mejor diseño de la BD, reduciendo las redundancias o anomalías que el actual modelo pudiera ocasionar. Se realizó la normalización hasta la Forma Normal de Boyce-Codd, pues es considerada una forma normal estable y generalmente suficiente.

- 1FN: el único atributo multivaluado, *Category* de *MIHObject*, se contempló a la hora de crear el modelo relacional, por lo que este ya está en 1FN. Los atributos de keypoints y descriptores de *RecognitionClass* se han considerado como simples bloques de bytes, por lo que tampoco pueden considerarse atributos multivaluados.
- 2FN: ningún atributo depende funcionalmente de parte de la clave, por lo que está en 2FN.
- 3FN: no hay dependencias funcionales transitivas, por lo que ya está en 3FN. Se ha considerado que los atributos de bloque de bytes, como *model3D* o *keypointsORB*, no pueden ser determinantes de dependencias funcionales.
- FNBC: todo determinante de una dependencia funcional es clave o clave alternativa, por lo que ya está en Forma Normal de Boyce-Codd.

Al contrario que en las iteraciones anteriores, el modelo relacional obtenido directamente desde el modelo Entidad-Relación no ha sufrido ninguna modificación durante la normalización, por lo que se puede llevar directamente al modelo SQL.

## B.3. Diseño SQL

Una vez realizada la normalización se puede comenzar su diseño en SQL. Para ello es necesario definir los tipos de datos de los atributos y establecer las relaciones pertinentes. De este modo la implementación de la base de datos resulta la siguiente:

```
CREATE TABLE MIHElement (
  id         MEDIUMINT NOT NULL AUTO_INCREMENT,
  type       TINYINT NOT NULL,
  lastUpdate DATETIME NOT NULL,
```

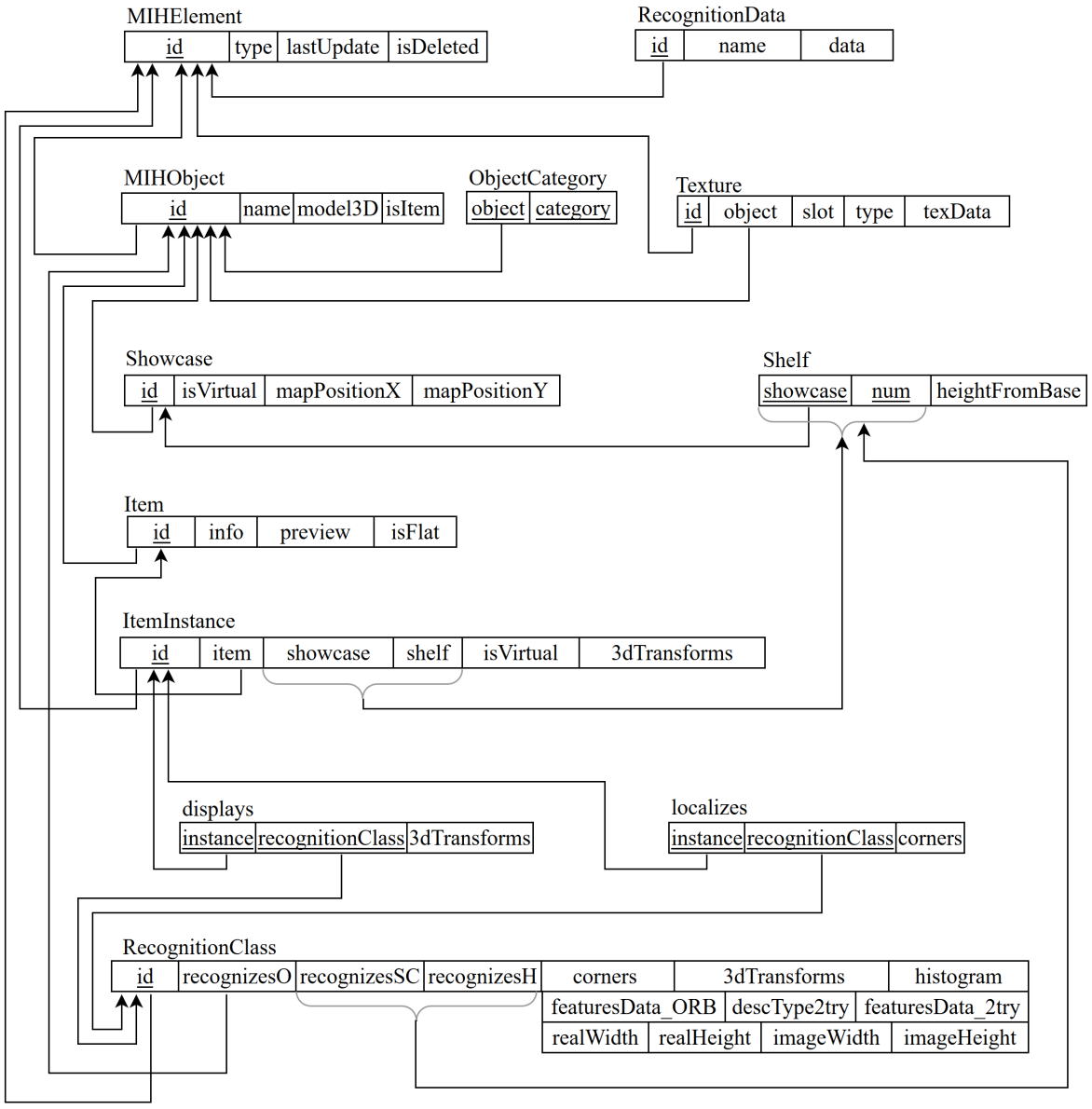



Figura B.9: Primera versión del modelo relacional

```
  isDeleted  BOOLEAN NOT NULL,
  PRIMARY KEY(id));

CREATE TABLE RecognitionData (
  id       MEDIUMINT NOT NULL,
  name     VARCHAR(100) NOT NULL,
  data     MEDIUMBLOB NOT NULL,
  PRIMARY KEY(id),
  FOREIGN KEY(id)
     REFERENCES MIHElement(id)
     ON DELETE CASCADE
     ON UPDATE CASCADE);
```

```
CREATE TABLE MIHObject (
  id      MEDIUMINT NOT NULL,
  name    VARCHAR(100) UNIQUE,
  model3d MEDIUMBLOB,
  isItem  BOOLEAN NOT NULL,
  PRIMARY KEY(id),
  FOREIGN KEY(id)
     REFERENCES MIHElement(id)
     ON DELETE CASCADE
     ON UPDATE CASCADE);

CREATE TABLE ObjectCategory (
  object     MEDIUMINT NOT NULL,
  category   VARCHAR(100),
  PRIMARY KEY(object, category),
  FOREIGN KEY(object)
     REFERENCES MIHElement(id)
     ON DELETE CASCADE
     ON UPDATE CASCADE);

CREATE TABLE Texture (
  id      MEDIUMINT NOT NULL,
  object  MEDIUMINT NOT NULL,
  slot    INTEGER,
  type    INTEGER,
  texData MEDIUMBLOB NOT NULL,
  PRIMARY KEY(id),
  FOREIGN KEY(id)
     REFERENCES MIHElement(id)
     ON DELETE CASCADE
     ON UPDATE CASCADE,
  FOREIGN KEY(object)
     REFERENCES MIHObject(id)
     ON DELETE CASCADE
     ON UPDATE CASCADE);

CREATE TABLE Showcase (
  id            MEDIUMINT NOT NULL,
  isVirtual     BOOLEAN NOT NULL,
  mapPositionX  DOUBLE,
  mapPositionY  DOUBLE,
  PRIMARY KEY(id),
  FOREIGN KEY(id)
     REFERENCES MIHElement(id)
     ON DELETE CASCADE
     ON UPDATE CASCADE);

CREATE TABLE Shelf (
```

```
  showcase   MEDIUMINT NOT NULL,
  num        TINYINT,
  heightFromBase DOUBLE,
  PRIMARY KEY(showcase, num),
  FOREIGN KEY(showcase)
     REFERENCES Showcase(id)
     ON DELETE CASCADE
     ON UPDATE CASCADE);

CREATE TABLE RecognitionClass (
  id      MEDIUMINT NOT NULL,
  recognizesO        MEDIUMINT,
  recognizesShowcase MEDIUMINT,
  recognizesShelf    TINYINT,
  cornerULX     DOUBLE,
  cornerULY     DOUBLE,
  cornerURX     DOUBLE,
  cornerURY     DOUBLE,
  cornerDLX     DOUBLE,
  cornerDLY     DOUBLE,
  cornerDRX     DOUBLE,
  cornerDRY     DOUBLE,
  positionX     DOUBLE,
  positionY     DOUBLE,
  positionZ     DOUBLE,
  rotationX     DOUBLE,
  rotationY     DOUBLE,
  rotationZ     DOUBLE,
  scaleX        DOUBLE,
  scaleY        DOUBLE,
  scaleZ        DOUBLE,
  keypointsORB    MEDIUMBLOB NOT NULL,
  descriptorsORB  MEDIUMBLOB NOT NULL,
  thresholdORB  INTEGER,
  descType2try  TINYINT,
  keypoints2try   MEDIUMBLOB,
  descriptors2try    MEDIUMBLOB,
  threshold2try INTEGER,
  imageWidth    INTEGER NOT NULL,
  imageHeight   INTEGER NOT NULL,
  realWidth     INTEGER NOT NULL,
  realHeight    INTEGER NOT NULL,
  histogram MEDIUMBLOB NOT NULL,
  PRIMARY KEY(id),
  FOREIGN KEY(id)
     REFERENCES MIHElement(id)
     ON DELETE CASCADE
     ON UPDATE CASCADE,
  FOREIGN KEY(recognizesO)
```

```
    REFERENCES MIHObject(id)
    ON DELETE CASCADE
    ON UPDATE CASCADE,
  FOREIGN KEY(recognizesShowcase, recognizesShelf)
    REFERENCES Shelf(showcase, num)
    ON DELETE CASCADE
    ON UPDATE CASCADE);


CREATE TABLE Item (
  id      MEDIUMINT NOT NULL,
  info    VARCHAR(3000),
  preview    BLOB,
  isFlat     BOOLEAN NOT NULL,
  PRIMARY KEY(id),
  FOREIGN KEY(id)
    REFERENCES MIHObject(id)
    ON DELETE CASCADE
    ON UPDATE CASCADE);

CREATE TABLE ItemInstance (
  id         MEDIUMINT NOT NULL,
  item       MEDIUMINT NOT NULL,
  showcase   MEDIUMINT,
  shelf      TINYINT,
  isVirtual  BOOLEAN NOT NULL,
  positionX  DOUBLE,
  positionY  DOUBLE,
  positionZ  DOUBLE,
  rotationX  DOUBLE,
  rotationY  DOUBLE,
  rotationZ  DOUBLE,
  scaleX     DOUBLE,
  scaleY     DOUBLE,
  scaleZ     DOUBLE,
  PRIMARY KEY(id),
  FOREIGN KEY(id)
    REFERENCES MIHElement(id)
    ON DELETE CASCADE
    ON UPDATE CASCADE,
  FOREIGN KEY(item)
    REFERENCES Item(id)
    ON DELETE CASCADE
    ON UPDATE CASCADE,
  FOREIGN KEY(showcase, shelf)
    REFERENCES Shelf(showcase, num)
    ON DELETE SET NULL
    ON UPDATE CASCADE);
```

```
CREATE TABLE localizes (
  instance        MEDIUMINT NOT NULL,
  recognitionClass MEDIUMINT NOT NULL,
  cornerULX     DOUBLE,
  cornerULY     DOUBLE,
  cornerURX     DOUBLE,
  cornerURY     DOUBLE,
  cornerDLX     DOUBLE,
  cornerDLY     DOUBLE,
  cornerDRX     DOUBLE,
  cornerDRY     DOUBLE,
  PRIMARY KEY(instance, recognitionClass),
  FOREIGN KEY(instance)
     REFERENCES ItemInstance(id)
     ON DELETE CASCADE
     ON UPDATE CASCADE,
  FOREIGN KEY(recognitionClass)
     REFERENCES RecognitionClass(id)
     ON DELETE CASCADE
     ON UPDATE CASCADE);

CREATE TABLE displays (
  instance        MEDIUMINT NOT NULL,
  recognitionClass MEDIUMINT NOT NULL,
  positionX  DOUBLE,
  positionY  DOUBLE,
  positionZ  DOUBLE,
  rotationX  DOUBLE,
  rotationY  DOUBLE,
  rotationZ  DOUBLE,
  scaleX     DOUBLE,
  scaleY     DOUBLE,
  scaleZ     DOUBLE,
  PRIMARY KEY(instance, recognitionClass),
  FOREIGN KEY(instance)
     REFERENCES ItemInstance(id)
     ON DELETE CASCADE
     ON UPDATE CASCADE,
  FOREIGN KEY(recognitionClass)
     REFERENCES RecognitionClass(id)
     ON DELETE CASCADE
     ON UPDATE CASCADE);
```

## B.4. Consultas y transacciones habituales

Las consultas y transacciones habituales que se realizarán sobre esta base de datos se pueden agrupar en dos categorías:

- Consultas o transacciones realizadas desde la aplicación MIHex. Son las realizadas por los usuarios de la aplicación, por lo que su finalidad será únicamente obtener información de la base de datos (nunca modificarla o borrarla). Las consultas de esta categoría pertenecen a una macro-transacción como parte del algoritmo de actualización, tal y como se muestra en el diagrama de secuencia de la figura B.10.

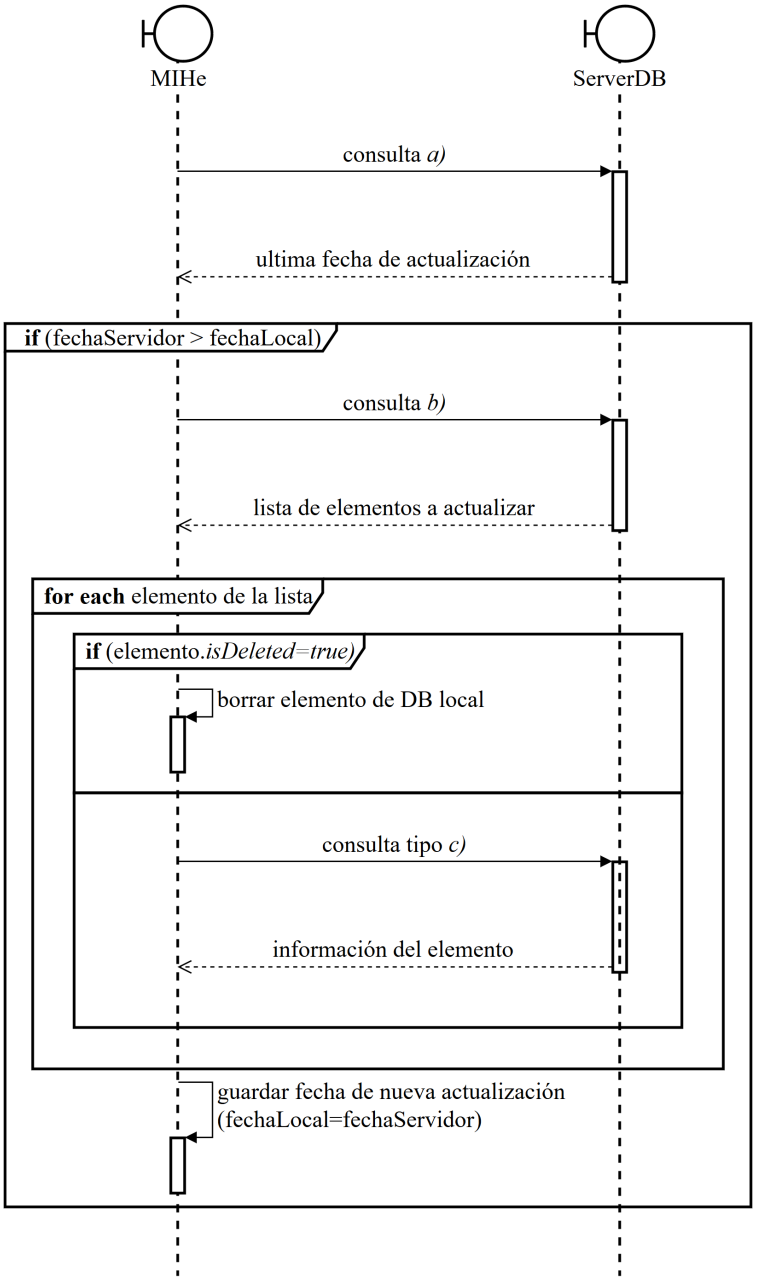



Figura B.10: Diagrama de secuencia de la macro-transacción realizada desde la app MIHex

- Consultas o transacciones realizadas desde la herramienta MIHDABAMA. Son realizadas por el administrador de la base de datos para añadir, modificar o eliminar la información de la misma. Al ser realizadas únicamente por un usuario se realizarán con menor frecuencia que las del tipo anterior.

A continuación se especifican algunas de las consultas y transacciones frecuentes que resultan de interés, bien sean realizadas por los usuarios (desde MIHex) o realizadas por el administrador (desde *MIHDABAMA*):

a) Obtener la última fecha de modificación de la BD (realizada desde MIHex):

```
SELECT max(lastUpdate) FROM MIHElement;
```

A continuación se muestra un breve análisis de la consulta:

| | |
|---|---|
| Tipo | SELECT |
| Tablas (cardinalidad) | *MIHElement* (7000) |
| Cond. selección (selectividad) | max(*lastUpdate*) (0.00014 %) |
| Cond. join (selectividad) | — |
| Atributos | *lastUpdate* |
| Frecuencia | 80 veces al día |
| Restricciones | Debe realizarse lo más rápido posible<br>Debe transmitir el mínimo de datos posible |

Tabla B.1: Tabla de análisis de la consulta *a)*

La consulta se realiza cada vez que alguien inicia la aplicación, por lo que es la más frecuente. Ello implica que se deba optimizar lo máximo posible, tanto en tiempo de ejecución como en datos transmitidos a la app, pues es posible que el usuario no esté conectado a la red Wifi y se desea que el impacto en su consumo de datos sea mínimo. Por todo esto se considera que esta es una consulta con prioridad crítica.

b) Obtener la lista de todos los elementos añadidos, modificados o borrados desde una versión anterior (realizada desde MIHex), ordenados por tipo para evitar problemas de dependencias:

```
SELECT id, type, isDeleted
FROM MIHElement
WHERE lastUpdate > <fecha de ultima actualizacion del cliente>
ORDER BY type;
```

Una vez obtenida la lista de los elementos a actualizar se realizarán consultas específicas para cada tipo de elemento de forma secuencial en orden de tipo, comenzando por *MIHObject* para facilitar las dependencias del resto de elementos. A continuación se muestra un breve análisis de la consulta:

| | |
|---|---|
| Tipo | SELECT |
| Tablas(cardinalidad) | *MIHElement*(7000) |
| Cond. selección | *lastUpdate*>fecha-ultima-actualización (20 %) |
| Cond. join | — |
| Atributos | *id, type, lastUpdate, isDeleted* |
| Frecuencia | 20 veces al mes |
| Restricciones | Debe realizarse lo más rápido posible |

Tabla B.2: Tabla de análisis de la consulta *b)*

Para esta consulta el usuario ya debería haberse conectado al wifi tras recibir una advertencia del uso de datos desde la app, por lo que la mínima transferencia de datos ya no es tan crítica, aunque sigue siendo prioritaria por motivos de velocidad. Esta consulta solo se realiza cuando a través de la consulta a la aplicación detecta que hay actualizaciones disponibles, por lo que no se realiza cada vez que el usuario abre la app.

c) Obtener toda la información de una estantería (realizada desde MIHex y MIHDABAMA):

```
SELECT *
FROM MIHObject
WHERE id=<id>;
SELECT *
FROM ObjectCategory
WHERE id=<id>;

SELECT *
FROM Showcase
WHERE id=<id>;
SELECT *
FROM Shelf
WHERE showcase=<id>;
```

Las consultas para obtener cualquier otro tipo de elemento resultan muy similares a esta. A continuación se muestra un breve análisis de la consulta:

| | |
|---|---|
| Tipo | SELECT |
| Tablas (cardinalidad) | *MIHObject* (2000), *ObjectCategory* (2500), *Showcase* (15), *Shelf* (90) |
| Cond. selección (selectividad) | *id*=idBuscado (0.05 %, 0.04 %, 6.6 % y 1.1 %) |
| Cond. join (selectividad) | — |
| Atributos | todos los de las entidades buscadas |
| Frecuencia | 60 veces al mes |
| Restricciones | Debe realizarse lo más rápido posible |

Tabla B.3: Tabla de análisis de la consulta *c)*

Los datos de la tabla B.3 se han calculado para la consulta exacta anterior. Si tenemos en cuenta todas las consultas similares que tienen por objetivo obtener los datos de las especializaciones de *MIHElement* y derivados, estas tienen en su conjunto una selectividad aproximada de un 20 % de las tuplas de *MIHElement* (tal y como se ha visto en la consulta b). Por tanto, el conjunto de este tipo de consultas suponen una frecuencia de 28000 al mes y afectarían a todas las tablas.

d) Añadir un item (desde *MIHDABAMA*):

```
INSERT INTO MIHElement (type, lastUpdate, isDeleted) VALUES (0, NOW(),
    false);
INSERT INTO MIHObject (id, name, model3D, isItem)
  VALUES (LAST_INSERT_ID(), <nombre>, <modelo 3D>, true);
INSERT INTO Item (id, info, preview, isFlat)
  VALUES (LAST_INSERT_ID(), <descripcion>, <preview>, <es plano?>);
INSERT INTO ObjectCategory (object, category) VALUES (LAST_INSERT_ID(),
    <categoria 1>);
...
INSERT INTO ObjectCategory (object, category) VALUES (LAST_INSERT_ID(),
    <categoria n>);
```

Las consultas para añadir una estantería resultan muy similares a esta. A continuación se muestra un breve análisis de la consulta:

| Tipo | INSERT |
|---|---|
| Tablas (cardinalidad) | *MIHElement* (7000), *MIHObject* (2000), *Item* (2000), *ObjectCategory* (2500) |
| Cond. selección (selectividad) | — |
| Cond. join (selectividad) | — |
| Atributos | *Todos los de las entidades implicadas* |
| Frecuencia | 200 veces al mes |
| Restricciones | — |

Tabla B.4: Tabla de análisis de la consulta *d)*

En esta ocasión la consulta no se realiza desde la app, por lo que ya no es tan crítica la restricción de tiempo.

e) Modificar un item (desde *MIHDABAMA*):

```
UPDATE MIHElement SET lastUpdate=NOW() WHERE id=<id>;
UPDATE MIHObject SET name=<nuevo nombre>, model3D=<nuevoModelo> WHERE
    id=<id>;
UPDATE Item info=<nueva descripcion>, preview=<nuevo preview>, isFlat=<es
    plano?> WHERE id=<id>;
```

Las consultas para modificar una estantería resultan muy similares a esta. A continuación se muestra un breve análisis de la consulta:

| Tipo | UPDATE |
|---|---|
| Tablas (cardinalidad) | *MIHElement* (7000), *MIHObject* (2000), *Item* (2000) |
| Cond. selección (selectividad) | *id*=idBuscado (0.014 %, 0.05 % y 0.05 %) |
| Cond. join (selectividad) | — |
| Atributos | *lastUpdate, name, model3D, info, preview, isFlat* |
| Frecuencia | 100 veces al mes |
| Restricciones | — |

Tabla B.5: Tabla de análisis de la consulta *e)*

f) Borrar un objeto (desde *MIHDABAMA*):

```
DELETE FROM MIHObject WHERE id=<id>;
UPDATE MIHElement SET isDeleted=true, lastUpdate=<fecha actual> WHERE
    id=<id>;
```

Las tuplas de las especializaciones *Item* y *Showcase* son eliminadas automáticamente gracias al *DELETE CASCADE* de sus claves extranjeras. Lo mismo ocurre con las tablas *ObjectCategory*, *RealItem*, *VirtualItem* y *Shelf*. A continuación se muestra un breve análisis de la consulta:

| Tipo | UPDATE, DELETE |
|---|---|
| Tablas (cardinalidad) | *MIHElement* (7000), *MIHObject* (2000) |
| Cond. selección (selectividad) | *id*=idBuscado (0.014 % y 0.05 %) |
| Cond. join (selectividad) | — |
| Atributos | *lastUpdate, isDeleted* |
| Frecuencia | 20 veces al mes |
| Restricciones | — |

Tabla B.6: Tabla de análisis de la consulta *f)*

g) Obtener todas las categorías introducidas hasta la fecha (desde *MIHDABAMA*):

```
SELECT DISTINCT Category FROM ObjectCategory;
```

A continuación se muestra un breve análisis de la consulta:

| Tipo | SELECT |
|---|---|
| Tablas (cardinalidad) | *ObjectCategory* (2500) |
| Cond. selección (selectividad) | DISTINCT *category* (2 %) |
| Cond. join (selectividad) | — |
| Atributos | *category* |
| Frecuencia | 4 veces al mes |
| Restricciones | — |

Tabla B.7: Tabla de análisis de la consulta *g)*

h) Actualizar el vocabulario y los histogramas (desde *MIHDABAMA*):

```
UPDATE MIHElement SET lastUpdate=NOW() WHERE id=<id-data>;
UPDATE RecognitionData SET data=<descriptores-de-palabras> WHERE
    name='words';

-- Para todas las clases de reconocimiento
UPDATE MIHElement SET lastUpdate=NOW() WHERE id=<id-clase>;
UPDATE RecognitionClass SET histogram=<nuevo histograma> WHERE id=<id-clase>;
```

A continuación se muestra un breve análisis de la consulta:

| | |
|---|---|
| Tipo | UPDATE |
| Tablas (cardinalidad) | *RecognitionData* (5), *RecognitionClass* (2500) |
| Cond. selección (selectividad) | *name*='words'(20 %), *id*=idClase(0.04 %) |
| Cond. join (selectividad) | — |
| Atributos | *data*, *histogram* |
| Frecuencia | 4 veces al mes |
| Restricciones | — |

Tabla B.8: Tabla de análisis de la consulta *h)*

Aunque la condición de selectividad con *RecognitionClass* tenga valor bajo, esa selección ha de realizarse para todos los elementos de la tabla, por lo que realmente afectará al 100 % de los elementos de la tabla, suponiendo un impacto bastante significativo.

Se desea que todas las transacciones anteriores cumplan el principio *ACID*[5], así como evitar los problemas de concurrencia de las mismas. Se ha diseñado esta BD con el objeto de ser modificada por un único administrador, por lo que no debería haber problemas de escritura por concurrencia en esta base de datos. Por otro lado, se desea que la macro-transacción mostrada en la figura B.10 garantice que el estado de la BD local de MIHex sea consistente con el del servidor. Al haber únicamente un administrador con estimación de frecuencia de modificaciones baja se ha considerado que el problema se podrá abordar desde el lado de MIHex con técnicas optimístas de serializabilidad o mediante lectura consistente.

## B.5. Diseño físico

A la vista de las consultas habituales y sus características resulta conveniente realizar un diseño físico que optimice estas transacciones, sobre todo las realizadas desde MIHex. También resulta deseable que las transacciones que requieran de varios accesos a la BD se automaticen lo máximo para ahorrar trabajo al administrador. Esto se puede solventar mediante el uso de *triggers*, los cuales también serán capaces de evitar las inserciones, modificaciones y borrados por parte del administrador en la tabla *MIHElement*, haciendola transparente en estos usos.
Para ello se han decidido tomar las siguientes medidas:

- Definir el atributo *lastUpdate* de *MIHElement* como índice de la tabla. De esta forma se obtendrá una gran optimización en las consultas a y b, pues al ordenarse las tuplas por este valor resulta muy sencillo obtener cual es su máximo y cuáles son las tuplas más recientes a una determinada fecha.

[5]Atomicity, Consistency, Isolation and Durability

- Para realizar la consulta c se llevan a cabo 4 transacciones. Entre ellas, la que accede a la tabla de *MIHObject* se puede eliminar moviendo los atributos a sus especializaciones y a su generalización, pues siempre se van a recuperar juntos tales atributos. De este modo, los atributos *name* y *model3D* pasarán a formar parte de las tablas *Item* y *Showcase*, mientras que el atributo *isItem* desaparecerá al poderse diferenciar directamente en *MIHElement* gracias al atributo *type*.

  El resto de consultas similares a esta no se pueden optimizar más, pues la búsqueda se realiza a través de la clave y no hace falta definir otros atributos como índices. Así pues, esta consulta queda reducida a la siguiente:

```
SELECT *
FROM ObjectCategory
WHERE id=<id>;

SELECT *
FROM Showcase
WHERE id=<id>;
SELECT *
FROM Shelf
WHERE showcase=<id>;
```

  La consulta e también se ve beneficiada por esta decisión, pues no tendrá que modificar la tabla *MIHObject*, resultando en la siguiente nueva consulta:

```
UPDATE MIHElement SET lastUpdate=NOW() WHERE id=<id>;
UPDATE Item name=<nuevo nombre>, model3D=<nuevoModelo>,
      info=<nueva descripcion>, preview=<nuevo preview>,
      isFlat=<es plano?> WHERE id=<id>;
```

- Se definió el siguiente *trigger* para automatizar las inserciones de objetos, items, clases de reconocimiento y texturas, asi como las inserciones en tablas que afectan a ellas como *localizes displays*:

```
CREATE TRIGGER insertParents<table>
BEFORE INSERT ON <table>
FOR EACH ROW
BEGIN
  INSERT INTO MIHElement (type, lastUpdate, isDeleted)
  VALUES (0, NOW(), false);

  -- En caso de item o showcase
  INSERT INTO MIHObject (id) VALUES (LAST_INSERT_ID());

  SET NEW.id=LAST_INSERT_ID();
END$$
```

  Cabe destacar que las inserciones realizadas desde dentro del trigger no se llevarán a

cabo si se produce algún error durante el insertado (debido por ejemplo a un incumplimiento de la restricción de unicidad de *name* en *item*). De este modo, no es necesario que el administrador inserte una tupla en la tabla *MIHElement* ni en *MIHObject* en la consulta d, que queda reducida a la siguiente consulta:

```
INSERT INTO Item (name, model3D, info, preview, isFlat)
  VALUES (<nombre>, <modelo 3D>, <descripcion>, <preview>, <es plano?>);
INSERT INTO ObjectCategory (object, category) VALUES (LAST_INSERT_ID(),
    <categoria 1>);
...
INSERT INTO ObjectCategory (object, category) VALUES (LAST_INSERT_ID(),
    <categoria n>);
```

- Se definió el siguiente *trigger* para automatizar las modificaciones de objetos, items, imágenes de reconocimiento y texturas, asi como las modificaciones en tablas que afectan a ellas como *localizes displays*:

```
DELIMITER $$
CREATE TRIGGER setUpdated<tabla>
AFTER UPDATE ON <tabla>
FOR EACH ROW
BEGIN
  UPDATE MIHElement
  SET lastUpdate=NOW()
  WHERE id=OLD.id;
END$$
```

De este modo, no es necesario que el administrador modifique la tabla *MIHElement* en la consulta e, que queda reducida a la siguiente consulta:

```
UPDATE Item name=<nuevo nombre>, model3D=<nuevoModelo>,
        info=<nueva descripcion>, preview=<nuevo preview>,
        isFlat=<es plano?> WHERE id=<id>;
```

- Se definió el siguiente *trigger* para automatizar el borrado de objetos, items, imágenes de reconocimiento y texturas, asi como los borrados en tablas que afectan a ellas como *localizes displays*:

```
DELIMITER $$
CREATE TRIGGER setDeleted<tabla>
BEFORE DELETE ON <tabla>
FOR EACH ROW
BEGIN
  UPDATE MIHElement
  SET lastUpdate=NOW(), isDeleted=true
  WHERE id=OLD.id;
END$$
```

De este modo, no es necesario que el administrador modifique la tabla *MIHElement* en la consulta f, que queda reducida a la siguiente consulta:

```
DELETE FROM <tabla> WHERE id=<id>;
```

- Se realizaron las siguientes particiones verticales en *RecognitionClass* para evitar introducir muchos valores nulos:
  - *Features2try*. Puesto que la mayoría de las veces no vendrán definidos, se han separado los atributos *keypoints2try*, *descriptors2try*, *threshold2try* de *RecognitionClass* en esta nueva tabla.
  - *RecognitionClassARData*. Por la misma razón, se han separado los atributos de transformación (posición, rotación y escala) de *RecognitionClass* en esta nueva tabla.
- Se separó el atributo *histogram* en *RecognitionClass* en una nueva tabla con el mismo nombre. Esto se debe a que, tal y como se explica en la consulta h, cada vez que se ha de actualizar el vocabulario de palabras visuales en *RecognitionData* se han de actualizar los histogramas de todas las clases de reconocimiento. Como también se definió el *trigger* de actualización en *RecognitionClass*, si se actualiza un histograma se marcará ese elemento como actualizado y la app descargará todos los datos del elemento, incluyendo aquellos pesados como los puntos de interés y los descriptores. Por ello, separando el histograma en la nueva tabla, no se realizará marcará la clase de reconocimiento asociada y la obtención de datos será más eficiente. Esta solución requerirá que cuando la app detecte un cambio en el vocabulario descargue explícitamente los datos de esta tabla. También se deberán descargar cuando se obtengan los datos de la clase de reconocimiento asociada.

Tras aplicar estas modificaciones se consideró que el diseño estaba lo suficientemente optimizado, por lo que se procedió directamente con su implementación en MySQL.

## B.6. Modelo SQL definitivo

Tras las decisiones tomadas durante el diseño físico se llegó a la siguiente implementación en MySQL, la cual incluye tanto las tablas como los triggers anteriormente definidos:

```
-- CREATE TABLES
CREATE TABLE MIHElement (
  id         MEDIUMINT NOT NULL AUTO_INCREMENT,
  type       TINYINT NOT NULL,
  lastUpdate DATETIME NOT NULL,
  isDeleted  BOOLEAN NOT NULL,
  PRIMARY KEY(id),
  INDEX lvu (lastUpdate));

CREATE TABLE RecognitionData (
  id      MEDIUMINT NOT NULL,
```

```
  name     VARCHAR(100) NOT NULL,
  data     MEDIUMBLOB NOT NULL,
  PRIMARY KEY(id),
  FOREIGN KEY(id)
     REFERENCES MIHElement(id)
     ON DELETE CASCADE
     ON UPDATE CASCADE,
  INDEX recdi (name));

CREATE TABLE MIHObject (
  id       MEDIUMINT NOT NULL,
  PRIMARY KEY(id),
  FOREIGN KEY(id)
     REFERENCES MIHElement(id)
     ON DELETE CASCADE
     ON UPDATE CASCADE);

CREATE TABLE ObjectCategory (
  object     MEDIUMINT NOT NULL,
  category   VARCHAR(100),
  PRIMARY KEY(object, category),
  FOREIGN KEY(object)
     REFERENCES MIHElement(id)
     ON DELETE CASCADE
     ON UPDATE CASCADE);

CREATE TABLE Texture (
  id         MEDIUMINT NOT NULL,
  object     MEDIUMINT NOT NULL,
  slot       INTEGER,
  type       INTEGER,
  texData    MEDIUMBLOB NOT NULL,
  PRIMARY KEY(id),
  FOREIGN KEY(id)
     REFERENCES MIHElement(id)
     ON DELETE CASCADE
     ON UPDATE CASCADE,
  FOREIGN KEY(object)
     REFERENCES MIHObject(id)
     ON DELETE CASCADE
     ON UPDATE CASCADE,
  INDEX obji (object));

CREATE TABLE Showcase (
  id            MEDIUMINT NOT NULL,
  name          VARCHAR(100) UNIQUE,
  model3d       MEDIUMBLOB,
  isVirtual     BOOLEAN NOT NULL,
  mapPositionX  DOUBLE,
```

```
  mapPositionY  DOUBLE,
  PRIMARY KEY(id),
  FOREIGN KEY(id)
     REFERENCES MIHElement(id)
     ON DELETE CASCADE
     ON UPDATE CASCADE);

CREATE TABLE Shelf (
  showcase        MEDIUMINT NOT NULL,
  num             TINYINT,
  heightFromBase DOUBLE,
  PRIMARY KEY(showcase, num),
  FOREIGN KEY(showcase)
     REFERENCES Showcase(id)
     ON DELETE CASCADE
     ON UPDATE CASCADE);

CREATE TABLE RecognitionClass (
  id      MEDIUMINT NOT NULL,
  recognizesO        MEDIUMINT,
  recognizesShowcase MEDIUMINT,
  recognizesShelf    TINYINT,
  cornerULX     DOUBLE,
  cornerULY     DOUBLE,
  cornerURX     DOUBLE,
  cornerURY     DOUBLE,
  cornerDLX     DOUBLE,
  cornerDLY     DOUBLE,
  cornerDRX     DOUBLE,
  cornerDRY     DOUBLE,
  keypointsORB    MEDIUMBLOB NOT NULL,
  descriptorsORB MEDIUMBLOB NOT NULL,
  thresholdORB  INTEGER,
  descType2try  TINYINT,
  imageWidth    INTEGER NOT NULL,
  imageHeight   INTEGER NOT NULL,
  realWidth     INTEGER NOT NULL,
  realHeight    INTEGER NOT NULL,
  PRIMARY KEY(id),
  FOREIGN KEY(id)
     REFERENCES MIHElement(id)
     ON DELETE CASCADE
     ON UPDATE CASCADE,
  FOREIGN KEY(recognizesO)
     REFERENCES MIHObject(id)
     ON DELETE CASCADE
     ON UPDATE CASCADE,
  FOREIGN KEY(recognizesShowcase, recognizesShelf)
     REFERENCES Shelf(showcase, num)
```

```
    ON DELETE CASCADE
    ON UPDATE CASCADE,
  INDEX roi (recognizesO),
  INDEX rsci (recognizesShowcase),
  INDEX rsi (recognizesShelf));

CREATE TABLE Features2try (
  id              MEDIUMINT NOT NULL,
  keypoints2try   MEDIUMBLOB,
  descriptors2try MEDIUMBLOB,
  threshold2try INTEGER,
  PRIMARY KEY(id),
  FOREIGN KEY(id)
    REFERENCES RecognitionClass(id)
    ON DELETE CASCADE
    ON UPDATE CASCADE);

CREATE TABLE RecognitionClassARData (
  id      MEDIUMINT NOT NULL,
  positionX     DOUBLE,
  positionY     DOUBLE,
  positionZ     DOUBLE,
  rotationX     DOUBLE,
  rotationY     DOUBLE,
  rotationZ     DOUBLE,
  scaleX        DOUBLE,
  scaleY        DOUBLE,
  scaleZ        DOUBLE,
  PRIMARY KEY(id),
  FOREIGN KEY(id)
    REFERENCES RecognitionClass(id)
    ON DELETE CASCADE
    ON UPDATE CASCADE);

CREATE TABLE Histogram (
  id      MEDIUMINT NOT NULL,
  histogram  MEDIUMBLOB NOT NULL,
  PRIMARY KEY(id),
  FOREIGN KEY(id)
    REFERENCES RecognitionClass(id)
    ON DELETE CASCADE
    ON UPDATE CASCADE);

CREATE TABLE Item (
  id         MEDIUMINT NOT NULL,
  name       VARCHAR(100) UNIQUE,
  model3d    MEDIUMBLOB,
  info       VARCHAR(3000),
  preview    BLOB,
```

```
  isFlat     BOOLEAN NOT NULL,
  PRIMARY KEY(id),
  FOREIGN KEY(id)
     REFERENCES MIHObject(id)
     ON DELETE CASCADE
     ON UPDATE CASCADE);

CREATE TABLE ItemInstance (
  id         MEDIUMINT NOT NULL,
  item       MEDIUMINT NOT NULL,
  showcase   MEDIUMINT,
  shelf      TINYINT,
  isVirtual  BOOLEAN NOT NULL,
  positionX  DOUBLE,
  positionY  DOUBLE,
  positionZ  DOUBLE,
  rotationX  DOUBLE,
  rotationY  DOUBLE,
  rotationZ  DOUBLE,
  scaleX     DOUBLE,
  scaleY     DOUBLE,
  scaleZ     DOUBLE,
  PRIMARY KEY(id),
  FOREIGN KEY(id)
     REFERENCES MIHElement(id)
     ON DELETE CASCADE
     ON UPDATE CASCADE,
  FOREIGN KEY(item)
     REFERENCES Item(id)
     ON DELETE CASCADE
     ON UPDATE CASCADE,
  FOREIGN KEY(showcase, shelf)
     REFERENCES Shelf(showcase, num)
     ON DELETE SET NULL
     ON UPDATE CASCADE,
  INDEX itemi (item));


CREATE TABLE localizes (
  instance        MEDIUMINT NOT NULL,
  recognitionClass MEDIUMINT NOT NULL,
  cornerULX     DOUBLE,
  cornerULY     DOUBLE,
  cornerURX     DOUBLE,
  cornerURY     DOUBLE,
  cornerDLX     DOUBLE,
  cornerDLY     DOUBLE,
  cornerDRX     DOUBLE,
  cornerDRY     DOUBLE,
```

```
  PRIMARY KEY(instance, recognitionClass),
  FOREIGN KEY(instance)
     REFERENCES ItemInstance(id)
     ON DELETE CASCADE
     ON UPDATE CASCADE,
  FOREIGN KEY(recognitionClass)
     REFERENCES RecognitionClass(id)
     ON DELETE CASCADE
     ON UPDATE CASCADE);

CREATE TABLE displays (
  instance        MEDIUMINT NOT NULL,
  recognitionClass MEDIUMINT NOT NULL,
  positionX  DOUBLE,
  positionY  DOUBLE,
  positionZ  DOUBLE,
  rotationX  DOUBLE,
  rotationY  DOUBLE,
  rotationZ  DOUBLE,
  scaleX     DOUBLE,
  scaleY     DOUBLE,
  scaleZ     DOUBLE,
  PRIMARY KEY(instance, recognitionClass),
  FOREIGN KEY(instance)
     REFERENCES ItemInstance(id)
     ON DELETE CASCADE
     ON UPDATE CASCADE,
  FOREIGN KEY(recognitionClass)
     REFERENCES RecognitionClass(id)
     ON DELETE CASCADE
     ON UPDATE CASCADE);


-- CREATE TRIGGERS
DELIMITER $$

-- Insert triggers
CREATE TRIGGER insertParentsItem
BEFORE INSERT ON Item
FOR EACH ROW
BEGIN
  INSERT INTO MIHElement (type, lastUpdate, isDeleted)
  VALUES (0, NOW(), false);
  INSERT INTO MIHObject (id) VALUES (LAST_INSERT_ID());
  SET NEW.id=LAST_INSERT_ID();
END$$

CREATE TRIGGER insertParentsShowcase
BEFORE INSERT ON Showcase
```

```
FOR EACH ROW
BEGIN
  INSERT INTO MIHElement (type, lastUpdate, isDeleted)
  VALUES (1, NOW(), false);
  INSERT INTO MIHObject (id) VALUES (LAST_INSERT_ID());
  SET NEW.id=LAST_INSERT_ID();
END$$

CREATE TRIGGER insertParentsTexture
BEFORE INSERT ON Texture
FOR EACH ROW
BEGIN
  INSERT INTO MIHElement (type, lastUpdate, isDeleted)
  VALUES (2, NOW(), false);
  SET NEW.id=LAST_INSERT_ID();
END$$

CREATE TRIGGER insertParentsRC
BEFORE INSERT ON RecognitionClass
FOR EACH ROW
BEGIN
  INSERT INTO MIHElement (type, lastUpdate, isDeleted)
  VALUES (3, NOW(), false);
  SET NEW.id=LAST_INSERT_ID();
END$$

CREATE TRIGGER insertParentsItemInstance
BEFORE INSERT ON ItemInstance
FOR EACH ROW
BEGIN
  INSERT INTO MIHElement (type, lastUpdate, isDeleted)
  VALUES (4, NOW(), false);
  SET NEW.id=LAST_INSERT_ID();
END$$

CREATE TRIGGER insertParentsRecognitionData
BEFORE INSERT ON RecognitionData
FOR EACH ROW
BEGIN
  INSERT INTO MIHElement (type, lastUpdate, isDeleted)
  VALUES (5, NOW(), false);
  SET NEW.id=LAST_INSERT_ID();
END$$


-- Update triggers
CREATE TRIGGER setUpdatedItem
AFTER UPDATE ON Item
FOR EACH ROW
```

```
BEGIN
  UPDATE MIHElement
  SET lastUpdate=NOW()
  WHERE id=OLD.id;
END$$

CREATE TRIGGER setUpdatedShowcase
AFTER UPDATE ON Showcase
FOR EACH ROW
BEGIN
  UPDATE MIHElement
  SET lastUpdate=NOW()
  WHERE id=OLD.id;
END$$

CREATE TRIGGER setUpdatedRecClass
AFTER UPDATE ON RecognitionClass
FOR EACH ROW
BEGIN
  UPDATE MIHElement
  SET lastUpdate=NOW()
  WHERE id=OLD.id;
END$$

CREATE TRIGGER setUpdatedRecClassArDataOnInsert
AFTER INSERT ON RecognitionClassARData
FOR EACH ROW
BEGIN
  UPDATE MIHElement
  SET lastUpdate=NOW()
  WHERE id=NEW.id;
END$$

CREATE TRIGGER setUpdatedRecClassOnUpdateArData
AFTER UPDATE ON RecognitionClassARData
FOR EACH ROW
BEGIN
  UPDATE MIHElement
  SET lastUpdate=NOW()
  WHERE id=OLD.id;
END$$

CREATE TRIGGER setUpdatedRecClassOnDeleteArData
AFTER DELETE ON RecognitionClassARData
FOR EACH ROW
BEGIN
  UPDATE MIHElement
  SET lastUpdate=NOW()
  WHERE id=OLD.id;
```

```
END$$

CREATE TRIGGER setUpdatedRecClassOnInsertFeatures2try
AFTER INSERT ON Features2try
FOR EACH ROW
BEGIN
  UPDATE MIHElement
  SET lastUpdate=NOW()
  WHERE id=NEW.id;
END$$

CREATE TRIGGER setUpdatedRecClassOnUpdateFeatures2try
AFTER UPDATE ON Features2try
FOR EACH ROW
BEGIN
  UPDATE MIHElement
  SET lastUpdate=NOW()
  WHERE id=OLD.id;
END$$

CREATE TRIGGER setUpdatedRecClassOnDeleteFeatures2try
AFTER DELETE ON Features2try
FOR EACH ROW
BEGIN
  UPDATE MIHElement
  SET lastUpdate=NOW()
  WHERE id=OLD.id;
END$$

CREATE TRIGGER setUpdatedItemInstance
AFTER UPDATE ON ItemInstance
FOR EACH ROW
BEGIN
  UPDATE MIHElement
  SET lastUpdate=NOW()
  WHERE id=OLD.id;
END$$

CREATE TRIGGER setUpdatedItemInstanceOnInsertLocalizes
AFTER INSERT ON localizes
FOR EACH ROW
BEGIN
  UPDATE MIHElement
  SET lastUpdate=NOW()
  WHERE id=NEW.instance;
END$$

CREATE TRIGGER setUpdatedItemInstanceOnUpdateLocalizes
AFTER UPDATE ON localizes
```

```
FOR EACH ROW
BEGIN
  UPDATE MIHElement
  SET lastUpdate=NOW()
  WHERE id=OLD.instance;
END$$

CREATE TRIGGER setUpdatedItemInstanceOnDeleteLocalizes
AFTER DELETE ON localizes
FOR EACH ROW
BEGIN
  UPDATE MIHElement
  SET lastUpdate=NOW()
  WHERE id=OLD.instance;
END$$

CREATE TRIGGER setUpdatedItemInstanceOnInsertDisplays
AFTER INSERT ON displays
FOR EACH ROW
BEGIN
  UPDATE MIHElement
  SET lastUpdate=NOW()
  WHERE id=NEW.instance;
END$$

CREATE TRIGGER setUpdatedItemInstanceOnUpdateDisplays
AFTER UPDATE ON displays
FOR EACH ROW
BEGIN
  UPDATE MIHElement
  SET lastUpdate=NOW()
  WHERE id=OLD.instance;
END$$

CREATE TRIGGER setUpdatedItemInstanceOnDeleteDisplays
AFTER DELETE ON displays
FOR EACH ROW
BEGIN
  UPDATE MIHElement
  SET lastUpdate=NOW()
  WHERE id=OLD.instance;
END$$

CREATE TRIGGER setUpdatedTex
AFTER UPDATE ON Texture
FOR EACH ROW
BEGIN
  UPDATE MIHElement
  SET lastUpdate=NOW()
```

```
  WHERE id=OLD.id;
END$$

CREATE TRIGGER setUpdatedRecData
AFTER UPDATE ON RecognitionData
FOR EACH ROW
BEGIN
  UPDATE MIHElement
  SET lastUpdate=NOW()
  WHERE id=OLD.id;
END$$

-- Delete triggers
CREATE TRIGGER setDeletedObj
BEFORE DELETE ON MIHObject
FOR EACH ROW
BEGIN
  UPDATE MIHElement
  SET lastUpdate=NOW(), isDeleted=true
  WHERE id=OLD.id;
END$$

CREATE TRIGGER setDeletedRecClass
BEFORE DELETE ON RecognitionClass
FOR EACH ROW
BEGIN
  UPDATE MIHElement
  SET lastUpdate=NOW(), isDeleted=true
  WHERE id=OLD.id;
END$$

CREATE TRIGGER setDeletedItemInstance
BEFORE DELETE ON ItemInstance
FOR EACH ROW
BEGIN
  UPDATE MIHElement
  SET lastUpdate=NOW(), isDeleted=true
  WHERE id=OLD.id;
END$$

CREATE TRIGGER setDeletedTex
BEFORE DELETE ON Texture
FOR EACH ROW
BEGIN
  UPDATE MIHElement
  SET lastUpdate=NOW(), isDeleted=true
  WHERE id=OLD.id;
END$$
```

```
CREATE TRIGGER setDeletedRecognitionData
BEFORE DELETE ON RecognitionData
FOR EACH ROW
BEGIN
  UPDATE MIHElement
  SET lastUpdate=NOW(), isDeleted=true
  WHERE id=OLD.id;
END$$

DELIMITER ;
```

## B.7. Diferencias con BD de MIHex

La base de datos utilizada en la aplicación se implementó bajo la misma estructura que la utilizada en el servidor, pues al almacenar la misma información las actualizaciones resultan más sencillas. Se podría haber realizado un diseño físico que optimizase esta BD, pero las consultas realizadas desde la app resultaron bastante simples y no requerían este proceso para llevarse a cabo eficientemente. Sin embargo, se llevaron a cabo pequeñas modificaciones:

- Se suprimió la tabla *MIHElement*, pues únicamente era necesaria en la BD del servidor para facilitar las actualizaciones. En la app no aportaba ninguna utilidad.
- Se suprimió la clave *id* de *RecognitionData* y se sustituyó por *name*, pues se realizaban búsquedas a través de este atributo. La *id* únicamente era utilizada para la actualización, pero como también era posible realizarla a través del nombre no aportaba utilidad extra.
- Se eliminaron los índices de las tablas, pues, debido al SGBD utilizado en Android (SQLite), no era posible definirlos.

## B.8. Modelos descartados

En una primera iteración se consideró que el propósito de la base de datos era únicamente almacenar la información de los objetos del museo. Por ello se optó por un diseño simple, con dos únicas entidades y sin relaciones entre ambas. Este diseño queda reflejado en el modelo ER de la figura B.11.

Tras finalizar la primera iteración, quedaron a la vista algunas de las deficiencias de este modelo. El principal problema era almacenar toda la información de los objetos en una única entidad. De esta forma, si se realizaba cualquier pequeño cambio en un objeto (por ejemplo cambiar el nombre o la estantería), era necesario descargar de nuevo toda la información que tiene asociada, incluyendo campos pesados como los descriptores de AKAZE, las texturas o los modelos. Todo esto repercutía en el tiempo de actualización de la app, que crecía considerablemente.

Para el comienzo de una segunda iteración, se solucionó este problema disgregando la entidad *MIHObject* en varias, añadiéndoles además varios atributos para otorgar nuevas funcionalidades. También se especializó la entidad *MIHObject* para dar soporte a las estanterías

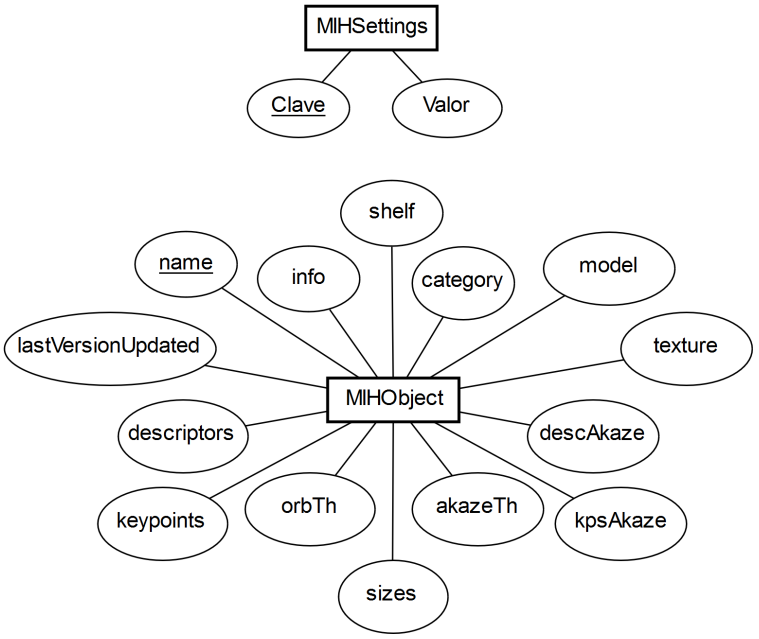



Figura B.11: Primera versión del modelo Entidad-Relación

virtuales y los objetos que contendrán. El modelo de esta iteración está reflejado en la figura B.12. De esta forma la app solo descargará los campos pesados de las entidades que se hayan modificado. Además, al añadir nuevos atributos y relaciones, las nuevas entidades han adquirido más poder semántico respecto al diseño anterior.
Sin embargo, esto no supuso la versión definitiva, pues aparecieron nuevas deficiencias:

- La nomenclatura de *objects* resultaba confusa, sobre todo al especializarse de nuevo en estanterías. Lo mismo ocurría con la entidad *MIHSettings* y la relación *moves*.
- Se consideró que la entidad *Model3D* no era necesaria, pues al corresponder únicamente a un *MIHObject* (y viceversa) se puede mover el atributo *modelData* a el propio *MIHObject*.
- Se decidió no utilizar AKAZE, por lo que los campos que almacenaban sus descriptores y keypoints tampoco son necesarios.
- No se contemplaba la posibilidad de detectar objetos virtuales dentro de estanterías virtuales.
- No permitía el reconocimiento por baldas.
- No se contemplaba la posibilidad de que un mismo objeto estuviera varias veces en el museo, en estanterías diferentes.
- No se había considerado la posibilidad de que un objeto perteneciera a varias categorías.

Tras corregir estos aspectos se elaboró una tercera versión de la base de datos, apreciable en la figura B.13.

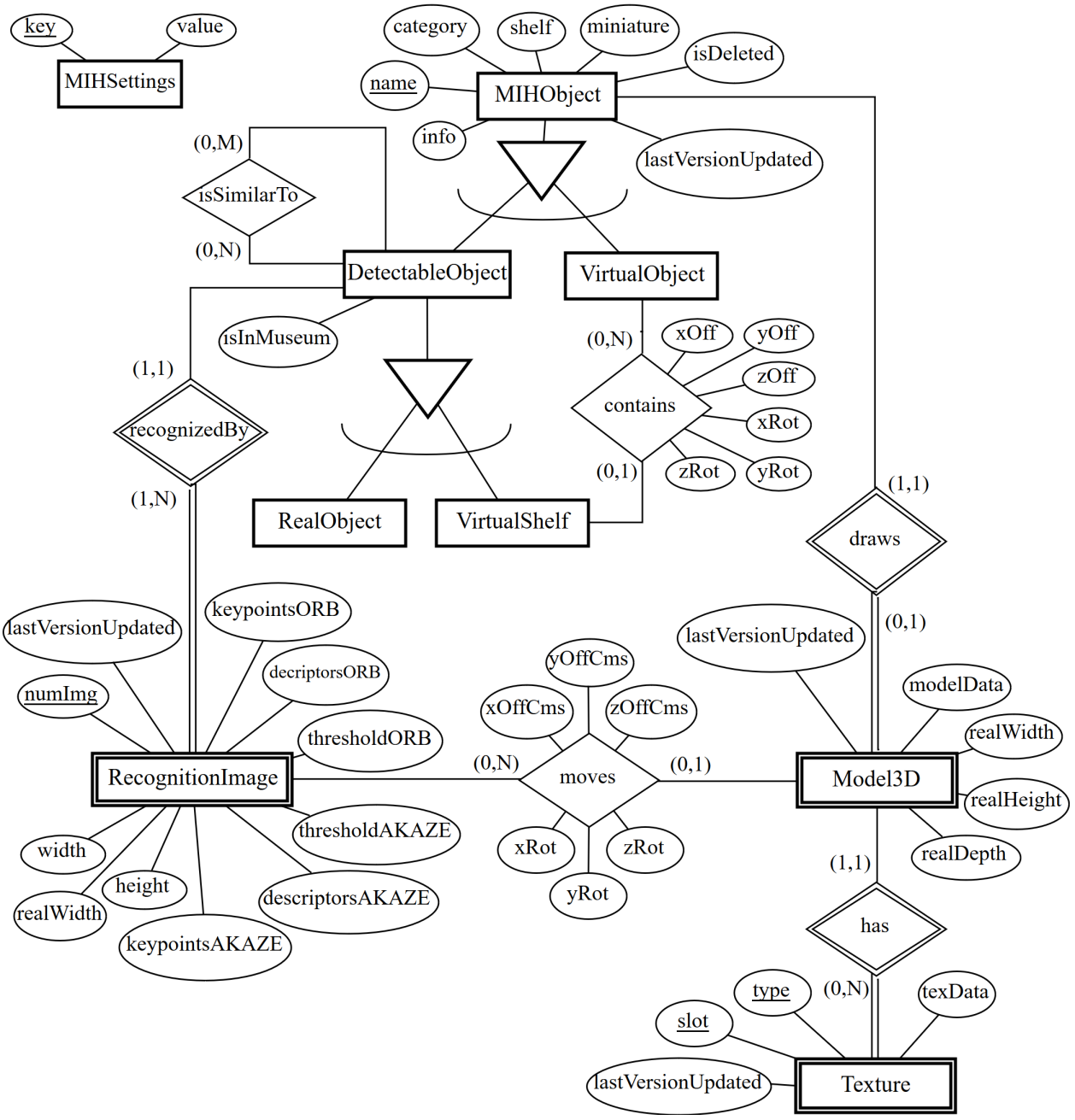



Figura B.12: Segunda versión del modelo Entidad-Relación

Esta versión se realizó antes de la elaboración del algoritmo de la bolsa de palabras, por lo que se tuvo que volver a retocar para incluir los datos que eran necesarios para este nuevo reconocimiento. Junto con unas pequeñas consideraciones de concepto en las instancias de los items del museo, estos cambios definieron la última versión de la base de datos.

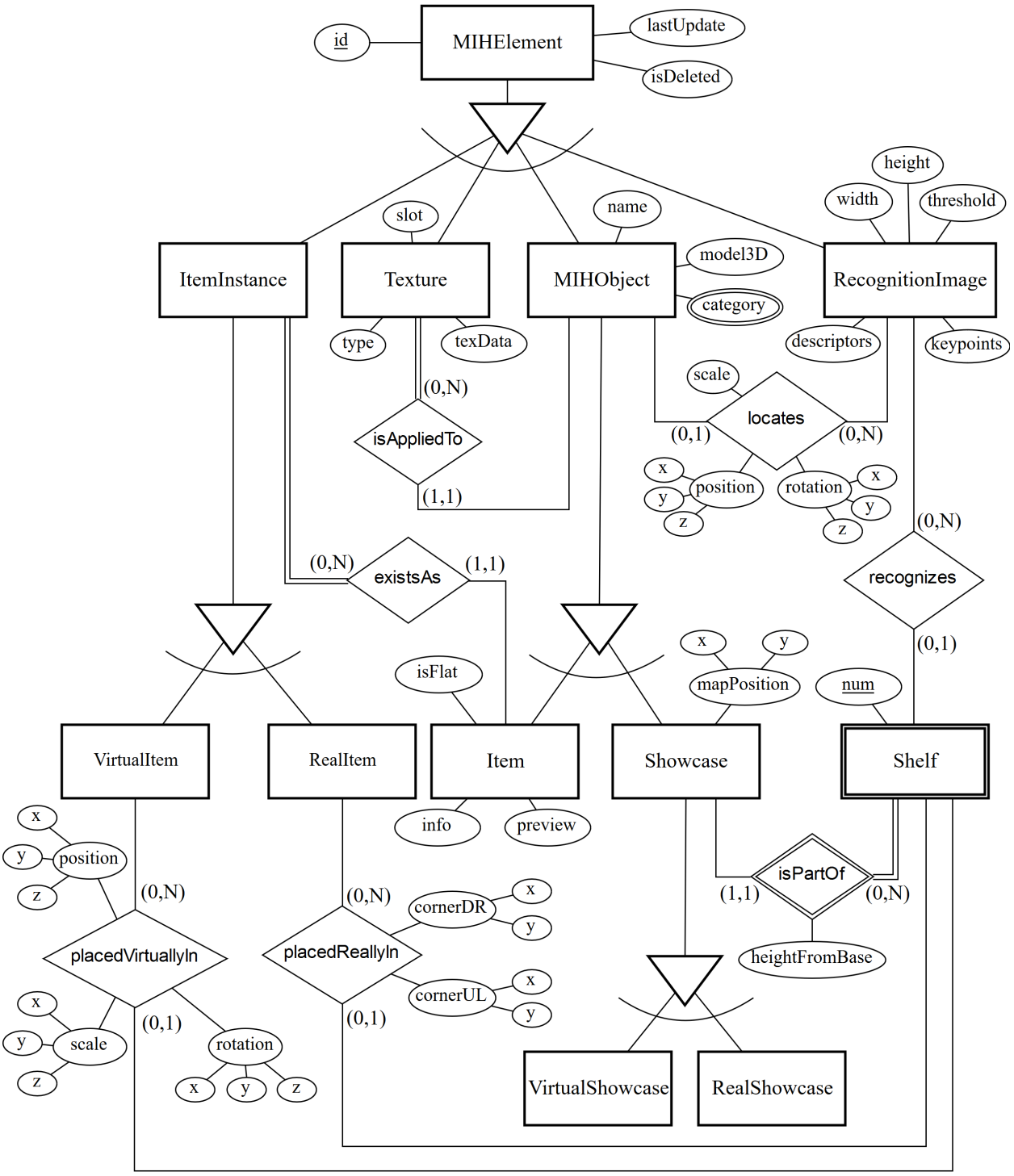



Figura B.13: Tercera versión del modelo Entidad-Relación

# Anexo C

# Optimizaciones

El reconocimiento de objetos ha sido el tema tratado que ha supuesto una mayor dificultad. Este anexo pretende recoger todos los problemas surgidos, las posibilidades exploradas, las optimizaciones aplicadas y las soluciones tomadas. Está estrechamente relacionado con el apartado 4, donde se muestran únicamente las metodologías y estrategias finales utilizadas.

## C.1. Mejora del rendimiento

La estrategia inicial para la detección de objetos consistía en reconocer los objetos en tiempo real para cada *frame* de la cámara. Tras implementar la primera versión de la aplicación, programada enteramente en Java, se comprobó que esta estrategia no funcionaba a tiempo real. Además, al añadir en un futuro más objetos que los utilizados para las pruebas el sistema resultaría inviable. Por ello, se exploraron a fondo las siguientes optimizaciones:

| **Opción** | **Uso en MIHex** |
|---|---|
| Reducir escala de la imagen | Utilizada |
| Utilizar JNI | Utilizada |
| Utilizar TBB | Utilizada (por defecto) |
| Utilizar openCL + TAPI | Descartada |
| Utilizar openCL con kernels traducidos de openCV | Descartada |
| Utilizar renderscript | Descartada |
| Volver a la versión compilada de openCV para Android | Utilizada |
| Finalización temprana | Activable |
| Transcripción de la generación de histogramas a C++ | Utilizada |
| Cálculo manual de la distancia de Hamming | Utilizada |
| Uso de *multithreading* | Utilizada |

Tabla C.1: Tabla resumen de las optimizaciones realizadas y su uso en la aplicación

A continuación se detalla cada una de ellas, describiendo para cada una su objetivo, sus requerimientos, sus resultados y la conclusión obtenida.

### C.1.1. Reducir la escala de la imagen

- Contexto: ambas iteraciones.
- Objetivo: reducir la escala de la imagen a la mitad hace posible encontrar los puntos ORB hasta 4 veces más rápido.

- Requiere: reducir el tamaño de la imagen.
- Resultado: el resultado es satisfactorio, aunque no afecta en la parte de búsqueda de objetos.
- Conclusión: se mantiene activa esta opción, pudiéndose modificar desde el menú de la *app*.

### C.1.2. Utilizar JNI

- Contexto: ambas iteraciones.
- Objetivo: el llevar todo lo posible el codigo implementado en Java a C++ permitiría un posible incremento de tiempo. La programación en C++ es más eficiente que en Java, pero además las llamadas JNI desde Java suponen un pequeño impacto en el tiempo de ejecución. El uso de las funciones de openCV desde la interfaz de Java conlleva al menos una llamada JNI por función, lo cual se puede reducir llamando a openCV únicamente desde el código nativo propio que englobe el máximo número posible de funciones de openCV.
- Requiere: aprender como funciona JNI en Android y reprogramar el código en C++.
- Resultado: el tiempo ganado reduciendo las llamadas JNI de openCV desde Java no supuso un impacto considerable para el sistema. Por otro lado, el uso de las funciones de openCV desde C++ otorga mucha más flexibilidad que la interfaz proporcionada a Java.
- Conclusión: el sistema no se verá potencialmente mejorado por esta opción. El tiempo de ejecución de las llamadas a las funciones críticas de openCV no resulta suficiente para una aplicación a tiempo real en Android, por lo que es necesario una búsqueda de nuevas alternativas. Sin embargo, se decidió mantener el uso de JNI en un futuro por la flexibilidad que ofrece al tratar con las funciones de openCV.

### C.1.3. Utilizar TBB

- Contexto: ambas iteraciones.
- Objetivo: ciertas funciones paralelizables de openCV se pueden optimizar realizándolas en paralelo a través de TBB, aprovechando en la medida de lo posible los distintos núcleos de la CPU.
- Requiere: recompilar openCV con la opción de TBB habilitada.
- Resultado: se comprobó que openCV ya estaba utilizando TBB por defecto.
- Conclusión: la librería openCV ya está explotando esta posibilidad, por lo que no es necesaria ninguna modificación.

### C.1.4. Utilizar openCL + TAPI de openCV

- Contexto: ambas iteraciones.
- Objetivo: llevar las funciones paralelizables de openCV a la GPU permitiría un cálculo más rápido de las mismas. Esto puede realizarse a través de librerías GPGPU, como openCL[5]. Además, openCV tiene ya implementadas las funciones más costosas en openCL, que pueden utilizarse de forma transparente gracias a una 'Transparent API' (TAPI). De esta forma sería posible aprovechar la capacidad de la GPU únicamente cambiando las matrices necesarias del formato 'Mat' a 'UMat' en el código propio.
- Requiere: recompilar openCV con la opción de openCL habilitada y cambiar todas las variables 'Mat' por 'UMat'.
- Resultado: hubo que recompilar la librería de openCV para habilitar esta opción. Sin embargo, a pesar de seguir todos los pasos no se logró hacerlo funcionar. Tras buscar más información sobre los motivos del problema se encontró que esta opción no funciona en dispositivos Android.
- Conclusión: hubiera sido buena opción si openCV permitiera openCL en Android, pero de momento resultó imposible esta implementación.

### C.1.5. Utilizar openCL con código fuente de openCV dentro de la app

- Contexto: ambas iteraciones.
- Objetivo: en lugar de hacer llamadas a openCL desde el código previamente compilado de openCV, debería ser posible hacer llamadas directamente a openCL en un dispositivo Android. Al ser openCV una librería *opensource* es posible acceder a su código fuente y extraer las funciones necesarias. De esta forma se puede incluir el código de openCV directamente en la *app* y hacer llamadas a openCL directamente.
- Requiere: extraer la librería necesaria directamente del dispositivo a utilizar, añadir los *headers* de openCL e incluir el código necesario de openCV en la aplicación.
- Conclusión: no se consiguió incluir con éxito la librería de openCL directamente. Además, el uso de openCL en Android no está soportado por Google.
- Resultado: se decidió descartar esta opción dada la dificultad de añadir esta librería y, además, para incluirla es necesario extraer la librería anteriormente comentada de cada dispositivo antes de compilar la aplicación. Esto haría el proceso de instalación muy tedioso para usuarios externos. Se podría retomar la idea en un futuro con fines experimentales.

### C.1.6. Utilizar renderscript

- Contexto: ambas iteraciones.

- Objetivo: como contrapartida a openCL, Google ofrece su *framework renderscript*. A diferencia de openCL, *renderscript* puede funcionar tanto en GPU como en CPU. Además, el programador no posee control para determinar en cuál de ellos han de realizarse los cómputos.

- Requiere: aprender el funcionamiento de *renderscript* y traducir los *kernels* de openCL del código fuente de openCV a *renderscript*.

- Resultado: se llegó a transcribir la obtención de puntos FAST, su puntuación de Harris y el ángulo, es decir, los cálculos necesarios para obtener los mejores puntos ORB, excepto la supresión de no máximos. Esto es debido a que se comprobó que los cómputos nunca se realizaban en GPU, y por tanto, se descartó esta opción antes de continuar transcribiendo el código de openCV a *renderscript*. Resultó que para poder realizarse en GPU el código debía cumplir determinadas condiciones, las cuales no era posible asumir dados los requerimientos de las funciones a programar. Se probó a implementar *scripts* que sí cumplían con esas condiciones, incluso en formato *filterscript* (que en teoría garantiza computo en GPU), pero no se apreció ningún proceso en GPU a través del monitor de Android Studio.

- Conclusión: la versión actual de *renderscript* no permite realizar los cómputos necesarios en GPU, y por tanto, no se obtiene ninguna mejora de rendimiento. Por ello, se decidió descartar esta opción antes de continuarla.

### C.1.7. Volver a la versión compilada de openCV para Android

- Contexto: ambas iteraciones.

- Objetivo: hasta este momento se estaba trabajando con la versión de openCV compilada manualmente con las opciones requeridas para admitir openCL. Esta versión se compiló únicamente para los procesadores con instrucciones armeabi-v7a, por lo que la compatibilidad de la aplicación era bastante limitada. Con el objetivo de ampliar esta compatibilidad se volvió a la versión que ofrece openCV específica para Android, la cual es compatible con 7 abis distintas.

- Requiere: eliminar la versión actual de openCV del proyecto e incorporar la compilada para Android.

- Resultado: esta acción trajo consigo efectos secundarios positivos en tiempo de ejecución, posiblemente por realizar un mejor uso del juego de instrucciones de openCV.

- Conclusión: se ha decidido mantener esta opción, pues no se iba a incorporar openCL en un futuro cercano, y las ventajas que otorgaba eran bastante satisfactorias.

### C.1.8. Finalización temprana

- Contexto: primera iteración.

- Objetivo: hasta ahora las búsquedas para el reconocimiento se estaban realizando siempre para toda la colección, con el fin de poder representar varios reconocimientos en pantalla al mismo tiempo. Sin embargo, esta posiblidad resulto prescindible, por lo que pudo realizarse la siguiente optimización: buscar en la colección hasta que reconozca un objeto, es decir, en el momento en el que encuentra un objeto deja de buscar.
- Requiere: habilitar la opción.
- Resultado: puesto que la busqueda se realiza iterando sobre una lista de objetos, esta optimización favorece al reconocimiento de los primeros objetos de la colección. Resulta especialmente útil en el caso de reconocimiento mediante puntos AKAZE, pues son más costosos de computar. Por otro lado, esta opción crea problemas en el caso de intentar reconocer objetos que puedan causar confusión, pues puede reconocer un falso positivo antes que encontrar el objeto que realmente corresponde. Una posible solución a este problema podría ser establecer familias de objetos confundibles para que, de esta forma, al reconocer un objeto perteneciente a una familia de confundibles pudiera realizar una comprobación con todos los objetos de esa familia y seleccionar el mejor de ellos.
- Conclusión: se ha decidido dejar esta opción activada por defecto, pero con posibilidad de desactivarla desde el menú de configuración.

### C.1.9. Transcripción de la generación de histogramas a C++

- Contexto: segunda iteración, tras implementar la bolsa de palabras.
- Objetivo: utilizar el generador de histogramas implementado en el paquete *commons* implicaba unos tiempos de generación aproximadamente de 45 segundos. Para tratar de reducir este tiempo, se transcribió el código de Java anteriormente implementado a una versión en C++, accesible a través de JNI.
- Requiere: haber implementado el algoritmo BoVW y programar un nuevo generador de histogramas.
- Resultado: reducir el tiempo de generación de histogramas de 45 segundos a 28.
- Conclusión: mantener esta opción.

### C.1.10. Cálculo manual de la distancia de Hamming

- Contexto: segunda iteración, tras implementar la bolsa de palabras.
- Objetivo: los descriptores de ORB utilizan 256 bits, que openCV devuelve como un *array* de 32 bytes. Por tanto, para obtener la distancia de Hamming entre dos descriptores será necesario calcular y sumar las distancias de Hamming entre cada par de bytes de ambos vectores. Para cada par de bytes, esta operación es posible realizarla en dos fases: aplicar la operación XOR[1] bit a bit y contar el número de bits con valor 1 en el resultado. Para esta última operación se implementó inicialmente el algoritmo de

[1] Del inglés *Exclusive OR*. Operación binaria cuyo valor es 1 si sus dos operandos tienen valores diferentes.

Kernighan's[6], pero resultó más eficiente el uso de la instrucción proporcionada por GCC llamada '*__builtin_popcount*', que obtiene directamente el valor. Además, tanto esta función como la operación XOR admiten valores enteros, por lo que en vez de convertir cada byte a entero se optó por concatenar los bytes de cada descriptor de 4 en 4 para formar enteros. De esta manera, cada descriptor pasará a estar compuesto por un *array* de 8 enteros y, por tanto, serán necesarias únicamente 8 operaciones para compararlos (en vez de 32 si fueran bytes). Este proceso se encuentra representado en la figura C.1.

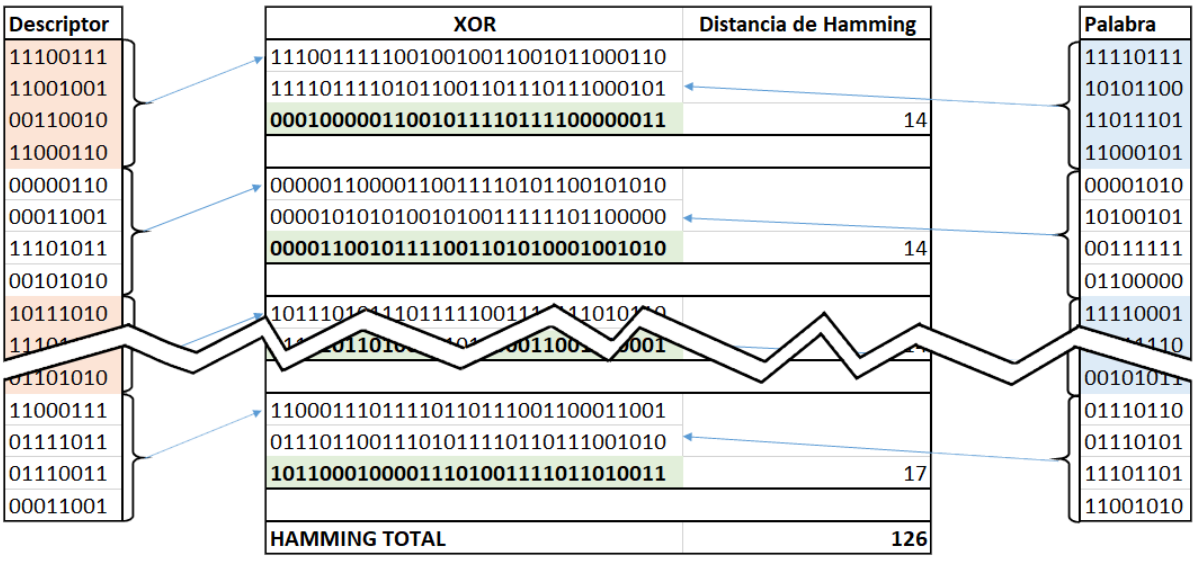



Figura C.1: Cálculo de la distancia de Hamming entre descriptores.

- Requiere: haber implementado el algoritmo BoVW y modificar el generador de histogramas.
- Resultado: reducir el tiempo de generación de histogramas de 28 segundos a 2.
- Conclusión: mantener esta opción.

### C.1.11. Uso de *multithreading*

- Contexto: segunda iteración, tras implementar la bolsa de palabras.
- Objetivo: paralelizar la generación de histogramas y la validación geométrica para aprovechar las capacidades del hardware. Además permitirá el reconocimiento múltiple en la imagen.
- Requiere: haber implementado el algoritmo BoVW, modificar el generador de histogramas y permitir múltples objetos en el algoritmo de *tracking*.
- Resultado: reducir el tiempo de generación de histogramas de 2 segundos a 1 utilizando 6 *threads*, número obtenido al evaluar el sistema con diferentes números de *threads*. Este resultado queda reflejado en la figura C.2.

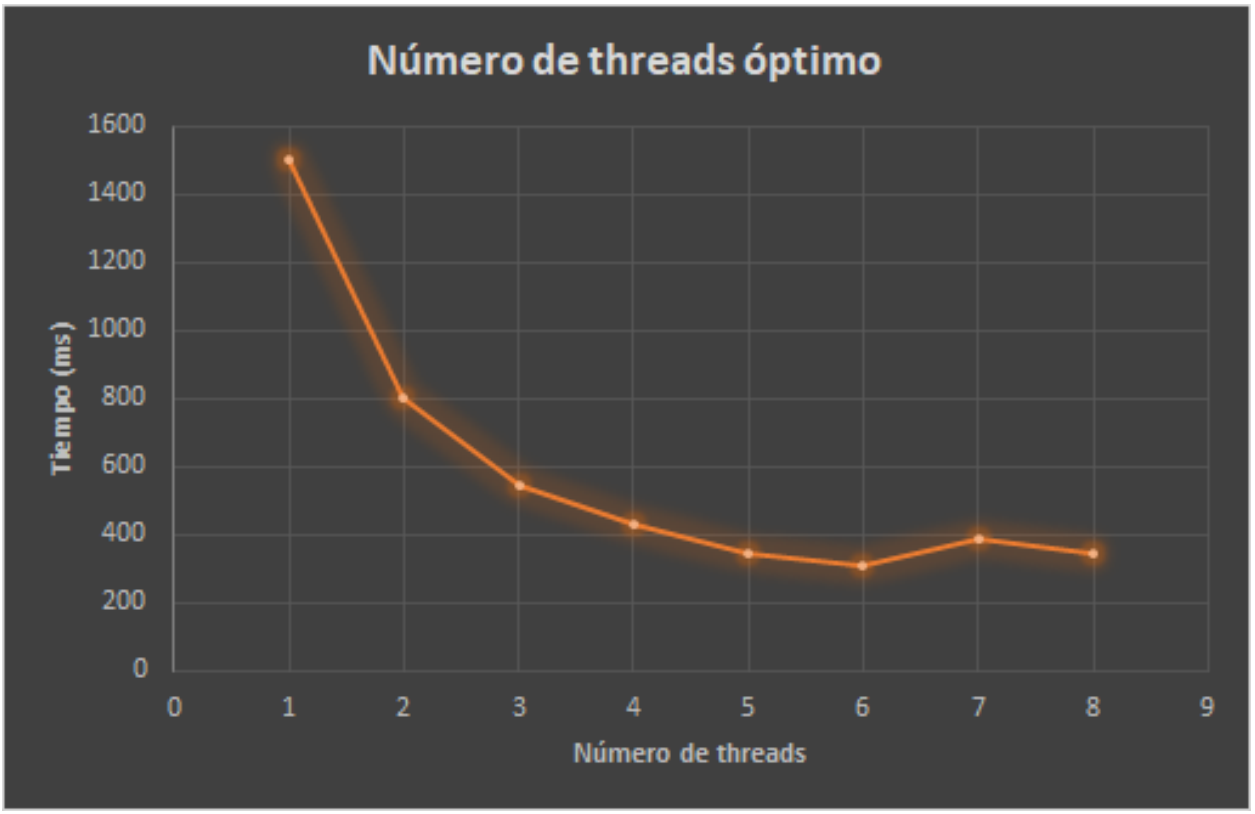



Figura C.2: Número de threads óptimo.

Para la validación geométrica se lanzarán tantos *threads* como objetos candidatos aparezcan en la imagen (máximo 8, por defecto). De esta forma, según vayan finalizando los *threads*, si el resultado es exitoso, se agregarán de manera asíncrona como objetos reconocidos al algoritmo de *tracking*.

- Conclusión: mantener la generación de histogramas utilizando 6 *threads* y permitir al usuario seleccionar la opción de reconocimiento múltiple, pues en ocasiones puede entorpecer los resultados.

## C.2. Cambios de estrategia

A la vista de no poder mejorar el rendimiento con la estrategia inicial (obtener en cada *frame* los puntos ORB y realizar la búsqueda de los objetos), se optó por aplicar distintas estrategias:

| **Opción** | **Uso en MIHex** |
|---|---|
| Explorar ARToolkit | Descartada |
| Utilizar YOLO | Descartada |
| Reconocimiento en segundo plano | Utilizada |
| Utilizar BoVW | Utilizada |

Tabla C.2: Cambios de estrategia planteados para el reconocimiento

Como en el apartado anterior, se explican a continuación estas estrategias planteando su objetivo, sus requerimientos, sus resultados y la conclusión obtenida.

### C.2.1. Explorar ARToolkit

- Objetivo: utilizar APIs existentes de RA puede servir para realizar la tarea de realidad aumentada de manera eficiente y sencilla. Se decidió probar ARToolkit por ser gratuita y más sencilla de incorporar, frente a Vuforia (que sólo esta disponible de forma gratuita como *plugin* de Unity) o Kudan (que resultó más complicada de incorporar).
- Requiere: descargar ARToolkit e incorporarlo, generar *targets* a partir de imágenes.
- Resultado: tras probar ARToolkit con los *targets* por defecto y explorar el código fuente se comprobó que el funcionamiento para localizar los objetos en la imagen consiste en dos partes: en la primera trata de reconocer a qué objeto corresponde el *target* de la imagen y localizarlo en la imagen; en la segunda mantendrá las coordenadas reconocidas del objeto en la imagen mediante *tracking* de la imagen (a partir de puntos ORB). De las dos partes únicamente funciona a tiempo real la segunda, mientras que la primera mantiene un retraso considerable con muy pocos *targets* a buscar (únicamente 3). La interfaz de ARToolkit resultó además bastante limitada, y los *targets* a utilizar requieren cumplir bastantes condiciones.
- Conclusión: ARToolkit funciona bien sobre todo en la parte de *tracking*, siendo esta bastante rápida y precisa, pero pierde en la parte de reconocimiento. Además, su uso es bastante limitado y no resultaría sencillo integrarlo con reconocimiento de objetos no planos.

### C.2.2. Utilizar YOLO

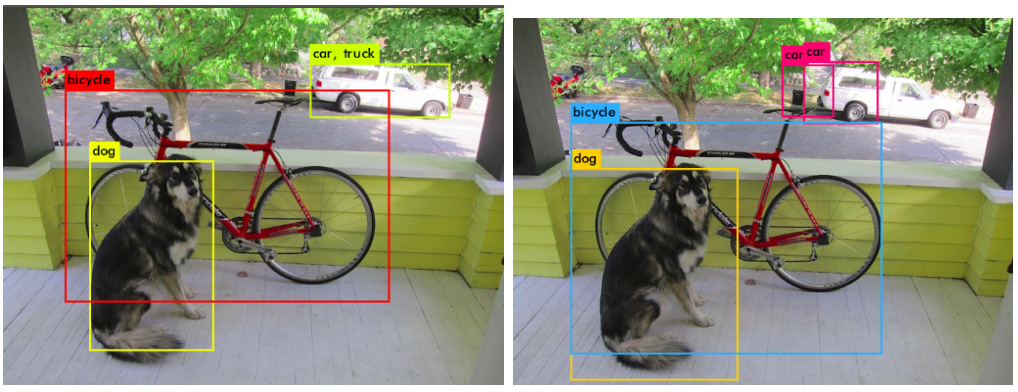
- Objetivo: YOLO es uno de los mejores reconocedores de objetos (situado en el estado del arte), capaz de realizar cálculos en tiempo real. Como resultado del reconocimiento es capaz de señalar qué objetos hay en la imagen y definir donde se situan gracias a un *bounding box*. Sin embargo no es capaz de estimar la perspectiva del objeto respecto a la cámara, por lo que habría que implementar la parte de RA. Para esta labor no sería necesario buscar puntos ORB en toda la imagen, solo dentro de los marcos donde se encuentran los objetos ya reconocidos. Además, dichos puntos únicamente se necesitaran emparejar con los del objeto reconocido (en vez de volver a buscarlos en toda la colección), por lo que se obtendría otra mejora de rendimiento en este aspecto. Sin embargo, la única implementación que se encontró compatible con Android es a través de la librería *tensorflow* de Google, por lo que además del entrenador por defecto que proporciona YOLO (*darknet*) es necesario un transcriptor del formato *darknet* a *tensorflow* llamado *darkflow*. Además, este transcriptor también permite entrenar la red neuronal.
- Requiere: descargar implementación de tensorflow para Android (interfaz Java + sistema), descargar darknet (entrenador + pesos iniciales) descargar *darkflow* (*darknet* + *tensorflow*) + dependencias (*tensorflow*, python3, opencv2, pip, etc), descargar etiquetador de imágenes (o implementar uno mejor), etiquetar imágenes y entrenar.

- Resultado: en la aplicación de ejemplo de Android, a través de *tensorflow*, se obtienen resultados a 2fps por segundo, por lo que no se puede aplicar al reconocimiento en tiempo real. Al igual que ARToolkit adopta una estrategia de reconocer primero y hacer *tracking* después, solo que en este caso el reconocimiento se realiza en segundo plano (por lo que es transparente al usuario) y constantemente, actualizando así la información y coordenadas a mantener mediante *tracking*.

  A continuación se realizó el siguiente set de pruebas:

  - Reconocimiento dentro de la *app* del MIH con los ficheros de la implementación encontrada de YOLO para Android. Los objetos entrenados por la red por defecto eran encontrados correctamente, pero con un coste en tiempo alto.

  - Reconocimiento de YOLO y TINY YOLO (una versión más ligera de YOLO pero menos precisa) con *darknet* y red por defecto. Se realizaron pruebas sobre el set de imágenes por defecto de YOLO. Los resultados fueron satisfactorios en precisión, pero insuficientes en tiempo, tal y como se aprecia en la figura C.3.



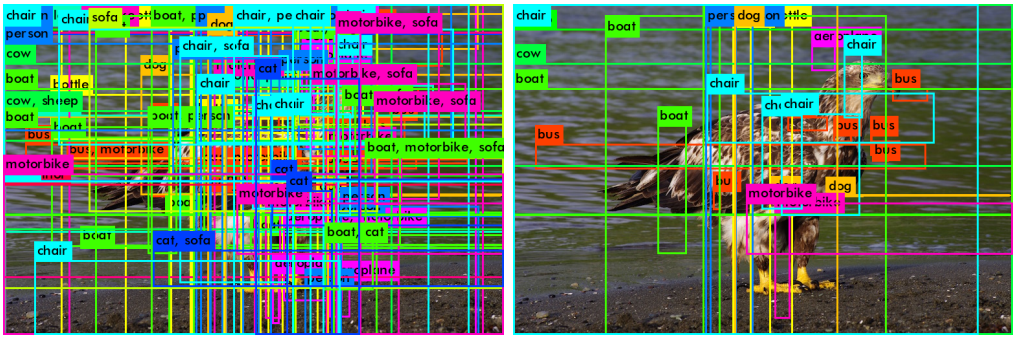

(a) Resultado de YOLO (53.14 segundos) (b) Resultado de TINY YOLO (6.3 segundos)

Figura C.3: Reconstrucción de la imagen con los centroides iniciales

  - Entrenamiento con datos del VOC[2] desde *darknet*. Se siguió el tutorial proporcionado por *darknet* para realizar dicho entrenamiento, pero el error cometido nunca llegaba a bajar lo suficiente como para darlo por satisfactorio. Se pueden apreciar los resultados del entrenamiento realizado durante 4 días en la figura C.4, en la que se muestran unas imágenes sobre las que se ha aplicado la red, pero con *thresholds*[3] diferentes.

[2] Es un popular banco de datos que cuenta con imágenes etiquetadas cuyo propósito es ser utilizado como banco de pruebas para tareas de clasificación.

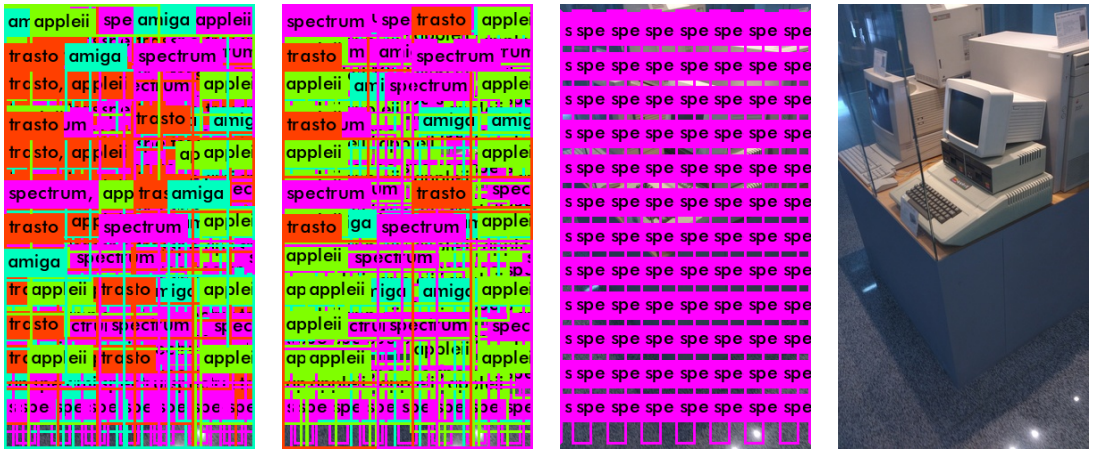
[3] Umbral de probabilidad a partir del cual se considera que el objeto ha sido reconocido.



(a) Resultado con *threshold*=0.25 (b) Resultado con *threshold*=0.50

Figura C.4: Resultado de entrenamiento con *darknet* de los datos del VOC. Tiempo aproximado: 4 días

- Entrenamiento con imágenes del MIH desde *darknet*. Se siguieron las mismas pautas que en el punto anterior, poniendo mayor cuidado en la configuración. Para este apartado se desarrolló además un programa en Java para etiquetar imágenes y exportar dichas etiquetas en el formato que exige *darknet*. Los resultados fueron igual de insatisfactorios (ver figura C.5).



(a) Resultado con *threshold*=0.12 (b) Resultado con *threshold*=0.125 (c) Resultado con *threshold*=0.126 (d) Resultado con *threshold*=0.13

Figura C.5: Resultado de entrenamiento con darknet de los datos del MIH. Tiempo aproximado: 4 días

- Entrenamiento con datos del VOC desde *darkflow*. Tras 3 días y medio el error (loss) no acaba de converger, tal y como se ve en la figura C.6.

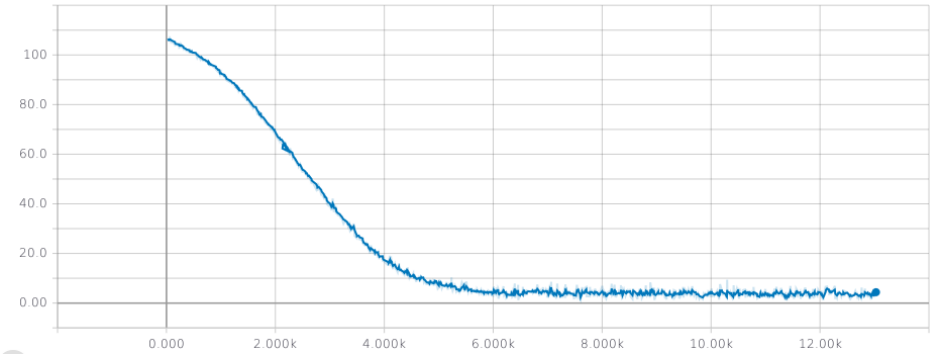



Figura C.6: Error cometido durante el entrenamiento de los datos del VOC con *darkflow*. Gráfica obtenida mediante la herramienta *Tensorboard*.

- Conclusión: debido a la imposibilidad de entrenar correctamente la red de YOLO se decidió descartar esta opción, aunque se sigue considerando su uso interesante, sobre todo para el reconocimiento de objetos no planos. Podría incorporarse al proyecto en un futuro, pero requiere un estudio más a fondo del funcionamiento de YOLO.

### C.2.3. Reconocimiento en segundo plano

- Objetivo: consiste en realizar el reconocimiento inicial en segundo plano, tal y como hace YOLO. Mientras tanto se realizará un *tracking* de la imagen desde la que se está realizando el reconocimiento. De esta forma, cuando acabe el reconocimiento se incorporarán las coordenadas de los objetos reconocidos en la imagen y se actualizarán a la posición de la imagen actual. Posteriormente se mantendrán dichas coordenadas en la imagen mediante *tracking*, bien desde la imagen original o desde la imagen anterior. De esta forma, el usuario final apreciará el movimiento de la imagen en tiempo real, a pesar de que la tarea de reconocimiento sea más costosa. Esto no supone una mejora en el rendimiento como tal, por lo que puede considerarse como una pseudo-solución. El reconocimiento podrá realizarse tanto automáticamente de forma secuencial (cuando acabe de reconocer la anterior imagen buscará nuevamente sobre la actual) como de forma manual (pulsando sobre el FAB). Esta última opción puede venir acompañada de una animación de procesamiento sobre la imagen (quizás via *renderscript*/*filterscript*). Por último, si los resultados de YOLO sobre el set de prueba del MIH resultan exitosos, podrá incorporarse a esta opción.

- Requiere: modificar el código propio para adaptar esta funcionalidad y añadir *tracking*. Realizar animación en *renderscript* para reconocimiento manual.

- Resultado: se incorporó únicamente la opción manual, pues el *tracking* funcionaba razonablemente bien y no es necesario buscar de nuevo los objetos en la imagen. La animación en pantalla y el reconocimiento de objetos en segundo plano proporcionan una experiencia más fluida de la aplicación, y no se aprecian paros ni retrasos en la imagen de la cámara.

- Conclusión: utilizar esta opción, ya que ofrecía una experiencia cómoda al usuario.

### C.2.4. Utilizar BoVW

- Objetivo: implementar el algoritmo de BoVW especificado en la sección 4.
- Requiere: implementar el algoritmo y modificar todo el sistema: bases de datos, MIHDatabaseManager, estructuras de datos...
- Resultado: esta ha sido la optimización con más carga de trabajo, ya que ha afectado a todo el sistema. Sin embargo, los resultados merecido el esfuerzo. A diferencia del algoritmo anterior, este es capaz de obtener los objetos más probables en un tiempo relativamente constante, teniendo poco impacto la cantidad de objetos del museo almacenados, por lo que es mucho más escalable.
- Conclusión: se ha seleccionado esta versión como la definitiva por ser la que mejores resultados ofrecía. Es la versión a la que se hace alusión a lo largo del trabajo.

# Anexo D

# Manual de usuario de MIHex

Este anexo supone una guía básica para el usuario de la aplicación MIHex, donde se muestran únicamente las funcionalidades básicas de la aplicación.

La pantalla principal de la aplicación se muestra en la figura D.1, así como las opciones y menús que se pueden seleccionar en ella. Mediante el botón de "iniciar reconocimiento" el sistema comienza a buscar los objetos que se encuentren en la imagen en ese mismo momento, comenzando a la par la animación de procesado. Tras finalizar el proceso, los objetos reconocidos aparecen enmarcados por un rectángulo rojo. Pulsando sobre un rectángulo es posible acceder a la pantalla de detalles del objeto seleccionado.

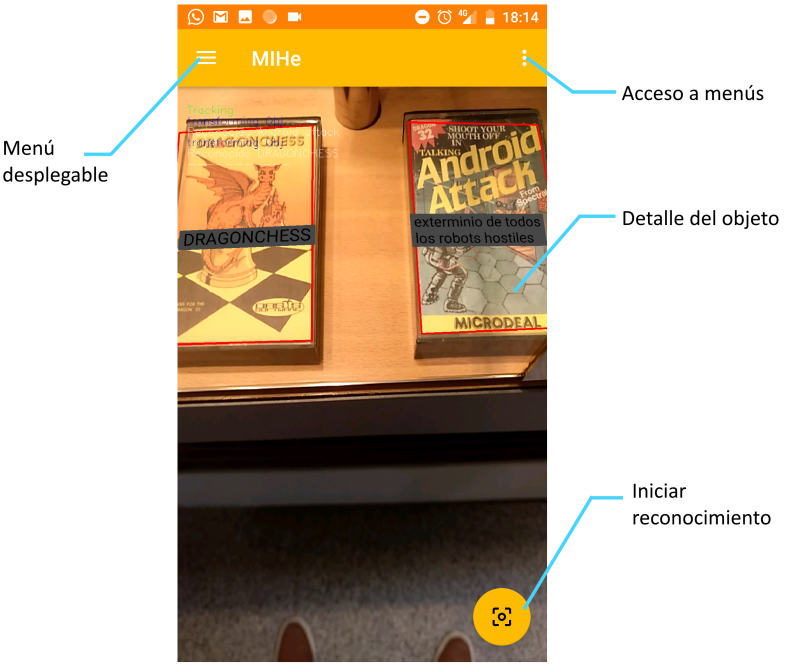
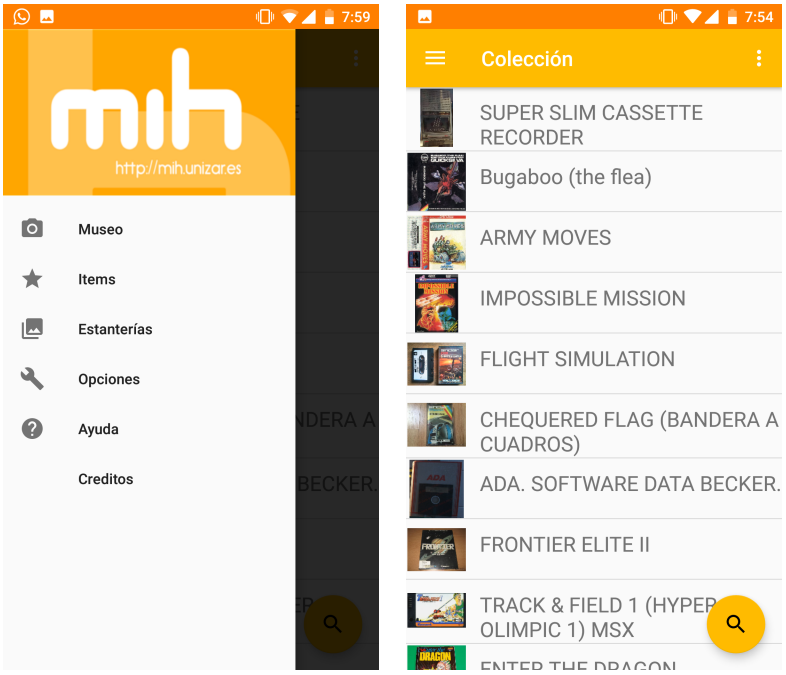



Figura D.1: Pantalla principal de la aplicación.

El menú lateral se muestra en la figura D.2a. Este menú es accesible deslizando el dedo por la pantalla de izquierda a derecha o pulsando sobre el botón indicado en la figura D.1. Al seleccionar la opción "Items" se accede a la la pantalla de listado de objetos (ver figura D.3b).

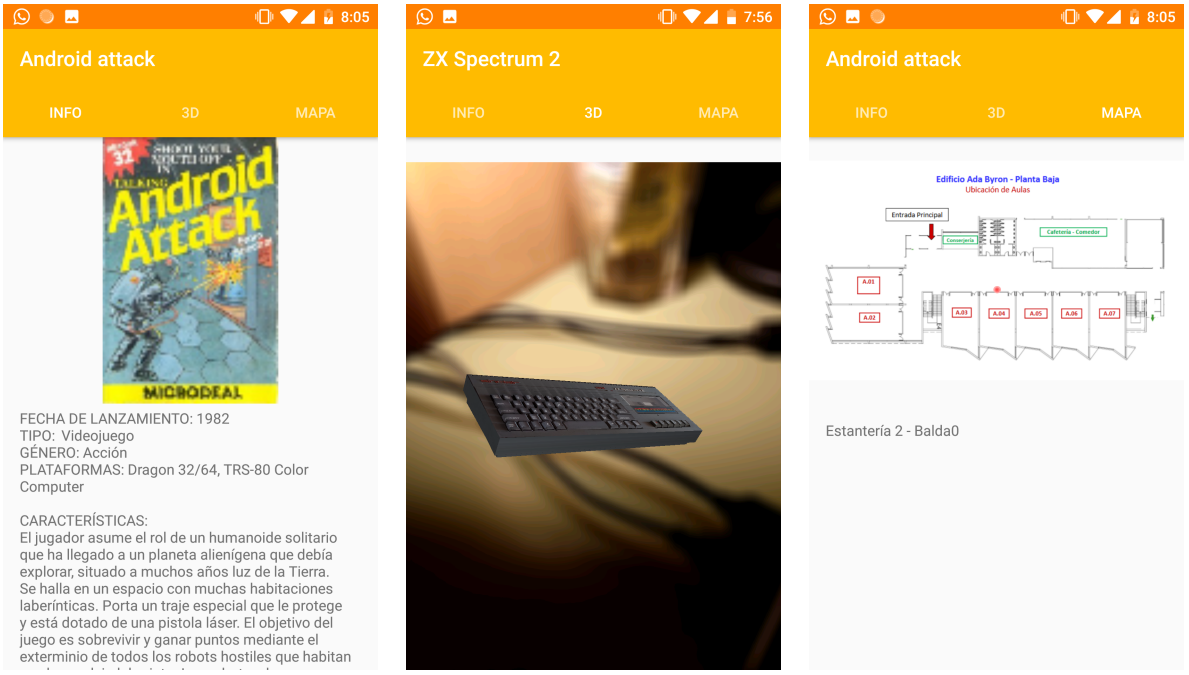
La pantalla de detalles de un objetos se muestra en la figura D.3. Esta pantalla es accesible desde la pantalla de lista de objetos o desde la pantalla principal, al pulsar sobre un objeto reconocido.




(a) Pantalla de imagen y descripción

(b) Pantalla de lista de objetos

Figura D.2: Pantallas intermediarias.




(a) Pantalla de imagen y descripción

(b) Pantalla de modelo en 3D

(c) Pantalla de localización en el museo

Figura D.3: Pantalla de lista de objetos.

# Anexo E

# Manual de usuario de MIHDatabaseManager

Este anexo supone una guía para el administrador de la herramienta MIHDatabaseManager. Cada una de las siguientes secciones explicará como llevara cabo las 4 principales tareas que el administrador puede requerir de la herramienta: gestión de opciones generales, de objetos, de estanterísa/baldas o de la bolsa de palabras.

## E.1. Gestión de objetos

En la pantalla principal de la herramienta, mostrada en la figura E.1, aparece la pestaña de objetos seleccionada por defecto. Desde esta ventana puedes seleccionar un objetos existente en el listado y, en el panel de la derecha, la operación de crear un objeto nuevo o editar/borrar el objeto seleccionado. Con las opciones de "nuevo" y "editar" se abrirá la ventana de edición de objetos, mostrada en la figura E.2. En ella se podrán realizar las siguientes acciones:

- Seleccionar la pestaña de "reconocimiento" (opción por defecto) para mostrar el panel de reconocimiento (ver sección E.1.1).
- Seleccionar la pestaña de "gráficos" para mostrar el panel de gráficos (ver sección E.1.2)
- Seleccionar la pestaña de "realidad aumentada" para mostrar el panel de realidad aumentada (ver sección E.1.3)
- Modificar el nombre del objeto
- Especificar una imagen de vista previa pulsando sobre el botón que está a la izquierda del nombre.
- Especificar una descripción. Si se desea especificar una descripción concreta para mostrar en la etiqueta de RA, se podrá especificar, como solución temporal, mediante el separador "##". De esta forma, el texto que figure antes de este separador será el destinado a la etiqueta, mientras que el texto posterior será el mostrado en la pantalla de detalles del objeto de MIHex.
- Asignar una o varias categorías. Al seleccionar el botón de añadir (cruz verde), se abrirá un desplegable que permitirá seleccionar una categoría existente o crear una nueva.

- Añadir una nueva instancia (opción todavía no implementada) desde esta pantalla. Es posible generar instancias desde la pantalla de baldas (ver sección E.2.1).
- Guardar o descartar los cambios.

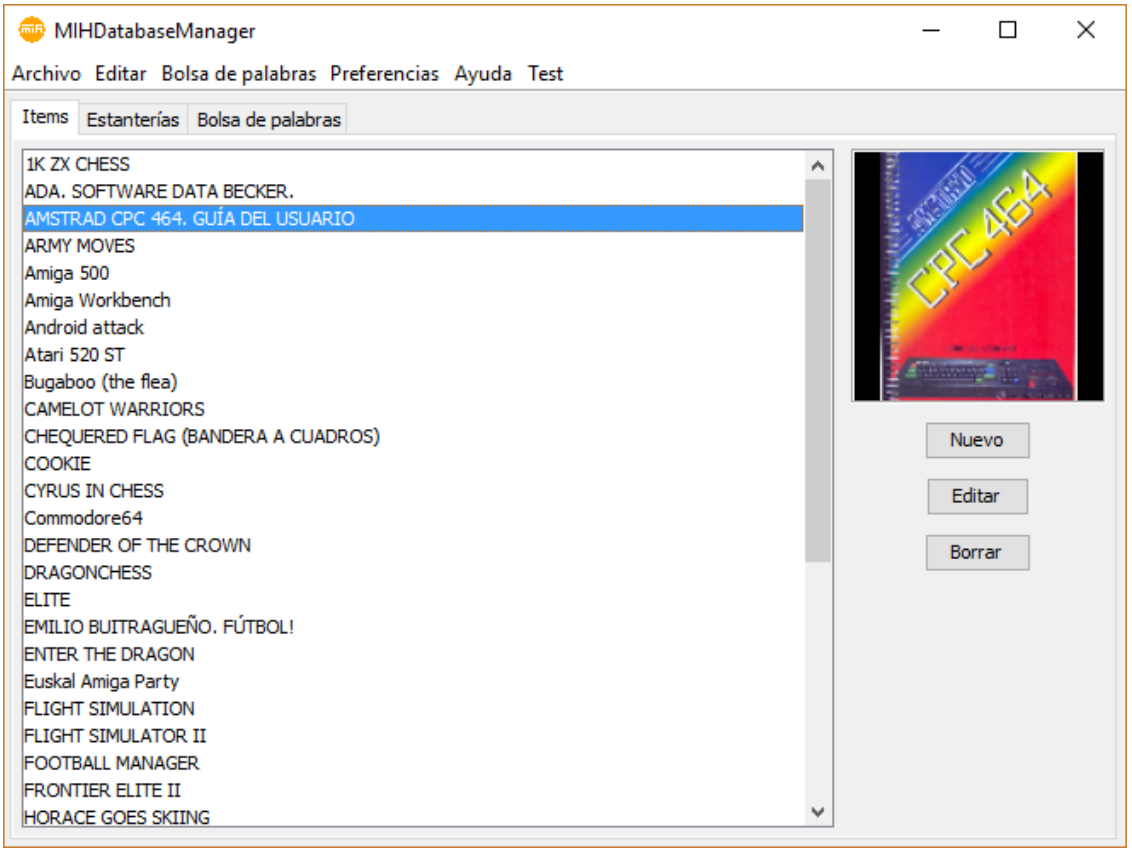



Figura E.1: Pantalla principal de la herramienta, mostrando los objetos existentes.

### E.1.1. Panel de reconocimiento

Desde este panel, mostrado en la figura E.2, es posible definir la imagen de detección de un item, estantería o balda, sus imágenes de reconocimiento y sus imágenes de test. En la pestaña de “Detección” es posible realizar las siguientes acciones.

- Añadir una nueva imagen de detección mediante el botón “cargar imagen”
- Enmarcar al objeto en la imagen. Para ello basta con arrastrar con el ratón los vértices del marco, situadas por defecto en las esquinas de la propia imagen.
- Corregir el ancho y el alto del marco de detección que debería tener el objeto visto perpendicularmente mediante el botón ”Dimensiones”
- Analizar la imagen de detección comparándola con las imágenes de reconocimiento. Esto también muestra los puntos de interés extraidos de la imagen.

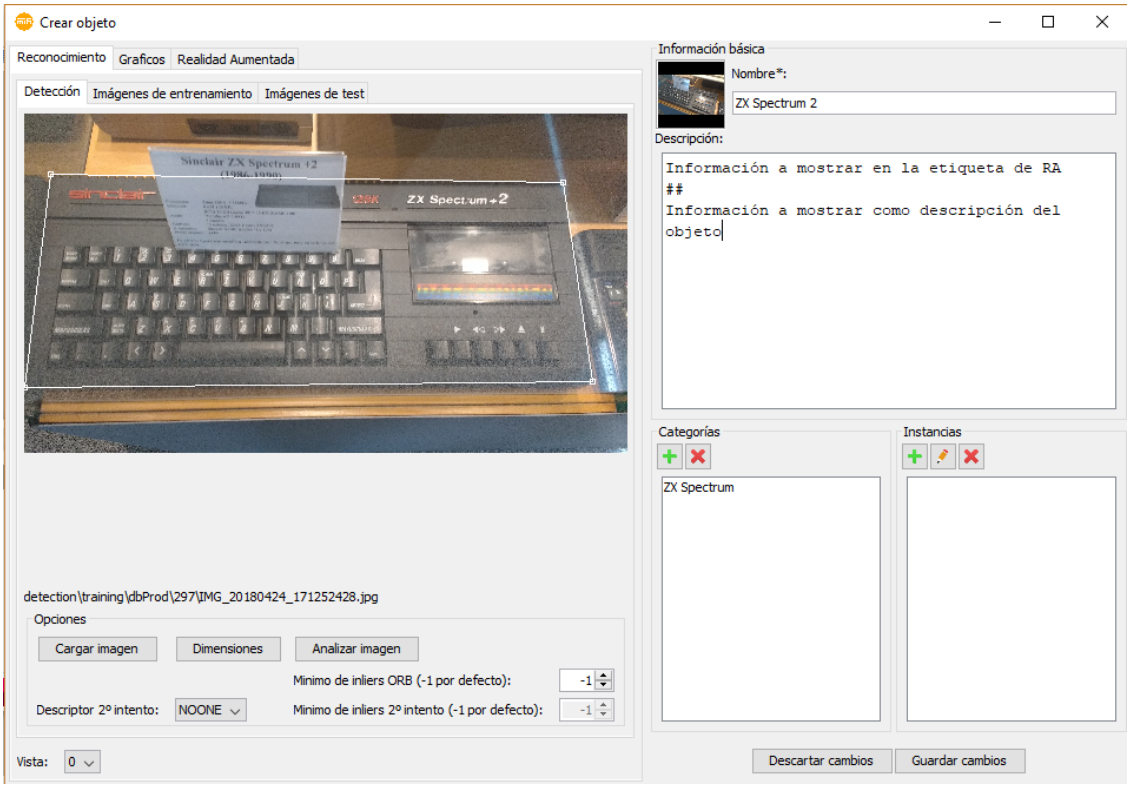



Figura E.2: Pantalla de edición de objetos.

- Definir un tipo de descriptor para un segundo intento, así como los umbrales de inliers para este descriptor y para ORB. Sin embargo, estas opciones no se llegaron a contemplar en la aplicación MIHe.

### E.1.2. Panel de gráficos

Desde el panel de gráficos, mostrado en la figura E.3, es posible definir el modelo del objeto y su textura. Aunque la interfaz admite múltiples texturas y tipos de texturas, no se llegaron a implementar, por lo que estas opciones no tienen efecto alguno. El sistema tomará únicamente la primera textura de tipo difuso.

### E.1.3. Panel de realidad aumentada

Desde el panel de RA, mostrado en la figura E.4, es posible aplicarle unas transformaciones de translación, rotación y escala respecto a la imagen que utilizará para detectar. También se podrá comprobar el resultado mediante la opción “probar RA”, que muestra la ventana que aparece en la figura E.5. A través de un desplegable, el usuario puede seleccionar cualquiera de las imágenes de reconocimiento e imágenes de test del objeto. Cuando esto ocurre, se localiza el objeto en la imagen seleccionada y se muestra sobre ella el modelo 3D del objeto aplicándole, además de las transformaciones definidas anteriormente, las mismas transformaciones utilizadas en la *app* para producir la realidad aumentada. Esto supone una vista previa del resultado que podrá apreciarse en la aplicación.

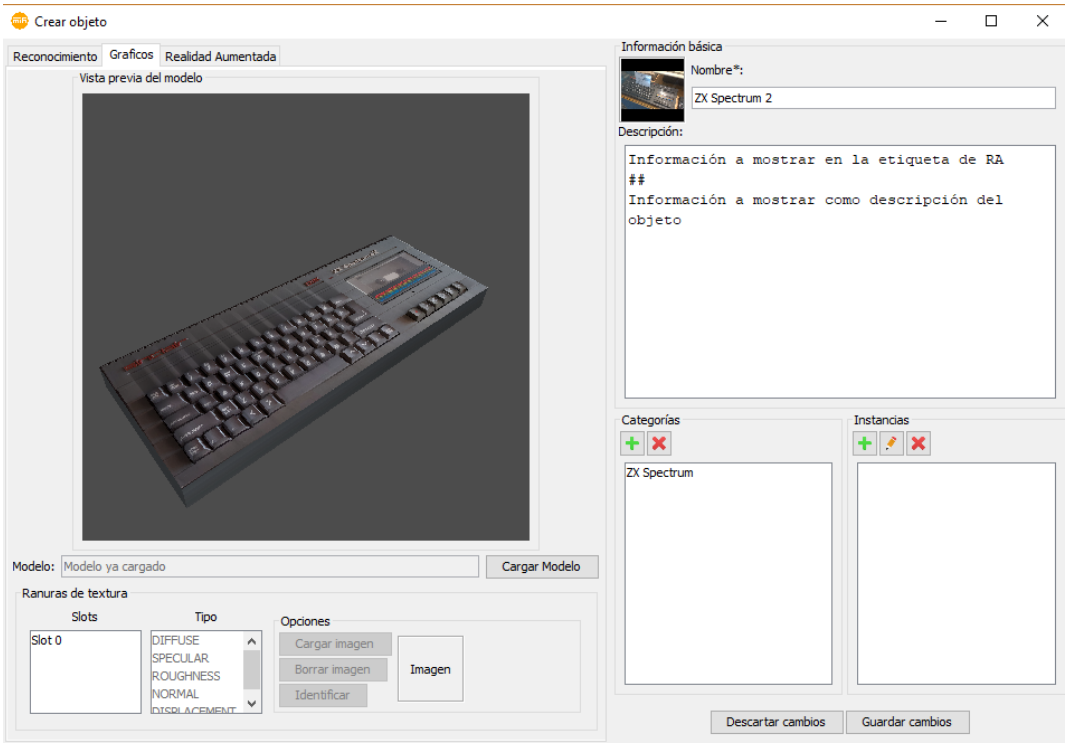



Figura E.3: Panel de gráficos.

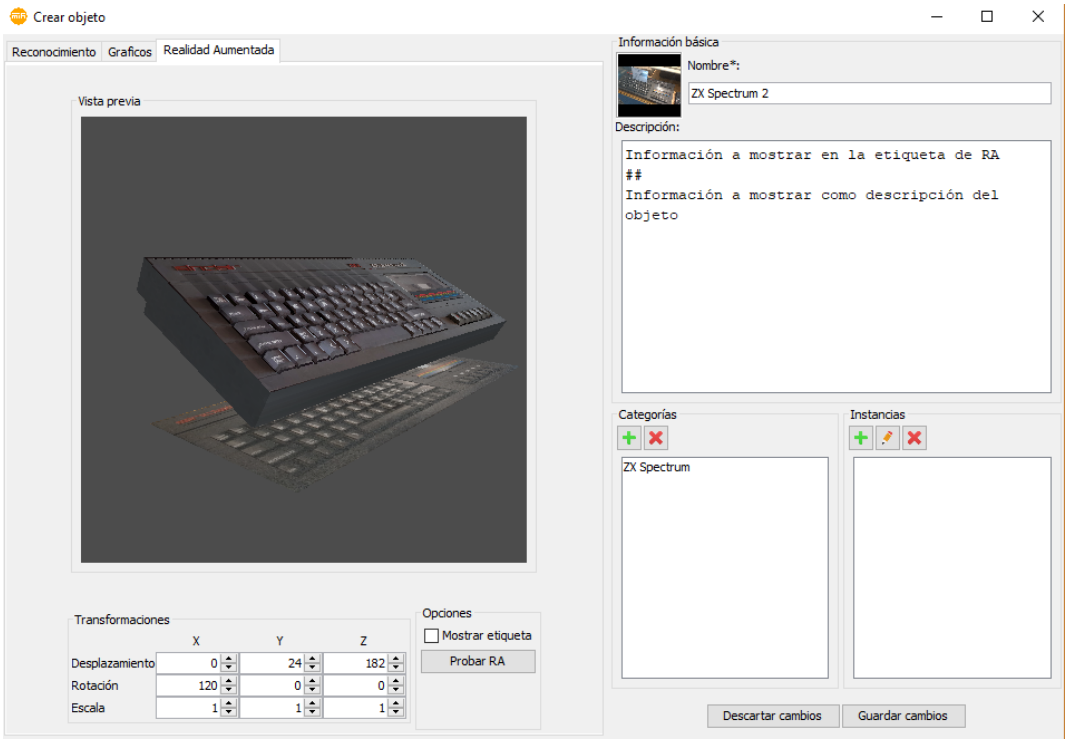



Figura E.4: Panel de realidad aumentada.


Figura E.5: Ventana de test de realidad aumentada.

## E.2. Gestión de estanterías y baldas

En la pantalla principal de la herramienta, al seleccionar la pestaña de "Estanterías", se muestra la vista de la figura E.6. Desde esta ventana es posible seleccionar una estantería existente en el listado y, en el panel de la derecha, la operación de crear un objeto nuevo o editar/borrar la estantería seleccionada. Con las opciones de "nuevo" y "editar" se abrirá la ventana de edición de estanterías, mostrada en la figura E.7. En ella se podrán realizar las siguientes acciones:

- Modificar el nombre de la estantería
- Escoger entre vitrina real o virtual. Si se selecciona la opción "virtual", la estantería se considerará como un objeto reconocible, por lo que aparecerán los mismos paneles de reconocimiento, gráficos y realidad aumentada que en la pantalla de objetos (ver secciones E.1.1, E.1.2 y E.1.3 respectivamente).
- Seleccionar su localización en el mapa del museo.
- Asignar una o varias categorías. Al seleccionar el botón de añadir (cruz verde), se abrirá un desplegable que permitirá seleccionar una categoría existente o crear una nueva.
- Añadir, modificar o eliminar baldas. Al seleccionar "Añadir" o "Editar" se abrirá la ventana de edición de baldas (ver sección E.2.1).
- Guardar o descartar los cambios

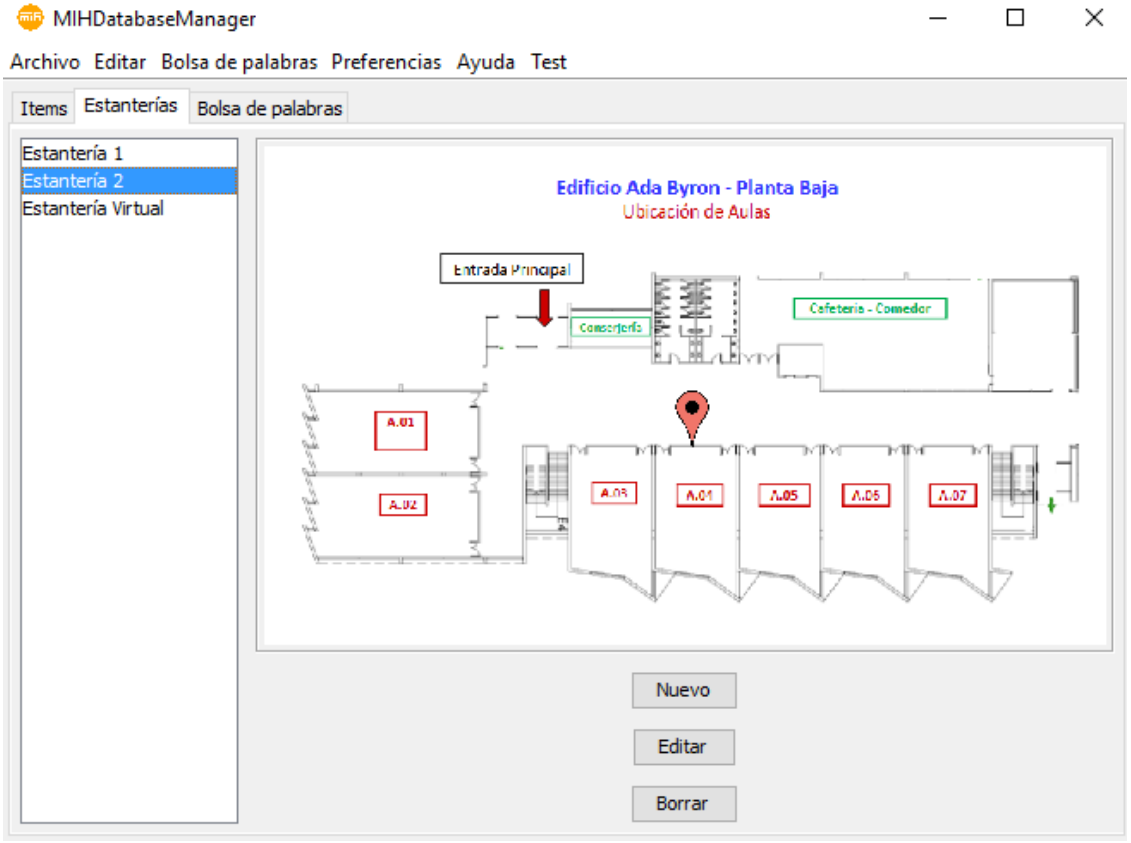



Figura E.6: Pantalla de estanterías existentes.

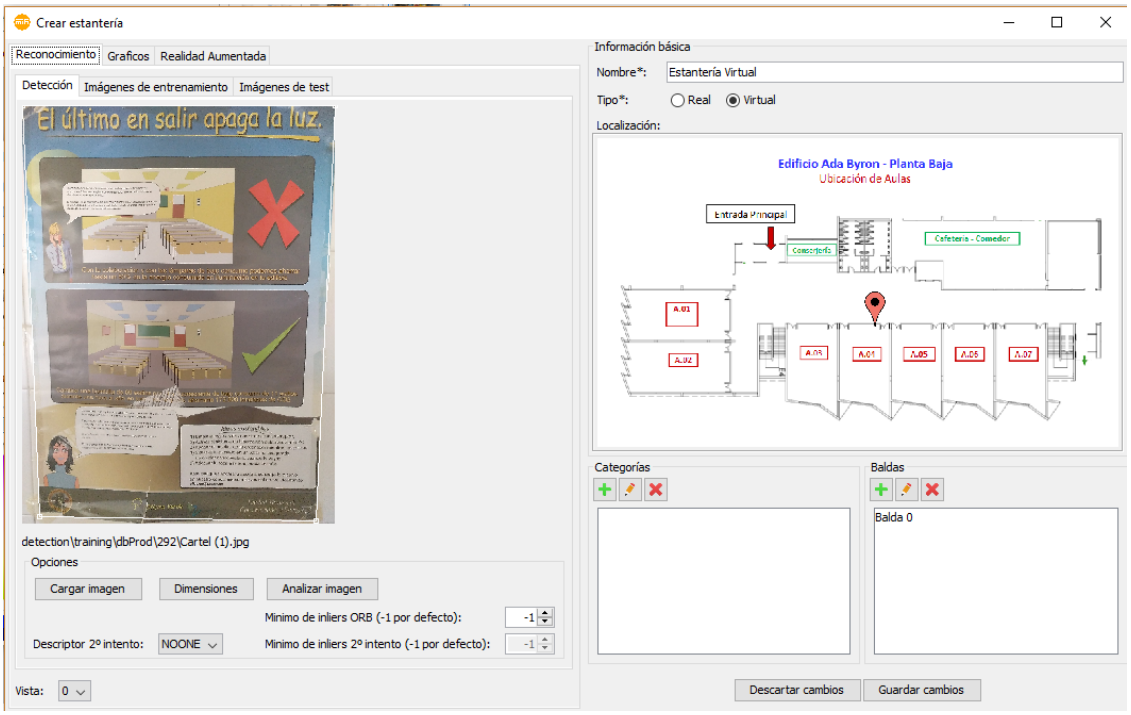



Figura E.7: Pantalla de edición de estanterías.

### E.2.1. Baldas

En función de si la estantería a la que pertenecen es real o virtual, la ventana de creación/edición de baldas contendrá opciones distintas.

En el caso de las baldas de estanterías reales se mostrará una pestaña con las mismas opciones de reconocimiento que los objetos (ver sección E.1.1) y otra pestaña en la que se podrán enmarcar sobre la imagen de la balda los items reales que en ella aparecen, tal y como se muestra en la figura E.8.

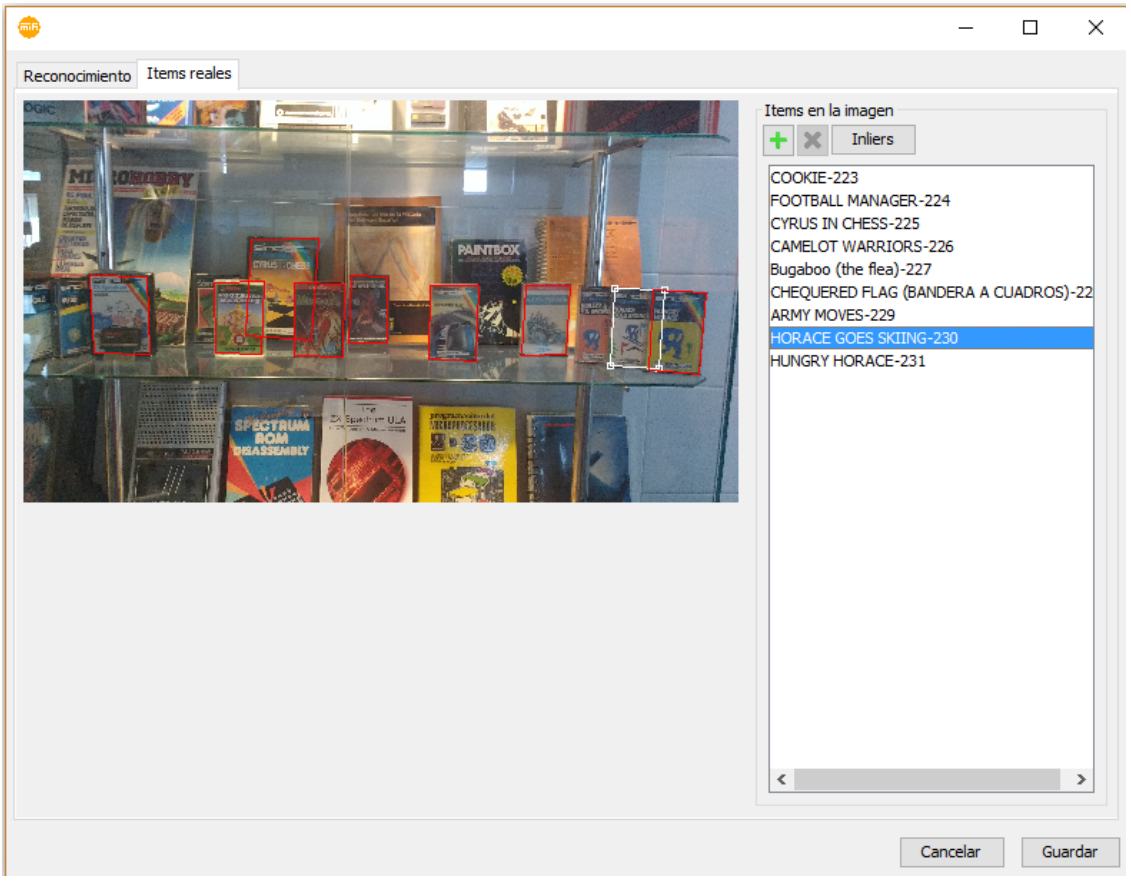



Figura E.8: Pantalla de gestión de items reales en estanterías reales.

En el caso de las baldas de estanterías virtuales se mostrará una única pestaña en la que se podrán colocar sobre el modelo 3D de la estantería los items virtuales que se deseen, tal y como se muestra en la figura E.9.

En ambas posibilidades se mostrará un panel lateral a la derecha desde el que poder añadir/borrar items en la balda, seleccionarlos y guardar los cambios.

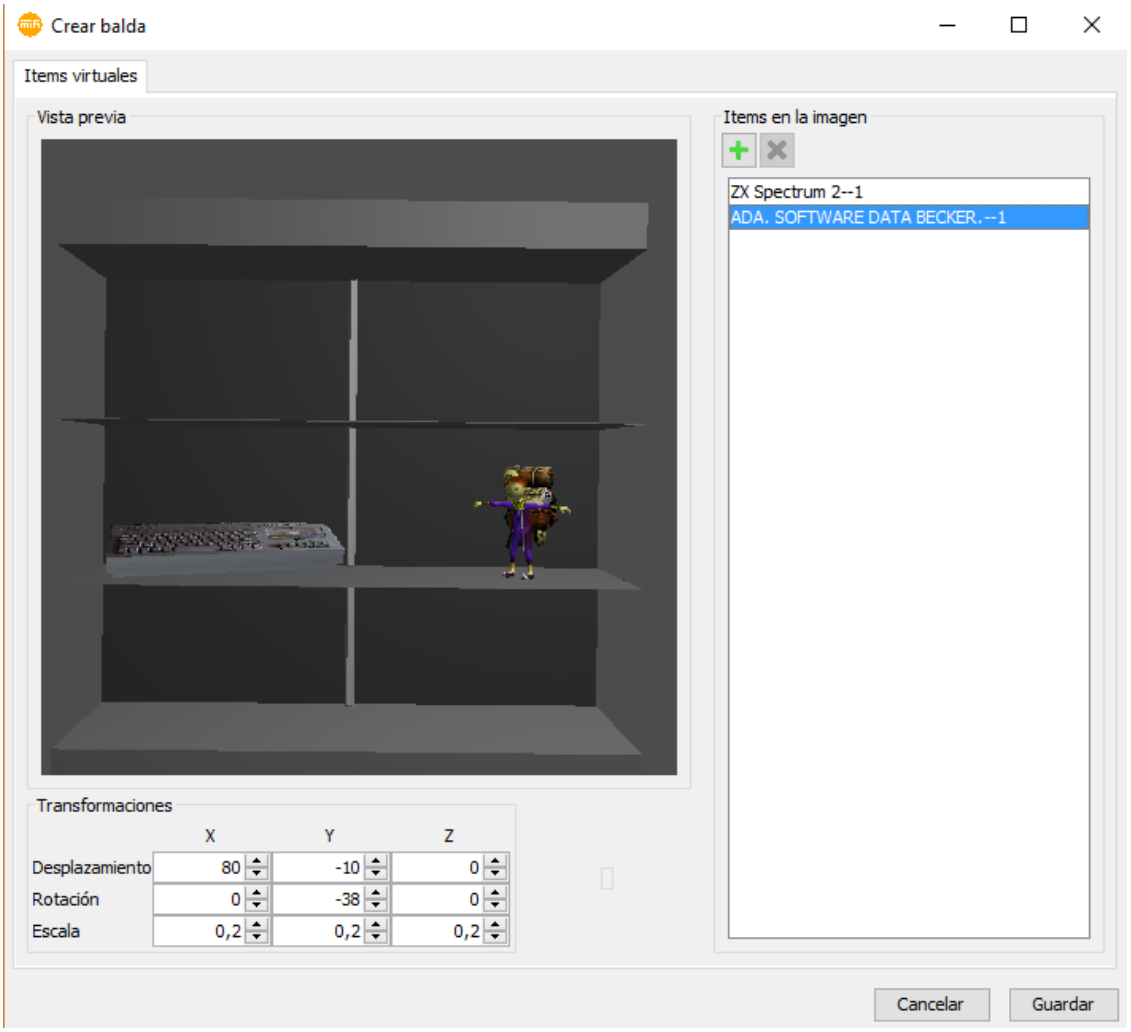



Figura E.9: Pantalla de gestión de items virtuales en estanterías virtuales.

## E.3. Bolsa de palabras

En la pantalla principal de la herramienta, al seleccionar la pestaña de "Bolsa de palabras", se muestra la vista de la figura E.10. Desde esta ventana es posible generar un nuevo vocabulario de palabras, generar los histogramas de los objetos y probar el reconocimiento contra las imágenes de prueba. Éstas se pueden añadir desde los paneles de reconocimiento (ver sección E.1.1) o desde el panel lateral derecho.

## E.4. Opciones generales

Desde el menú superior de la pantalla principal, seleccionando la opción "Archivo", es posible realizar algunas acciones de caracter general:

- Crear/borrar la base de datos
- Guardar los cambios realizados durante la sesión en la base de datos

- Poblar la base de datos desde un fichero
- Exportar el contenido de la base de datos a un fichero
- Comprobar que el fichero exportado es correcto
- Probar el reconocimiento y realidad aumentada a través de una cámara-web

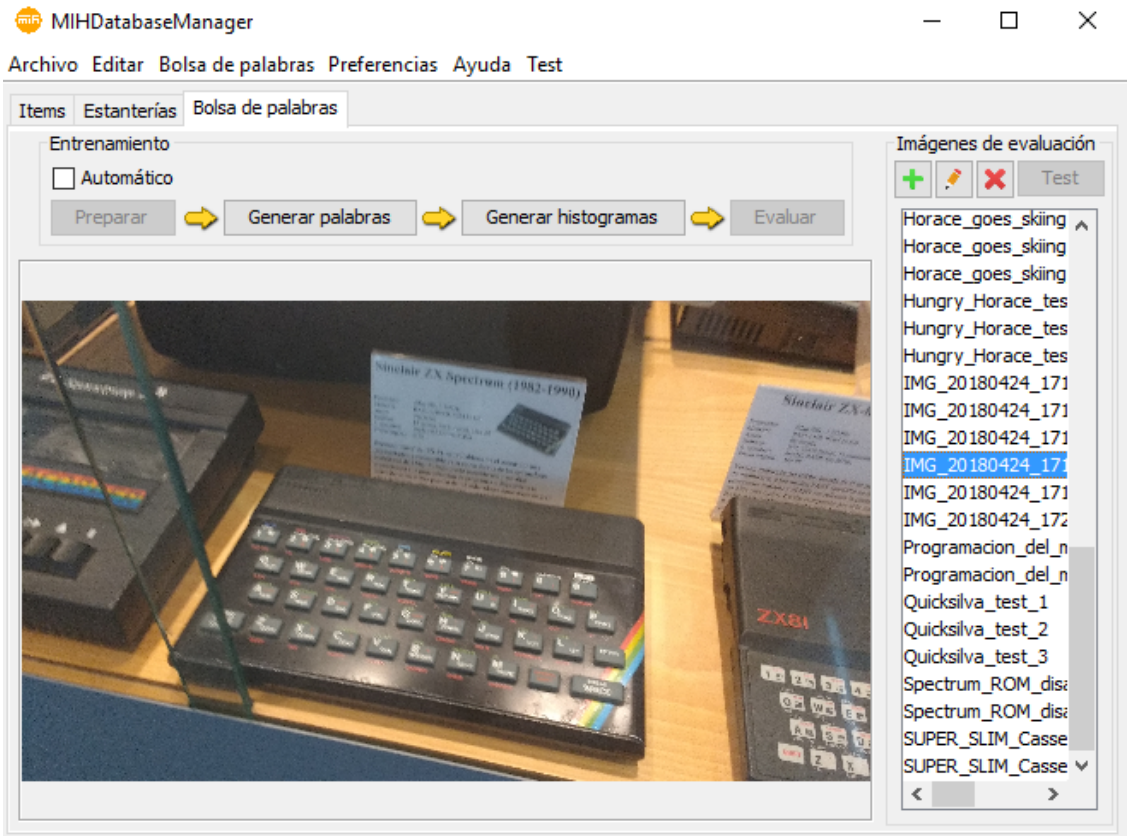



Figura E.10: Pantalla de gestión de la bolsa de palabras.