UNIVERSIDAD MIGUEL HERNÁNDEZ DE ELCHE
ESCUELA POLITÉCNICA SUPERIOR DE ELCHE
GRADO EN INGENIERÍA INFORMÁTICA EN
TECNOLOGÍAS DE LA INFORMACIÓN

Arquitectura y diseño de software on-premise multiplataforma y multilenguaje para el análisis de imágenes hiperespectrales

TRABAJO FIN DE GRADO
Febrero - 2026

AUTOR: Luis David Díaz Mesa
DIRECTOR: Miguel Onofre Martínez Rach

"Si no es cierto, no lo digas. Si no es correcto, no lo hagas."

- Marco Aurelio

RESUMEN

El análisis de imágenes hiperespectrales implica el procesamiento de grandes volúmenes de datos multidimensionales, lo que exige soluciones software capaces de ofrecer alto rendimiento computacional, eficiencia en la gestión de memoria y escalabilidad estructural. En este contexto, el presente Trabajo de Fin de Grado aborda el diseño e implementación de una arquitectura software on-premise —es decir, ejecutada localmente en la infraestructura propia para garantizar el control y rendimiento de los datos—, multiplataforma y multilenguaje, orientada a proporcionar una solución robusta y portable para el análisis avanzado de este tipo de información.

La propuesta arquitectónica se fundamenta en una estricta separación de responsabilidades entre la capa de presentación y la capa de procesamiento, garantizando modularidad, mantenibilidad y evolución independiente de cada componente. La interfaz de usuario se desarrolla mediante Kotlin Multiplatform y Compose Multiplatform, permitiendo la generación de aplicaciones nativas para distintos sistemas operativos a partir de una base de código común. El núcleo de procesamiento se implementa en C++, aprovechando su eficiencia y control de bajo nivel para el tratamiento intensivo de datos espectrales mediante librerias como OpenCV y OpenMP.

La interoperabilidad entre ambas capas se resuelve mediante Java Native Interface (JNI), lo que posibilita la integración segura y eficiente de código nativo dentro de un entorno multiplataforma moderno. Esta decisión arquitectónica permite combinar productividad en el desarrollo de interfaces con alto rendimiento computacional en el procesamiento, optimizando el equilibrio entre abstracción y control del sistema.

La solución es compatible con entornos Windows, Linux y macOS, abordando los desafíos asociados a la compilación, empaquetado y distribución en múltiples plataformas. Asimismo, se analizan las principales decisiones de diseño adoptadas, incluyendo el modelo de ejecución on-premise, la interoperabilidad entre lenguajes y la gestión eficiente de memoria y recursos.

Como resultado, se obtiene una herramienta funcional que valida la viabilidad de una arquitectura híbrida orientada a aplicaciones científicas de alto rendimiento, proponiendo un modelo arquitectónico replicable en otros dominios donde confluyan necesidades de portabilidad, eficiencia computacional y separación clara de responsabilidades.

AGRADECIMIENTOS

Agradecimientos a mis padres, por su paciencia infinita y ser mi sustento en esta etapa. También a los compañeros y amigos que he hecho durante esta etapa universitaria, tanto con los que he empecé como con los que finalizé. Por último pero no menos importante, a mi director de TFG, Miguel, por confiar en mí, ayudarme en el desarrollo de este proyecto y enseñarme distintos enfoques de ver y hacer las cosas.

ÍNDICE

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

1. Introducción

En los últimos años, el análisis de datos complejos ha adquirido un papel fundamental en múltiples ámbitos científicos e industriales. Entre estos, las imágenes hiperespectrales destacan por su capacidad para capturar información detallada del espectro electromagnético, permitiendo realizar análisis más precisos que los obtenidos mediante imágenes convencionales [1]. Sin embargo, el tratamiento de este tipo de datos implica un alto coste computacional y requiere herramientas software capaces de gestionar grandes volúmenes de información de forma eficiente.

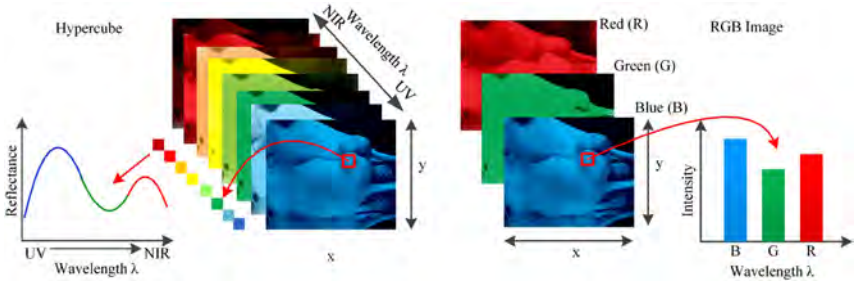

Figura 1.1: Comparación entre un hipercubo y una imagen RGB. Fuente: [1]

En este contexto, no solo resulta relevante el desarrollo de algoritmos de procesamiento, sino también el diseño de arquitecturas software que permitan integrar dichos algoritmos en sistemas robustos, escalables y mantenibles. La separación entre la lógica de procesamiento y la interfaz de usuario, así como la posibilidad de ejecutar la aplicación en distintos sistemas operativos, se convierte en un aspecto clave para garantizar la reutilización, evolución y adaptación del software a diferentes entornos.

Además, la creciente necesidad de trabajar en entornos distribuidos y colaborativos hace imprescindible plantear soluciones que puedan evolucionar hacia modelos de ejecución más flexibles, sin depender de tecnologías propietarias o limitaciones de plataforma. Por ello, este trabajo no se centra únicamente en la implementación funcional de una herramienta, sino en el diseño e integración de una arquitectura general que sirva como base para futuras ampliaciones y aplicaciones en distintos dominios.

1.1. Objetivos generales y específicos

Este trabajo tiene como finalidad diseñar un software escalable, modular, eficiente e intuitivo, con capacidad de evolución hacia una ejecución distribuida de la librería de procesamiento de datos.

Objetivo general

Definir y diseñar una arquitectura software que permita el desarrollo desacoplado de la interfaz de usuario y la librería de procesamiento de datos, utilizando tecnologías

que no requieran licencias propietarias y que faciliten su distribución.

Objetivos específicos

- Integrar la aplicación en entornos multiplataforma y multilenguaje.
- Implementar un procesamiento eficiente de datos hiperespectrales.
- Diseñar mecanismos de gestión y persistencia de datos.
- Desarrollar funcionalidades de visualización de imágenes hiperespectrales y representación gráfica.
- Validar el funcionamiento del sistema y evaluar la experiencia de usuario.

1.2. Alcance y limitaciones

El presente trabajo se centra en el prototipado de una aplicación que permita la visualización de hipercubos, la selección de regiones de interés y la generación de gráficos asociados, así como la posibilidad de modificar los datos del hipercubo mediante funciones de preprocesamiento o filtrado. Además, se busca proporcionar una solución que permita guardar y cargar proyectos, garantizando la persistencia de la información.

Entre las limitaciones del proyecto se encuentra el tipo de datos con los que se trabaja. La aplicación ha sido desarrollada específicamente para hipercubos generados por un modelo concreto de cámara, por lo que la lectura y visualización dependen de la estructura particular de los archivos generados por este dispositivo. No obstante, gracias a la modularidad de la aplicación, en un futuro sería posible incorporar soporte para otros formatos de cámaras y modos de lectura de datos distintos.

Otra limitación, menos crítica, está relacionada con la librería desarrollada en C/C++. Si bien en dispositivos de escritorio la aplicación funciona sin problemas, al ampliar la ejecución a otros tipos de dispositivos podrían surgir restricciones de hardware que afecten el rendimiento de la librería y el procesamiento de alta carga computacional. Esta limitación podría abordarse mediante un enfoque de ejecución distribuida, en el que la librería se ejecute en servidores o máquinas remotas, y la aplicación actúe únicamente como interfaz de interacción con dichos sistemas.

2. Tecnología hiperespectral

La tecnología hiperespectral es una técnica avanzada de adquisición de imágenes que captura información a lo largo de un amplio rango del espectro electromagnético. A diferencia de las imágenes convencionales, cada píxel de una imagen hiperespectral contiene un espectro completo, generando un *hipercubo* tridimensional que combina información espacial y espectral [1], [2]. Esta capacidad permite identificar y caracterizar materiales mediante el análisis de sus firmas espectrales únicas, lo que la hace aplicable en ámbitos tan variados como monitoreo ambiental, agricultura de precisión, industria alimentaria, y clasificación de residuos [3].

2.1. Funcionamiento

El proceso de adquisición de imágenes hiperespectrales(HSI, HyperSpectral Imaging) generalmente se realiza mediante cámaras que incorporan un espectrómetro. La secuencia típica de funcionamiento incluye:

1. **Captura de luz:** La cámara recoge la luz reflejada o emitida por la escena a través de lentes ópticas.
2. **Dispersión espectral:** La luz capturada se separa en sus componentes de longitud de onda mediante un elemento dispersivo como un prisma o rejilla de difracción.
3. **Selección de bandas:** El espectrómetro filtra las longitudes de onda de interés según la aplicación.
4. **Detección:** Detectores como CCDs o fotodiodos registran la intensidad de cada longitud de onda.
5. **Adquisición y construcción del hipercubo:** Los datos espectrales se organizan en un hipercubo tridimensional, donde las dimensiones espaciales representan la escena y la tercera dimensión corresponde a las longitudes de onda.
6. **Calibración:** Se aplican procedimientos de referencia para corregir sesgos del sensor y garantizar la precisión de los datos [1].

2.2. Sensores hiperespectrales y modos de adquisición

Los sistemas de imagen hiperespectral pueden clasificarse según cómo adquieren la información espacial y espectral, lo que influye en la resolución, velocidad de adquisición y formato de los datos generados [4]. Los principales modos de adquisición son:

- **Whiskbroom (escaneo punto a punto):** el detector captura todas las bandas espectrales de un único píxel a la vez. Ofrece alta resolución espectral, pero es lento y poco práctico para aplicaciones dinámicas.
- **Pushbroom (escaneo por líneas):** el sensor captura simultáneamente una línea completa de píxeles mientras se desplaza perpendicularmente. Este método es eficiente, estable y ampliamente utilizado en entornos industriales [5].
- **Wavelength scan (escaneo por longitud de onda):** se captura la escena completa mientras se varía secuencialmente la longitud de onda mediante filtros sintonizables. Adecuado para muestras estáticas.
- **Snapshot:** captura el hipercubo completo en un solo instante. Muy rápido, pero actualmente limitado en resolución espacial y espectral para aplicaciones de alta precisión [5].

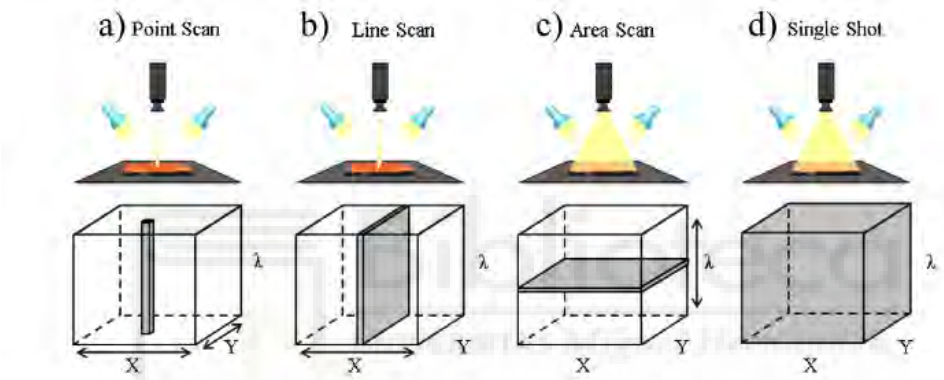

Figura 2.1: Configuraciones ópticas de los principales modos de adquisición hiperespectral: (a) *whiskbroom*, (b) *pushbroom*, (c) *wavelength scan* y (d) *snapshot*. Fuente: adaptado de [4].

En esta aplicación se ha adoptado el formato de lectura orientado a sensores **Pushbroom**, ya que corresponde a la organización inherente de los hipercubos generados por la cámara utilizada en el proyecto. Esta elección simplifica la lectura de los datos en crudo (*RAW*) y optimiza su integración con la librería nativa de procesamiento, garantizando compatibilidad y un acceso a memoria eficiente.

Debido a la naturaleza de la captura por barrido espacial, los datos generados por este tipo de sensores se almacenan habitualmente en disco siguiendo la estructura ***LINE-BAND-SAMPLE*** (LBS), la cual está directamente asociada al formato de archivo **BIL** (*Band Interleaved by Line*) [6]. En esta disposición, el hipercubo tridimensional se aplana en un arreglo lineal secuencial guardado línea por línea espacial; para cada línea escaneada en la escena, se almacenan consecutivamente todas las bandas espectrales y, dentro de cada banda, todos los píxeles (muestras o *samples*) correspondientes a dicha línea. Este enfoque ofrece un compromiso excelente para el procesamiento en tiempo real a medida que la cámara avanza a lo largo de una cinta transportadora industrial.

Dentro del estándar de codificación de imágenes hiperespectrales, existen otras dos estructuras clásicas de ordenación de memoria que difieren del formato BIL [6]:

- **BSQ (*Band Sequential*):** los datos se ordenan de forma que la banda espectral es la dimensión principal (*Band-Line-Sample*). Se almacena la imagen espacial 2D completa para la primera longitud de onda, seguida de la imagen completa para la segunda banda, y así sucesivamente. Es el formato óptimo cuando el software requiere visualizar o aplicar filtros espaciales sobre una banda de color concreta, ya que los píxeles de una misma frecuencia están contiguos en memoria.
- **BIP (*Band Interleaved by Pixel*):** la información se agrupa estructurando el píxel como la dimensión principal (*Line-Sample-Band*). Para cada píxel individual de la escena, se guardan de forma contigua los valores de todas sus bandas espectrales antes de pasar al siguiente píxel. Es la estructura ideal cuando el objetivo del procesamiento se centra en la extracción, comparación y clasificación de firmas espectrales puntuales.

2.2.1. Configuraciones ópticas de adquisición

Aunque el modo de adquisición define cómo se organiza la información espectral, la forma en que la luz interactúa con la muestra determina la configuración óptica utilizada [4]. Entre las más comunes se encuentran:

- **Reflectancia:** el detector capta la luz reflejada por la superficie de la muestra, evitando reflexiones especulares.
- **Transmitancia:** el detector registra la radiación que atraviesa el material, útil para analizar su estructura interna.
- **Interactancia:** la fuente de luz y el detector se colocan en el mismo lado de la muestra, captando radiación que ha penetrado parcialmente antes de emerger.

Estas configuraciones son importantes para la adquisición de datos precisos, pero no afectan directamente la forma en que se organizan los hipercubos ni la lectura implementada en la aplicación.

2.3. Ámbitos de uso

El análisis hiperespectral tiene aplicaciones en múltiples áreas científicas e industriales, entre ellas:

- **Monitoreo ambiental:** detección de contaminantes, seguimiento de la vegetación y calidad del agua.

- **Agricultura de precisión:** identificación de estrés hídrico, nutricional o enfermedades en cultivos.
- **Industria alimentaria:** control de calidad, detección de impurezas y análisis de composición de alimentos.
- **Gestión de residuos y reciclaje:** clasificación de materiales mediante firmas espectrales para optimizar el reciclaje [2].
- **Investigación científica:** estudios geológicos, análisis forense y aplicaciones biomédicas [3].

2.4. Características y requerimientos de los datos hiperespectrales

El hipercubo generado por HSI tiene características específicas que condicionan el diseño del software de análisis:

- **Alta dimensionalidad:** cientos de bandas espectrales por píxel, lo que genera grandes volúmenes de datos [1].
- **Resolución espectral y espacial:** la precisión de cada banda y el tamaño de píxel determinan la calidad de la información obtenida.
- **Formato específico de archivo:** cada cámara puede generar datos codificados de manera distinta, requiriendo módulos específicos para su lectura.
- **Necesidad de preprocesamiento:** corrección de ruido, calibración radiométrica y filtrado son pasos fundamentales antes del análisis.
- **Requerimientos de hardware:** el procesamiento eficiente de hipercubos requiere memoria suficiente y, en algunos casos, capacidades de cálculo paralelo o distribuido.

3. Herramientas Hardware y Software

El desarrollo de una aplicación multiplataforma orientada al análisis de imágenes hiperespectrales requiere la integración coordinada de herramientas hardware y software especializadas. En este proyecto se combinan dispositivos de adquisición espectral, entornos de desarrollo modernos y librerías optimizadas de procesamiento numérico e imagen, de forma que se garantice tanto la calidad de los datos como la eficiencia de los algoritmos implementados.

Antes de describir las herramientas concretas empleadas, se introducen brevemente los conceptos de *framework* y *librería*, ya que constituyen la base estructural del sistema desarrollado.

Framework

Un framework es un entorno de desarrollo que proporciona una estructura base sobre la cual construir una aplicación. Define convenciones arquitectónicas, componentes reutilizables y mecanismos de integración que facilitan el desarrollo, reducen errores estructurales y favorecen la mantenibilidad del software [7]. En la práctica, el framework invierte el control del flujo de la aplicación (*Inversion of Control*), de modo que el desarrollador encaja su código en puntos de extensión predeterminados.

En este proyecto se emplean frameworks como Kotlin Multiplatform y Compose Multiplatform, que permiten compartir lógica y, en el segundo caso, también parte de la interfaz de usuario entre distintas plataformas manteniendo capacidades nativas [8]. Asimismo, se utiliza Gradle como framework de automatización de construcción, que orquesta la compilación, la gestión de dependencias y la integración con otras herramientas del ecosistema.

Librería

Una librería es un conjunto de funciones, clases o módulos reutilizables que proporcionan funcionalidades específicas dentro de una aplicación. A diferencia de un framework, no impone una arquitectura global, sino que se integra según las necesidades del desarrollador [9]. En general, la aplicación invoca explícitamente los servicios de la librería, manteniendo el control del flujo de ejecución.

En este trabajo se utilizan librerías externas como OpenCV y OpenMP, además de un módulo nativo denominado *Hypercube_ native*, desarrollado en C++, que actúa como motor de procesamiento espectral del sistema. OpenCV es una librería de visión por computador de código abierto ampliamente utilizada para procesamiento de imágenes y vídeo en tiempo real, con cientos de algoritmos optimizados disponibles [10]. OpenMP, por su parte, es una API estándar para programación paralela en memoria compartida sobre C, C++ y Fortran, gestionada por el consorcio OpenMP Architecture Review Board [11].

3.1. Hardware de adquisición de hipercubos

Cámara hiperespectral Specim FX17 (900–1700 nm)

Para la adquisición de los hipercubos empleados durante el desarrollo del proyecto se ha utilizado la cámara hiperespectral Specim FX17. Se trata de un sistema de imagen tipo *pushbroom* que opera en el rango del infrarrojo cercano (NIR), entre 900 nm y 1700 nm, lo que la hace especialmente adecuada para aplicaciones de análisis y clasificación de materiales.

La cámara realiza la captura espectral línea por línea, generando el hipercubo mediante el movimiento relativo entre el sensor y la muestra. Su elevada resolución espectral y velocidad de adquisición la convierten en una solución idónea para entornos industriales y aplicaciones de procesamiento en tiempo real [12], [13].

Tabla 3.1: Especificaciones técnicas de la cámara Specim FX17. Fuente: Specim[13]

Parámetro	**Valor**
Rango espectral	900 – 1700 nm (NIR)
Número de bandas espectrales	224
Resolución espectral (FWHM)	8 nm
Resolución espacial	640 píxeles por línea
Velocidad de adquisición	Hasta 670 fps (modo completo)
Campo de visión (FOV)	38°
Número F	F/1.7
Relación señal/ruido (SNR pico)	1000:1
Interfaces de comunicación	CameraLink / GigE Vision
Dimensiones	150 × 85 × 75 mm
Peso	1.56 kg

3.2. Software y entornos de desarrollo

3.2.1. Sistema operativo

El desarrollo se realizó sobre Debian GNU/Linux (versión *forky/sid*) en una estación de trabajo equipada con un procesador Intel Core i7-5930K (6 núcleos / 12 hilos) y 32 GB de memoria RAM. La elección de un sistema Linux permite el acceso directo a compiladores nativos (GCC), herramientas de construcción (CMake, Make) y gestión de paquetes, facilitando la compilación de la librería nativa C++ y su integración con el entorno Java/Kotlin.

3.2.2. Java Development Kit (OpenJDK 17)

La aplicación de escritorio se ejecuta sobre la máquina virtual de Java, concretamente OpenJDK 17, que es la versión mínima requerida por Compose Multiplatform para el destino de escritorio (*Desktop*). La JVM gestiona la ejecución del código Kotlin compilado, la carga dinámica de la librería nativa a través de JNI y la interacción con el sistema de ventanas del sistema operativo.

3.2.3. Android Studio

Android Studio se utilizó como entorno de desarrollo integrado (IDE) principal para la gestión del proyecto, la compilación multiplataforma, la integración con Gradle y la depuración del sistema [14]. Desde este IDE se configuraron los módulos multiplataforma, las tareas de construcción y la ejecución en entornos de escritorio.

3.3. Frameworks

3.3.1. Kotlin y Kotlin Multiplatform

Kotlin es un lenguaje de programación moderno, de tipado estático y multiparadigma, desarrollado por JetBrains y adoptado por Google como lenguaje preferente para el desarrollo en Android desde 2019 [15]. Se ejecuta sobre la máquina virtual de Java (JVM), lo que le otorga interoperabilidad completa con el ecosistema Java existente, incluyendo el acceso a JNI para la invocación de código nativo. Entre sus características principales destacan la inferencia de tipos, las funciones de extensión, la seguridad frente a nulos (*null safety*) integrada en el sistema de tipos y el soporte nativo para corrutinas, que facilitan la programación asíncrona y concurrente.

Kotlin Multiplatform (KMP) es una extensión del lenguaje que permite compartir lógica común entre distintas plataformas —Android, iOS, escritorio (JVM) y web— manteniendo código específico de plataforma cuando es necesario [8]. El mecanismo de *expect/actual* permite declarar interfaces comunes que cada plataforma implementa de forma nativa, evitando la duplicación de código sin sacrificar el acceso a las API propias de cada sistema.

En este proyecto, Kotlin se emplea como lenguaje principal de la aplicación, mientras que KMP se utiliza para centralizar la lógica de gestión del hipercubo, la comunicación con el módulo nativo a través de JNI y la definición de los modelos de datos compartidos entre las distintas capas del sistema.

3.3.2. Compose Multiplatform

Compose Multiplatform es un framework declarativo para la construcción de interfaces gráficas modernas que extiende el paradigma de Jetpack Compose a múltiples

plataformas. Se utilizó para desarrollar la visualización interactiva del hipercubo y los gráficos espectrales asociados, aprovechando su modelo reactivo para actualizar de forma eficiente las vistas en función del estado de la aplicación.

3.4. Interfaces de interoperabilidad

3.4.1. JNI (Java Native Interface)

La Java Native Interface (JNI) es el mecanismo estándar de la plataforma Java que permite la comunicación bidireccional entre código gestionado por la máquina virtual (Kotlin/Java) y código nativo escrito en C/C++ [16]. Proporciona un conjunto de tipos y funciones que permiten invocar métodos nativos, gestionar referencias a objetos Java y transferir datos entre ambos entornos de ejecución.

En este proyecto, JNI constituye el elemento central de la arquitectura, ya que permite:

- Invocar funciones del módulo nativo *Hypercube_native* desde Kotlin.
- Transferir estructuras de datos espectrales (matrices `cv::Mat`, buffers de píxeles y arrays de espectros) entre memoria nativa y memoria gestionada.
- Ejecutar procesamiento intensivo en C++ mientras la interfaz gráfica permanece reactiva en Kotlin.

La utilización de JNI se justifica por la necesidad de alto rendimiento en operaciones numéricas y por la reutilización de código optimizado en C++.

No obstante, esta integración introduce riesgos inherentes como violaciones de acceso a memoria (*segmentation faults*) y fugas de memoria (*memory leaks*), lo que ha condicionado el diseño arquitectónico del sistema hacia una separación estricta de responsabilidades entre las capas Kotlin y C++.

Para ver el mapero de funciones externas declaradas en Kotlin y su implementación nativa en C++, se pueden consultar en el anexo .1.

3.5. Librerías

3.5.1. Módulo nativo C++ (Hypercube_native)

El núcleo de procesamiento espectral, denominado *Hypercube_native*, ha sido desarrollado en C++ como módulo nativo integrado en la aplicación. A diferencia de una librería independiente distribuible, este módulo forma parte del sistema y se compila específicamente para cada plataforma destino, enlazándose dinámicamente con la aplicación Kotlin a través de JNI.

Su desarrollo en C++ se justifica por la eficiencia en cálculo numérico, el bajo nivel de abstracción y el control explícito sobre la gestión de memoria. El módulo implementa estructuras de datos optimizadas para el almacenamiento de hipercubos tridimensionales, algoritmos de filtrado y normalización espectral, extracción de regiones de interés (ROI) y cálculo de firmas espectrales medias.

3.5.2. OpenCV

OpenCV (Open Source Computer Vision Library) es una biblioteca de visión por computadora y aprendizaje automático de código abierto, inicialmente desarrollada por Intel y posteriormente mantenida por la comunidad [10].

En el contexto de este proyecto, OpenCV se emplea específicamente para:

- **Almacenamiento de hipercubos**: en la clase `HcData` de la libería, usamos la clase `cv::Mat` multidimensional (3D) que se utiliza como contenedor principal de los datos espectrales procesados, con *layout* `[lines, bands, samples]`, de esta manera se evita modificar los datos originales (RAW) que corresponden a los datos que se leen del binario y se guardan en un puntero `uint16_t` dentro de la clase `Hypercube` de la librería.
- **Normalización de bandas**: funciones como `minMaxLoc` y `convertTo` permiten escalar los valores espectrales de 16 bits a 8 bits para su visualización.
- **Reordenación espacial**: la función `transpose` se emplea para adaptar la orientación de las bandas al formato de visualización requerido por la interfaz.

3.5.3. OpenMP

OpenMP es una API para paralelización en sistemas de memoria compartida que permite distribuir el cálculo entre múltiples hilos de ejecución [11]. La especificación define directivas de compilador, funciones de biblioteca y variables de entorno que facilitan la creación de regiones paralelas y el reparto de trabajo entre hilos.

En el contexto de este proyecto, OpenMP se emplea para paralelizar operaciones sobre el hipercubo, principalmente los bucles de normalización por bandas espectrales y el cálculo de estadísticos (mínimo, máximo, media) sobre regiones espaciales. La combinación de C++ y OpenMP permite escalar el rendimiento del sistema en función del número de núcleos disponibles en el hardware de ejecución, lo cual resulta especialmente relevante dado el volumen de datos manejado (hipercubos de hasta 458 MB con 224 bandas de 640×1600 píxeles).

3.5.4. Vico Charts

Vico Charts [17] es una librería de gráficos multiplataforma compatible con Compose Multiplatform, utilizada para la representación interactiva de firmas espectrales

dentro de la aplicación. Permite visualizar curvas de reflectancia asociadas a píxeles individuales o a regiones de interés (ROI), superponer múltiples series espectrales con codificación por color y ajustar dinámicamente los ejes en función del tipo de datos (valores crudos de 16 bits o valores normalizados en punto flotante).

Su integración con el modelo reactivo de Compose permite que los gráficos se actualicen automáticamente cuando el usuario selecciona un nuevo píxel, activa o desactiva una ROI, o cambia la versión del hipercubo visualizada.

3.5.5. OSHI

OSHI (Operating System and Hardware Information) [18] es una biblioteca de código abierto para Java basada en **JNA (Java Native Access)** [19]. JNA permite que la aplicación interactúe directamente con las librerías nativas del sistema operativo para obtener información que no está disponible de forma estándar en la Máquina Virtual de Java (JVM). Gracias a esta integración, OSHI permite monitorizar de manera eficiente el inventario de hardware (CPU, memoria, sensores), el estado del almacenamiento y las métricas de rendimiento de los procesos en ejecución.

3.6. Build Tools

3.6.1. Gradle

Gradle es el sistema de automatización de construcción empleado para la gestión de dependencias, la compilación multiplataforma y la integración del código nativo mediante CMake [20]. Su modelo basado en tareas permite definir flujos de construcción personalizados que abarcan desde la compilación de la librería C++ hasta la generación de los artefactos finales para cada plataforma.

3.6.2. CMake

CMake se emplea como sistema de construcción para compilar el módulo nativo C++ y generar binarios compatibles con distintas arquitecturas, integrándose dentro del flujo de compilación de Gradle [21]. A través de ficheros de configuración (`CMakeLists.txt`) se describen las fuentes, las opciones del compilador, los *flags* de enlace con OpenCV y OpenMP, y las dependencias necesarias para producir la librería compartida (`.so`/`.dll`) utilizada por la aplicación.

3.7. Control de versiones

3.7.1. Git

Git es un sistema de control de versiones distribuido utilizado para la gestión del código fuente, el control de cambios y el desarrollo incremental del proyecto [22]. Permite trabajar con ramas, fusionar modificaciones y mantener un historial detallado de la evolución del repositorio.

3.7.2. GitHub

GitHub se empleó como repositorio remoto para el almacenamiento del código, la gestión de ramas y el control de versiones colaborativo [23]. Además, facilita la integración con herramientas de automatización y proporciona funcionalidades adicionales como el seguimiento de incidencias y la revisión de código.

3.8. Herramientas de documentación

3.8.1. LaTeX

La presente memoria ha sido redactada utilizando LaTeX, un sistema de composición tipográfica ampliamente empleado en el ámbito académico y científico. LaTeX permite producir documentos con formato profesional, gestionar referencias bibliográficas de forma automatizada y mantener una separación clara entre contenido y presentación, lo que facilita la edición iterativa de documentos extensos como un trabajo de fin de grado.

3.8.2. PlantUML

Para la conceptualización y el diseño arquitectónico de la aplicación se ha empleado la extensión PlantUML [24]. Esta herramienta permite generar diagramas UML, como diagramas de clases y de secuencia, a partir de descripciones textuales simples. Su uso ha sido clave para documentar visualmente el flujo de datos entre las distintas capas del sistema, especialmente en las interacciones complejas de la interfaz JNI.

3.9. Herramientas de asistencia al desarrollo

3.9.1. Claude (Anthropic)

Durante el desarrollo del proyecto se empleó Claude, un modelo de lenguaje desarrollado por Anthropic, como herramienta de asistencia en tareas de programación.

Su uso se centró en la depuración de errores de compilación y ejecución en la capa C++/JNI, la resolución de problemas de interoperabilidad entre Kotlin y código nativo, y la revisión de estructuras de datos y flujos de memoria. En todos los casos, las sugerencias generadas fueron revisadas, adaptadas y validadas por el autor antes de su incorporación al código fuente.

3.9.2. Gemini (Google)

Junto al uso de Claude para la depuración en C++, el desarrollo de la interfaz gráfica se apoyó en los modelos de Google [25]. Específicamente, se utilizó la herramienta *Nanobana de Gemini* para la conceptualización y el diseño de la experiencia de usuario (UX/UI). Además, el propio modelo Gemini asistió en la generación y estructuración de los componentes reactivos en Compose Multiplatform, siempre bajo una estricta supervisión, revisión y adaptación del código resultante para garantizar su integración correcta en la arquitectura del proyecto.

4. Descripción de la aplicación

Para entender distintos aspectos que se muestran en la arquitectura del sistema, vamos a describir brevemente la aplicacion que vamos a desarrollar con dicha arquitectura, de esta forma quedaran mas claros algunos conceptos arquitectonicos.

La herramienta desarrollada, denominada *Hyperspectral Analysis Tool* (HAT),su funcionalidad es conseguir analizar, comparar y ver distintas firmas espectrales que aparezca en una captura. Su objetivo principal de este trabajo de fin de grado es proporcionar una solución *on-premise* robusta y escalable que logre equilibrar una experiencia de usuario ágil y moderna con el alto rendimiento computacional que exige el tratamiento de grandes volúmenes de datos multidimensionales.

Para dar respuesta a estas necesidades, la aplicación implementa el siguiente conjunto de funcionalidades principales:

- **Gestión de proyectos y entorno de trabajo:** Creación, guardado y restauración de sesiones de trabajo de forma no destructiva. La aplicación facilita la carga ágil de archivos binarios mediante interacciones de arrastrar y soltar (*drag & drop*) en su pantalla de inicio (Figura 7.1), así como la gestión completa del ciclo de vida del proyecto desde su menú principal (Figura 7.2), garantizando la persistencia de los metadatos sin alterar el hipercubo original.
- **Visualización interactiva:** Renderizado espacial de las distintas bandas del hipercubo en un visor central. Incluye soporte fluido para la exploración de imágenes de alta resolución mediante herramientas de *zoom* y desplazamiento matricial (*pan*).
- **Gestión de Regiones de Interés (ROIs) y Materiales:** Herramientas de dibujo interactivo para la selección geométrica de píxeles individuales o áreas

poligonales. Estas regiones pueden ser administradas desde un panel lateral dedicado (Figura 7.20) y vinculadas a clases o materiales específicos mediante cuadros de diálogo contextuales (Figura 7.19), sentando las bases para tareas de clasificación supervisada.

- **Análisis espectral y procesamiento numérico:** Extracción algorítmica y representación gráfica en tiempo real de las firmas espectrales correspondientes a las áreas seleccionadas. Adicionalmente, el sistema expone un panel de funciones (Figura 7.21) que permite configurar y aplicar algoritmos de preprocesamiento (como normalizaciones radiométricas). Estas transformaciones se gestionan mediante un historial de versiones en memoria, facilitando al usuario la comparación visual y analítica de los perfiles espectrales resultantes.

El diseño de todas estas funcionalidades se sustenta en una separación estricta de responsabilidades entre la interfaz gráfica y el motor numérico subyacente. Una exposición visual más amplia y detallada sobre la ejecución práctica de estas herramientas puede consultarse en el Capítulo 7, correspondiente a los resultados de este trabajo.

5. Arquitectura del Sistema

El sistema desarrollado sigue una arquitectura modular multicapa que separa claramente la interfaz de usuario, la lógica de aplicación y el motor de procesamiento nativo. Esta separación permite mantener la portabilidad del front-end sin comprometer el rendimiento del procesamiento espectral.

5.1. Arquitectura general

La arquitectura global del sistema se estructura en cuatro niveles claramente diferenciados, cada uno con responsabilidades delimitadas y un flujo de dependencias estrictamente descendente:

1. **Capa de presentación (Front-End):** desarrollada en Kotlin con Compose Multiplatform, gestiona la interfaz gráfica, la interacción del usuario y el estado reactivo de la aplicación. No contiene lógica de procesamiento espectral.
2. **Capa de lógica compartida:** implementada en Kotlin Multiplatform, centraliza los modelos de dominio, los gestores de estado (*managers*) y la orquestación de casos de uso. Traduce las acciones del usuario en operaciones concretas sobre los datos.
3. **Capa de interoperabilidad (JNI):** constituida por la clase `NativeBridge`, actúa como punto de acceso único al módulo nativo. Encapsula la conversión de tipos entre el mundo Java/Kotlin (objetos gestionados por la JVM) y el mundo C++ (punteros, buffers de memoria, arrays nativos).

4. **Motor de procesamiento nativo:** módulo compilado en C++ que encapsula toda la lógica de cálculo intensivo: carga de hipercubos, filtrado espectral, normalización, extracción de ROIs y cálculo de firmas espectrales medias. Se comunica con la capa superior exclusivamente a través de JNI.

La Figura 5.1 muestra la organización estructural del sistema.

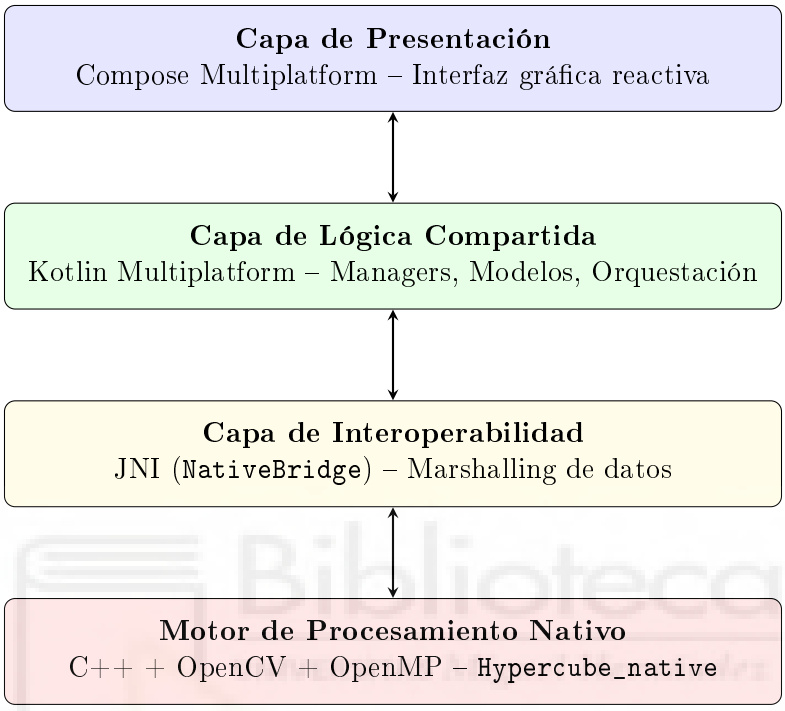

Figura 5.1: Arquitectura general del sistema en capas.

El flujo de datos principal sigue un patrón descendente para las operaciones de escritura —el usuario solicita una acción que se propaga hasta C++— y ascendente para las de lectura —C++ devuelve datos que se transforman y visualizan en la UI—. Esta comunicación bidireccional se realiza siempre a través de `NativeBridge`, que expone aproximadamente treinta funciones estáticas agrupadas por dominio funcional (gestión de hipercubos, información dimensional, datos espectrales, funciones de procesamiento de datos, ROIs y materiales).

La separación proporciona tres beneficios concretos. En primer lugar, desacopla la interfaz gráfica del motor de cálculo, de modo que es posible modificar la UI sin afectar a los algoritmos de procesamiento, y viceversa. En segundo lugar, la capa JNI aísla las complejidades de la gestión de memoria nativa (punteros, buffers, ciclo de vida de objetos C++) del código Kotlin, reduciendo el riesgo de violaciones de acceso a memoria (*segmentation faults*) y fugas de memoria (*memory leaks*). Finalmente, la centralización de la lógica de negocio en la capa compartida permite reutilizar el código si en el futuro se desarrollasen versiones para otras plataformas, ya que los modelos de dominio y los gestores no dependen de la tecnología de UI concreta.

5.2. Arquitectura del Front-End

5.2.1. Patrón arquitectónico: Clean Architecture

La capa de front-end se organiza siguiendo los principios de Clean Architecture, un patrón propuesto por Robert C. Martin que estructura el software en capas concéntricas con dependencias dirigidas siempre hacia el interior [26]. El objetivo fundamental es separar las reglas de negocio de los detalles de implementación —frameworks, bases de datos, interfaces de usuario—, de modo que el núcleo del sistema permanezca estable ante cambios en las tecnologías externas.

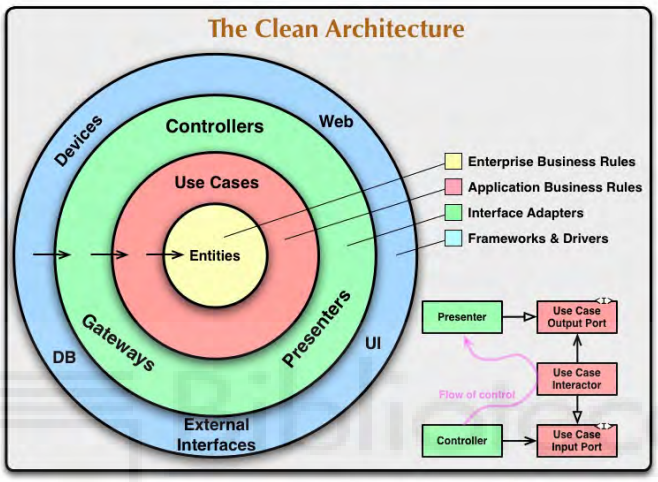

Figura 5.2: Diagrama general de Clean Architecture. Fuente: [26].

Los principios clave que esta arquitectura impone y que se han aplicado en el proyecto son:

- **Regla de dependencia (*Dependency Rule*):** las capas internas no conocen a las externas. El dominio no importa clases de la UI, ni de la librería nativa, ni del sistema de persistencia.
- **Inversión de dependencias (prevista):** el diseño contempla que las capas externas implementen interfaces definidas en el dominio. Se han definido las interfaces `ComputeRepository` y `PersistenceRepository` en `domain/`, aunque en la versión actual los managers acceden directamente a las implementaciones concretas. La activación de este principio se ha planificado como trabajo futuro.
- **Independencia de frameworks:** la lógica de negocio no depende de Compose ni de JNI. Los modelos de dominio (`Hypercube`, `RoiData`, `MaterialData`) son clases Kotlin puras, lo que facilita su reutilización y testeo.

La adopción de este patrón resulta especialmente adecuada para HAT(Hyperspectral Analysis Tool) por dos razones. Por un lado, la aplicación combina una interfaz gráfica reactiva con un motor de procesamiento nativo, y Clean Architecture permite que

ambos evolucionen de forma independiente. Por otro lado, el sistema de persistencia en disco (archivos `.hc`, JSONs de metadatos, binarios `.bhc`) y la comunicación JNI son detalles de infraestructura que no deben contaminar las reglas de negocio.

Cabe señalar que la aplicación de Clean Architecture en este proyecto es parcial y pragmática: se respeta la regla de dependencia en la mayoría de las capas, pero la inversión de dependencias mediante interfaces de repositorio no se ha activado en esta versión. Esta decisión se justifica por el alcance de un trabajo de fin de grado, donde la prioridad ha sido completar la funcionalidad del sistema. Los detalles de implementación de cada capa —incluyendo la comunicación JNI, la declaración de interfaces nativas en Kotlin y su implementación en C++— se describen en la Sección 6.

5.2.2. Aplicación al proyecto

La Figura 5.3 muestra cómo se proyectan los anillos de Clean Architecture sobre la estructura de paquetes del proyecto.

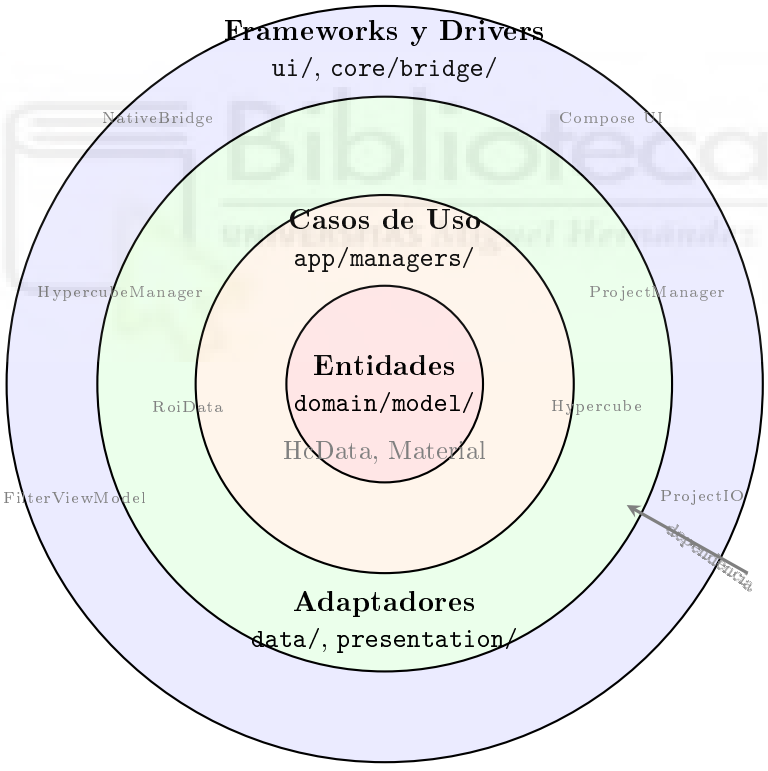

Figura 5.3: Proyección de Clean Architecture sobre la estructura de paquetes del proyecto.

La correspondencia entre los anillos de Clean Architecture y los paquetes del proyecto es la siguiente:

- **Entidades** (`domain/model/`): clases que representan los conceptos fundamentales del dominio (`Hypercube`, `HcData`, `RoiData`, `MaterialData`, `ProjectContext`).

No dependen de ningún framework.

- **Casos de uso** (`app/managers/`, `app/AppState.kt`): orquestadores que coordinan las operaciones del sistema. `AppState` actúa como punto de entrada para los casos de uso principales (crear proyecto, cargar hipercubo, guardar) y delega la ejecución a `ProjectManager` y `HypercubeManager`.
- **Adaptadores de interfaz** (`data/`, `presentation/`): traducen datos entre el formato del dominio y el formato requerido por los agentes externos. Incluyen `ProjectIO` (serialización a disco), `NativeCompute` (adaptador sobre JNI) y `FilterViewModel` (preparación de datos para la vista).
- **Frameworks y drivers** (`ui/`, `core/bridge/`): los detalles tecnológicos más externos. Los composables de Compose Multiplatform (`ui/`) y el singleton `NativeBridge` (`core/bridge/`) que ejecuta las llamadas JNI reales.

5.2.3. Estructura de directorios

El código fuente del front-end se organiza en los siguientes paquetes, cada uno con una responsabilidad específica dentro de la arquitectura. La Figura 5.4 muestra la estructura completa.

```
src/commonMain/kotlin/es/umh/tfg/luisdavid/
├── main.kt ... Punto de entrada
├── Platform.jvm.kt ... Implementación específica JVM
├── core/
│   └── bridge/
│       └── NativeBridge.kt ... Declaraciones JNI (~30 external .1)
├── domain/
│   ├── model/
│   │   ├── hypercube/ ... Hypercube, HcData, Roi, Material
│   │   └── project/ ... ProjectContext
│   ├── repository/ ... ComputeRepository, PersistenceRepository
│   └── filter/ ... FilterProvider, FilterInfo, FilterParam
├── data/
│   ├── io/ ... ProjectIO, HatSolution, HatSolutionParser
│   ├── dto/ ... ProjectDtos (serialización JSON)
│   └── repository/ ... NativeCompute, LocalPersistence (stubs)
├── app/
│   ├── AppState.kt ... Orquestador principal
│   ├── di/ ... AppModule (inyección de dependencias)
│   └── managers/ ... ProjectManager, HypercubeManager
├── presentation/
│   ├── viewmodels/ ... FilterViewModel
│   ├── state/ ... RoiSelectionState, EditorTab, MenuCategory
│   └── utils/ ... CoordinateUtils
└── ui/
    ├── layout/ ... AppDesktop, MainLayout, EmptyStates
    ├── modules/
    │   ├── image/ ... CentralImageViewer, ImageToolbar
    │   ├── workspace/ ... WorkspaceView, EditorTabBar
    │   ├── sidebar/ ... ProjectSidebar, ExplorerPanelView
    │   └── analysis/tabs/ ... Spectrum, ROIs, Materials, Functions
    ├── components/ ... Diálogos, botones, overlays, menús
    ├── charts/ ... SpectrumChart (Vico Charts)
    ├── dnd/ ... Sistema de Drag and Drop
    ├── theme/ ... Colors, Type, Theme
    └── utils/ ... ImageUtils, ViewportUtils, GeometryUtils
```

Figura 5.4: Árbol de directorios del código fuente Kotlin del proyecto.

5.2.4. Diagramas de clase de la capa de aplicación

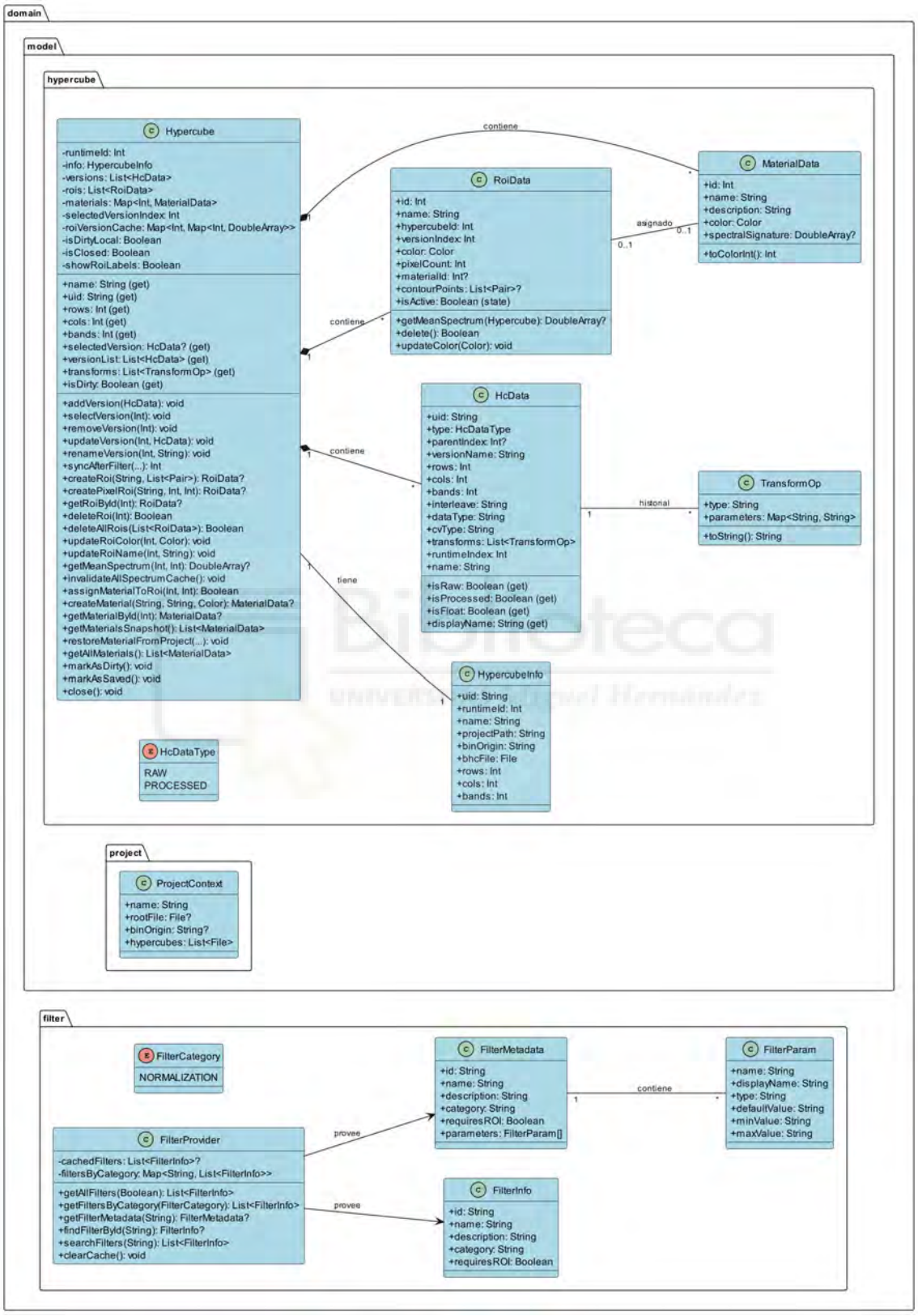

Figura 5.5: Diagrama de clases de los modelos de dominio en Kotlin. Se muestran las entidades fundamentales (`Hypercube`, `HcData`, `RoiData`, `MaterialData`) y sus relaciones.

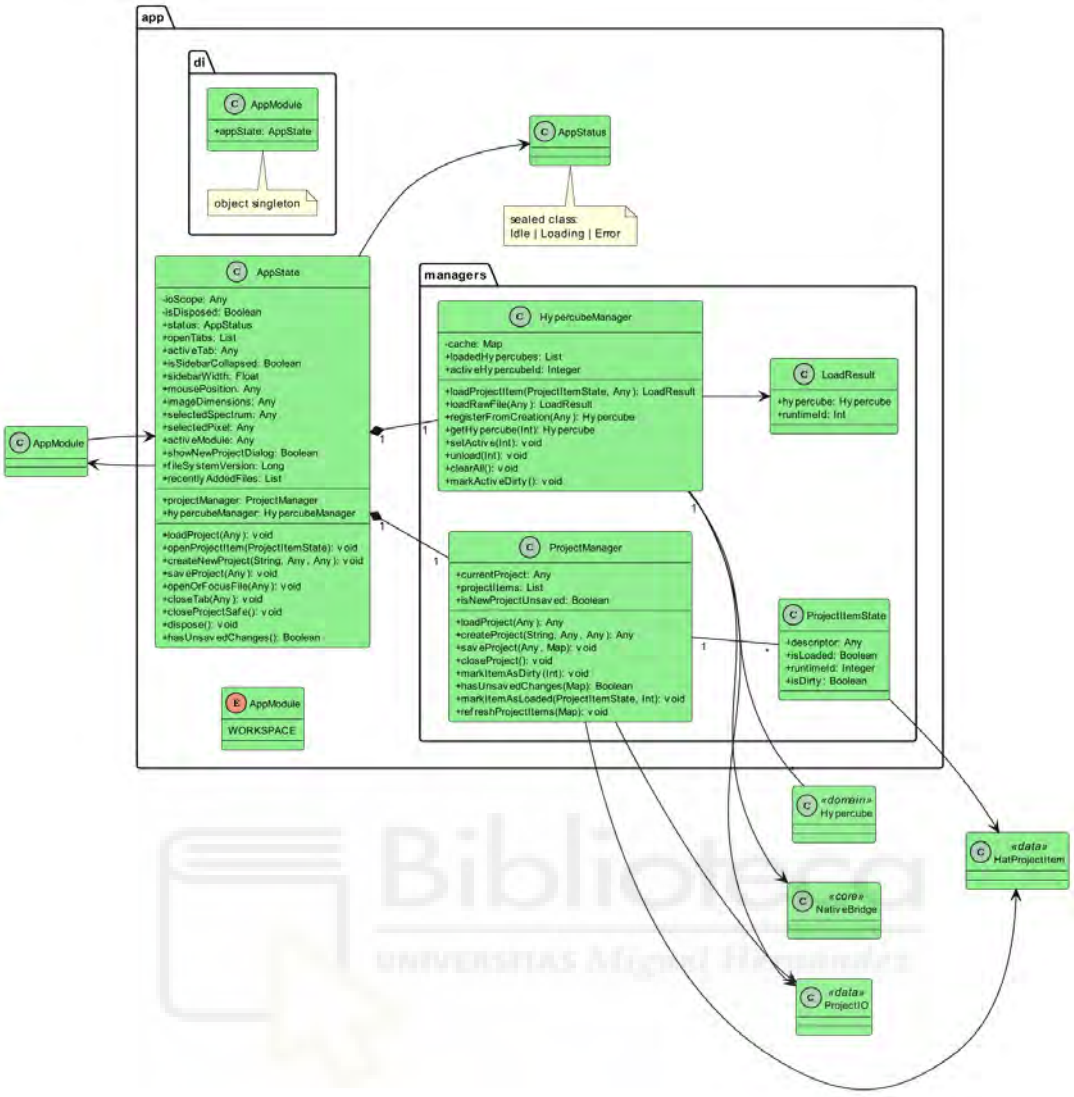

Figura 5.6: Diagrama de clases de la capa de aplicación. Se ilustra la estructura de `AppState` como orquestador principal, los managers (`HypercubeManager`, `ProjectManager`) y su relación con el estado de la UI.

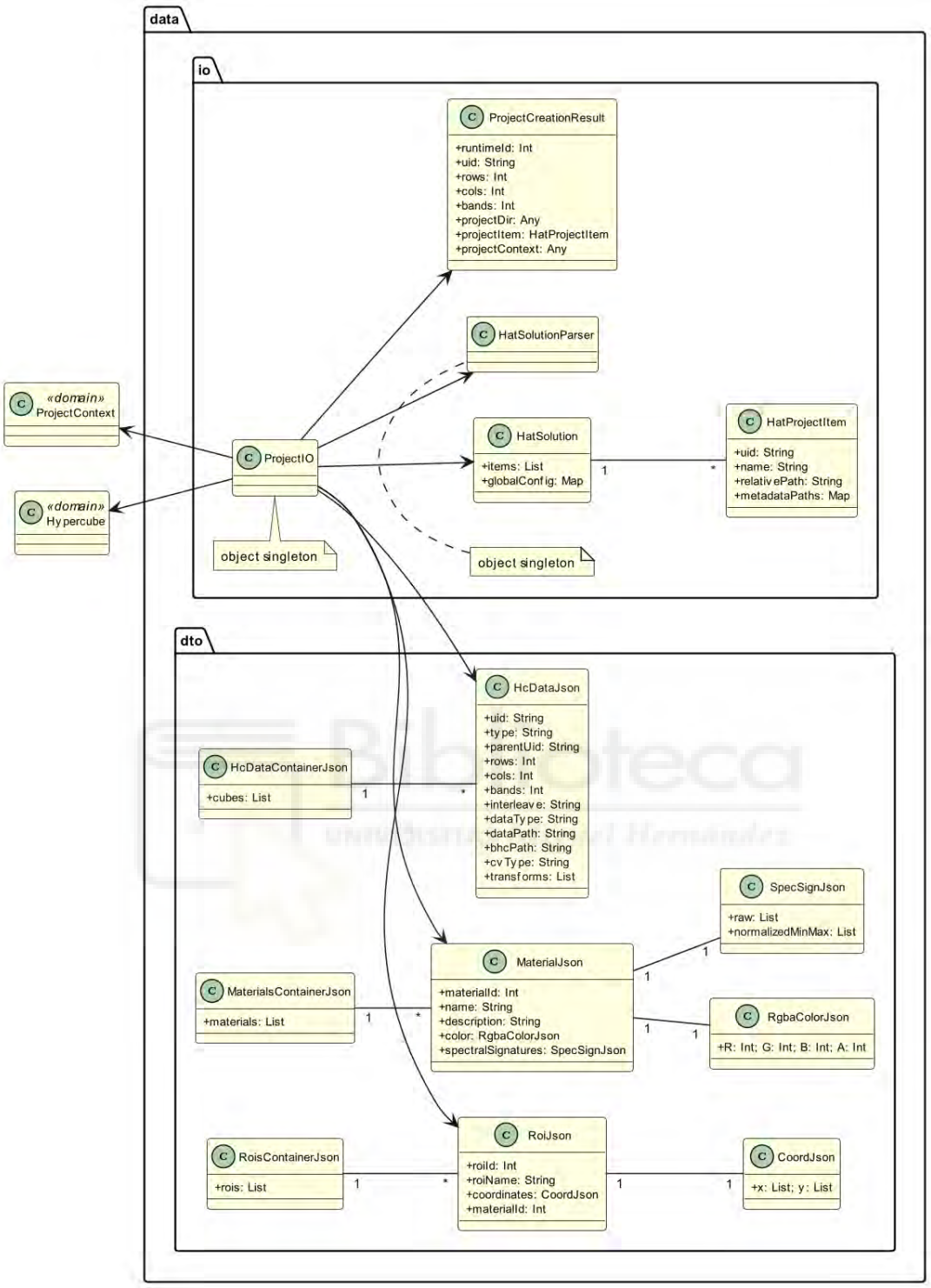

Figura 5.7: Diagrama de clases de la capa de datos. Se muestra la organización de los objetos de transferencia (DTOs) y los objetos de acceso a datos (`ProjectIO`, `HatSolution`, `HatProjectItem`) que gestionan la persistencia en disco.

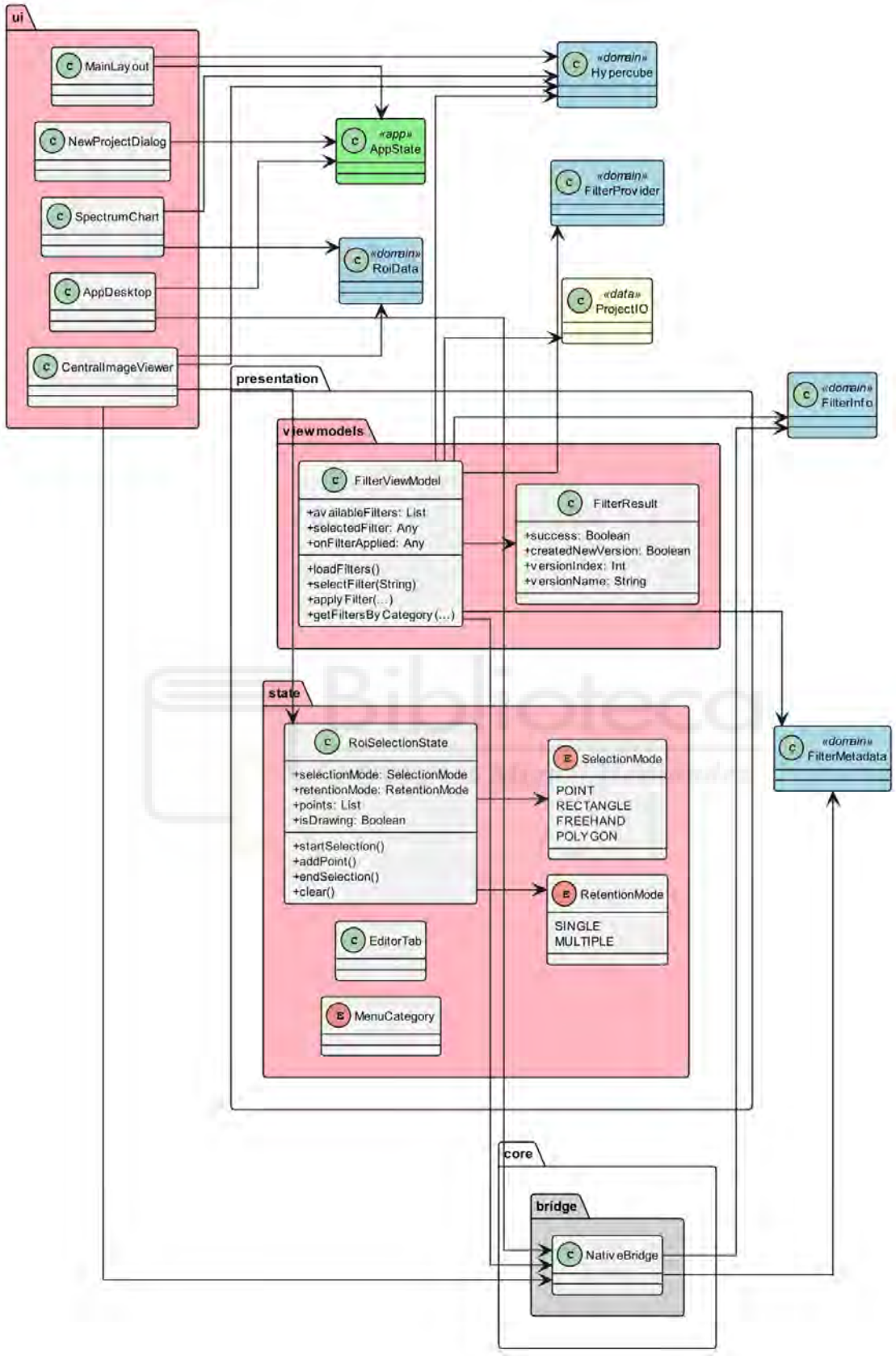

Figura 5.8: Diagrama de clases de la capa de presentación. Se representan los ViewModels (`FilterViewModel`), los estados de UI (`RoiSelectionState`) y su interacción con los componentes de la interfaz gráfica.

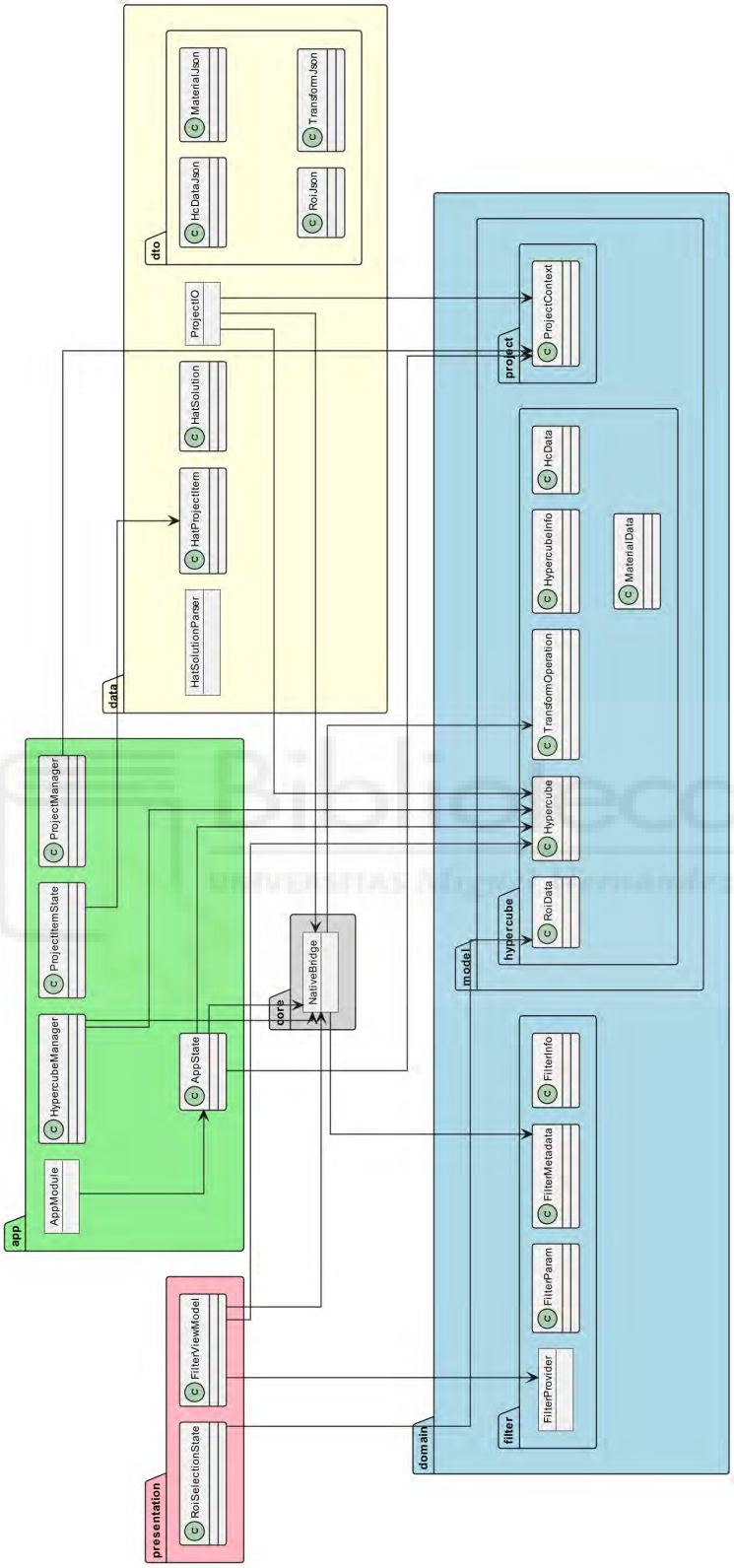

Figura 5.9: Diagrama relacional completo de la capa Kotlin. Se muestran todas las dependencias entre las distintas capas de la arquitectura: dominio (azul), aplicación (verde), datos (amarillo), presentación (rosa) y core/puente JNI (gris).

Descripción de las relaciones

El diagrama relacional completo (Figura 5.9) muestra la arquitectura de paquetes y las dependencias entre las distintas capas del front-end:

- **Capa de dominio (azul)**: Contiene las entidades fundamentales del sistema (`Hypercube`, `HcData`, `RoiData`, `MaterialData`) que modelan los conceptos de negocio. No depende de ninguna otra capa, cumpliendo con el principio de independencia de frameworks de Clean Architecture.
- **Capa de aplicación (verde)**: Alberga los orquestadores principales (`AppState`) y los managers (`HypercubeManager`, `ProjectManager`) que coordinan los casos de uso. Depende de la capa de dominio y de las capas inferiores (datos y core).
- **Capa de datos (amarillo)**: Implementa la persistencia mediante `ProjectIO` y los objetos de transferencia (DTOs) para la serialización JSON. Actúa como adaptador entre el dominio y el sistema de archivos.
- **Capa de presentación (rosa)**: Contiene los ViewModels (`FilterViewModel`) y los estados de UI (`RoiSelectionState`) que preparan los datos para su visualización. Depende del dominio y de la capa de datos.
- **Capa core (gris)**: Representa el puente JNI (`NativeBridge`) que actúa como fachada única hacia el motor nativo C++. Todas las capas que necesitan procesamiento espectral dependen de este componente.

Esta organización en capas con dependencias unidireccionales (siempre hacia el interior) garantiza un bajo acoplamiento y facilita el mantenimiento y la evolución del sistema. La separación clara entre la lógica de negocio (dominio), la orquestación (aplicación), la persistencia (datos) y la interfaz de usuario (presentación) permite modificar cualquiera de estas capas de forma independiente, siempre que se respeten los contratos establecidos.

5.3. Arquitectura de la librería de procesamiento

La librería de procesamiento, denominada `Hypercube_native`, constituye el motor de cálculo del sistema. Está implementada íntegramente en C++ (estándar C++17) y se compila como una librería compartida (`.so` en Linux, `.dll` en Windows) que se carga en tiempo de ejecución desde la JVM mediante JNI [16]. Su diseño sigue el patrón de gestores singleton [7] para administrar los recursos en memoria, y delega las operaciones numéricas intensivas a OpenCV [27] y OpenMP [11].

5.3.1. Estructura de directorios

La Figura 5.10 muestra la organización del código fuente de la librería.

```
Hypercube_native/
├── CMakeLists.txt ... Configuración de compilación
├── jni/
│   └── nativeBridge.cpp ... Interfaz JNI (∼35 funciones exportadas)
└── hat/
    ├── models/
    │   ├── include/
    │   │   ├── hypercube.h ... Hipercubo: versiones + ROIs + materiales
    │   │   ├── hcData.h ... Versión de datos (cv::Mat 3D)
    │   │   ├── roi.h ... Región de interés (contorno + máscara)
    │   │   └── material.h ... Material espectral (color + firma)
    │   ├── hypercube.cpp
    │   ├── hcData.cpp
    │   ├── roi.cpp
    │   └── material.cpp
    ├── managers/
    │   ├── include/
    │   │   ├── hypercubeManager.h ... Singleton de hipercubos
    │   │   ├── roiManager.h ... Singleton de ROIs
    │   │   ├── materialManager.h ... Singleton de materiales
    │   │   └── filterRegistry.h ... Registro de funciones de procesamiento y filtros
    │   ├── hypercubeManager.cpp
    │   ├── roiManager.cpp
    │   ├── materialManager.cpp
    │   └── filterRegistry.cpp
    ├── filters/
    │   ├── include/
    │   │   └── normalizeFilter.h ... Funciones de filtros de normalización
    │   └── normalizeFilter.cpp
    └── utils/
        ├── include/
        │   ├── imageIO.h ... Carga/guardado de binarios
        │   ├── spectralOps.h ... Operaciones espectrales
        │   └── bandImageAdapter.h ... Conversión banda → buffer 8 bits
        ├── imageIO.cpp
        ├── spectralOps.cpp
        └── bandImageAdapter.cpp
```

Figura 5.10: Árbol de directorios de la librería `Hypercube_native`.

5.3.2. Diagramas de clase de la librería de procesamiento

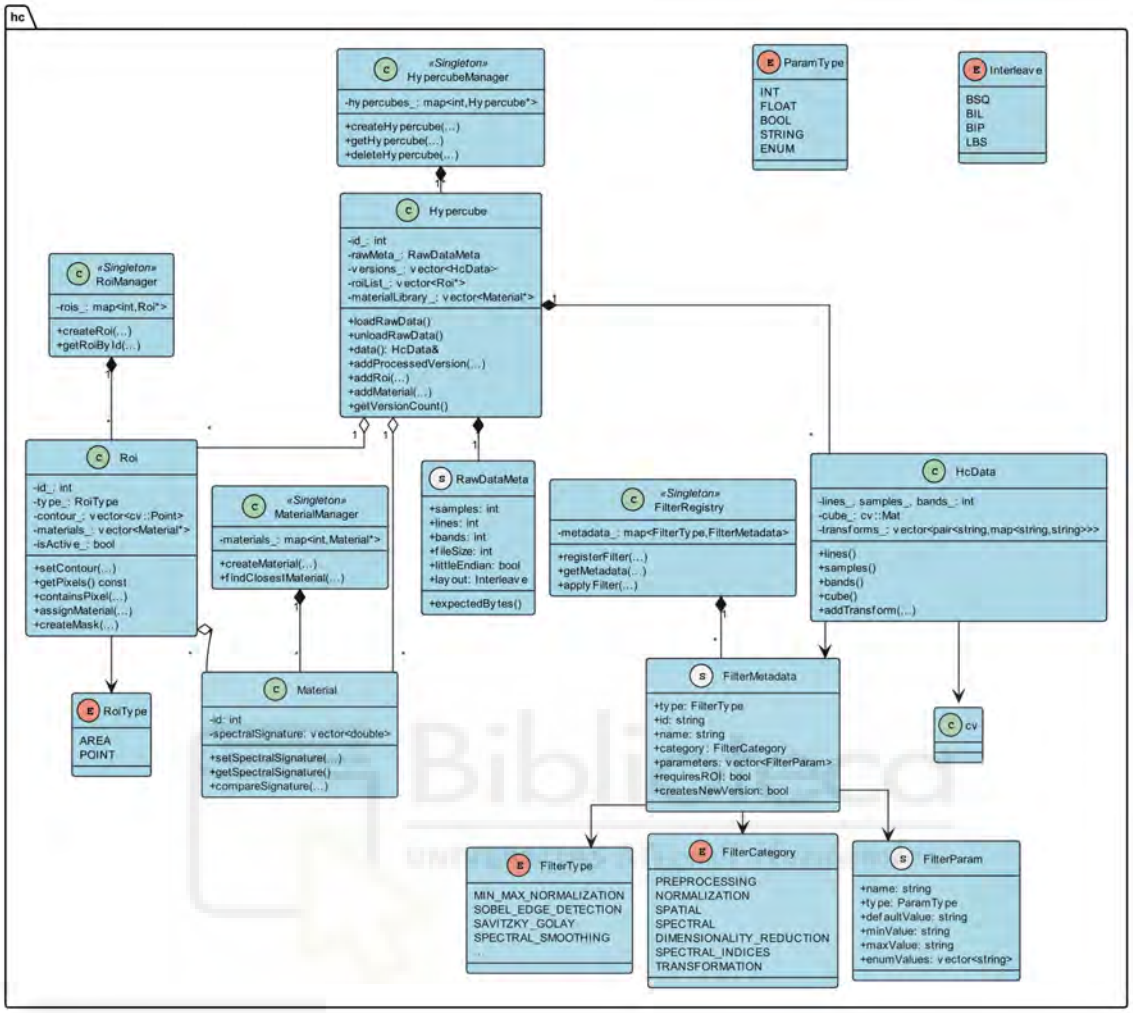

Figura 5.11: Diagrama de clases de la librería `Hypercube_native`.

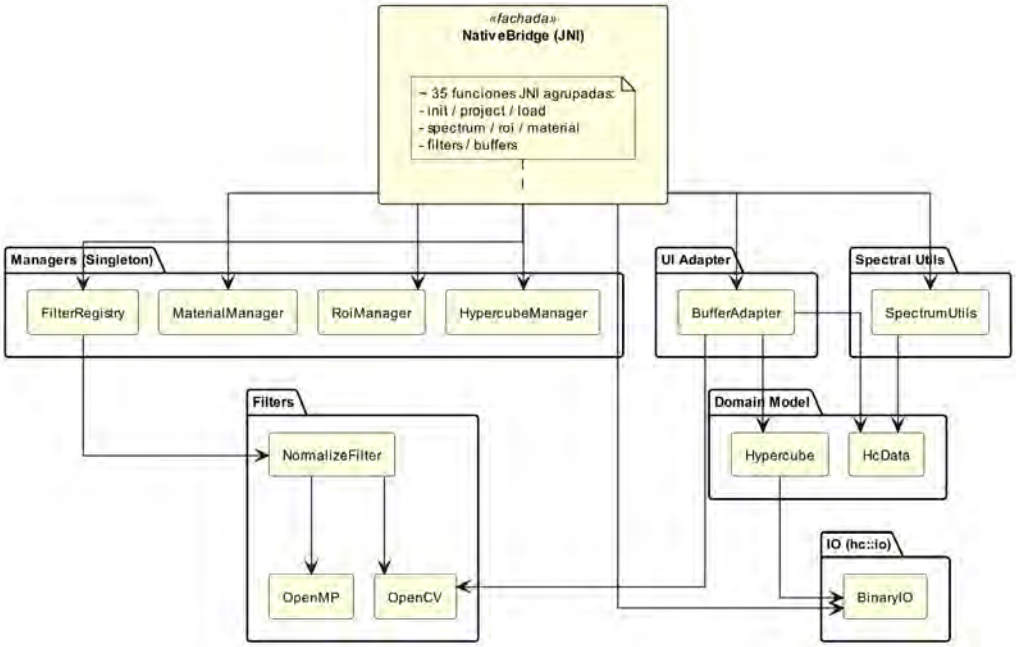

Figura 5.12: Diagrama de dependencias de la capa JNI y utilidades.

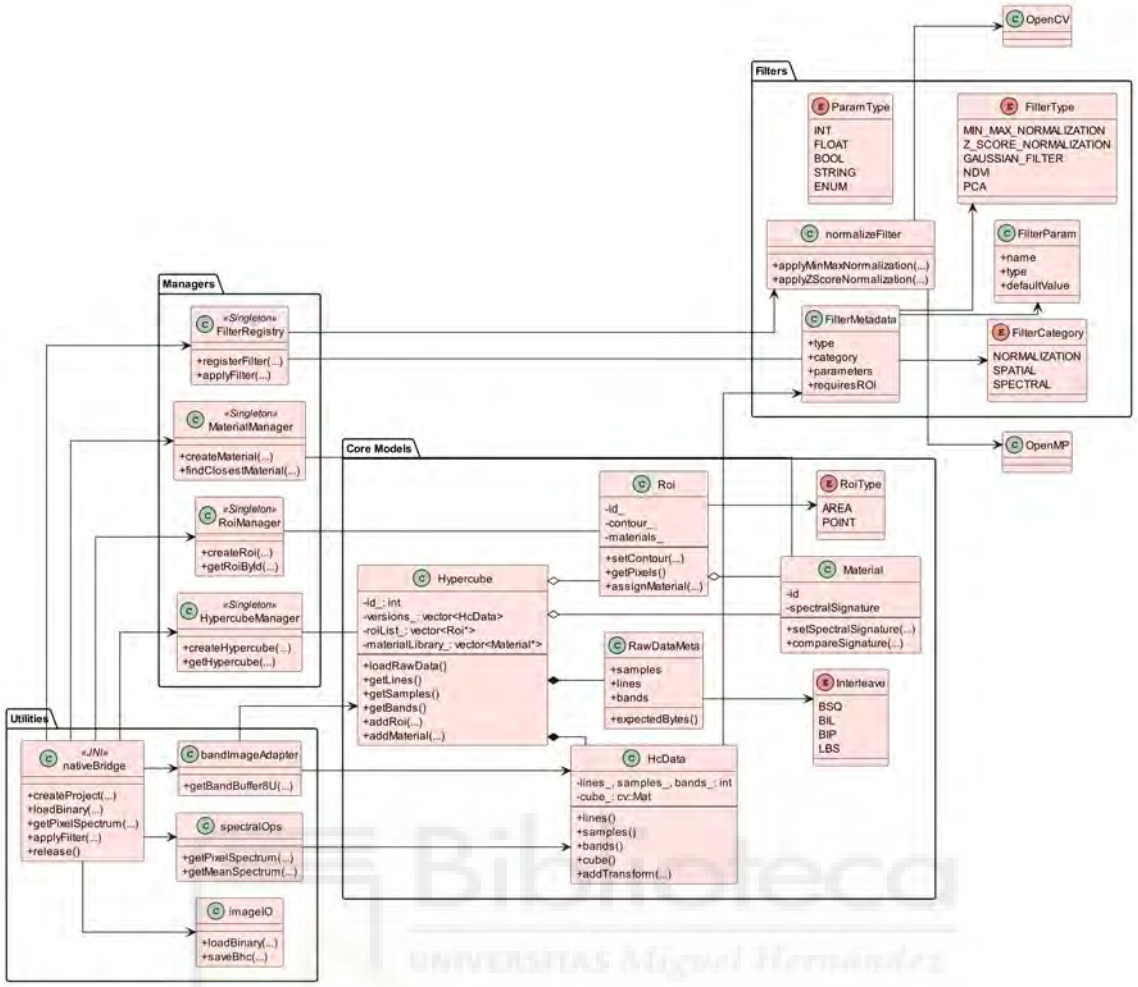

Figura 5.13: Diagrama relacional completo de la librería `Hypercube_native`.

Descripción de las relaciones

El diagrama de clases de la librería `Hypercube_native` muestra una arquitectura basada en el patrón Singleton para los gestores principales:

- **HypercubeManager**: Gestiona el ciclo de vida de todos los hipercubos en memoria. Mantiene un mapa que asocia IDs únicos con punteros a objetos `Hypercube`.
- **RoiManager**: Similar al anterior, gestiona las regiones de interés de forma global, aunque cada ROI también está asociada a un hipercubo específico.
- **MaterialManager**: Gestiona la biblioteca global de materiales espectrales, permitiendo su reutilización entre diferentes proyectos e hipercubos.
- **FilterRegistry**: Registro central de todos los filtros disponibles en el sistema, con sus metadatos y las funciones de procesamiento asociadas.

La clase `Hypercube` actúa como agregador principal:

- Composición fuerte con `HcData` (las versiones son parte del hipercubo)
- Referencias a `Roi` y `Material` (los ROIs y materiales pueden existir independientemente)
- Composición con `RawDataMeta` (metadatos intrínsecos del hipercubo)

La clase `HcData` encapsula los datos tridimensionales mediante `cv::Mat` de OpenCV y mantiene un historial de transformaciones aplicadas.

La fachada JNI (`nativeBridge`) actúa como punto único de entrada desde Kotlin, orquestando las llamadas a los diferentes gestores y utilidades, y realizando la conversión de tipos entre ambos mundos.

5.4. Casos de uso del sistema

Esta sección describe los casos de uso principales del sistema. Cada caso de uso se presenta en formato tabular siguiendo la estructura estándar propuesta por Cockburn [28]: identificador, nombre, actores, precondiciones, flujo principal, flujos alternativos y postcondiciones.

Para definir el alcance funcional de la aplicación y las interacciones del usuario con el sistema, se ha diseñado el diagrama de casos de uso general (Figura 5.14). Los casos de uso se han agrupado conceptualmente en cuatro paquetes funcionales: Gestión de Proyecto, Regiones de Interés (ROIs), Materiales y Procesamiento espectral.

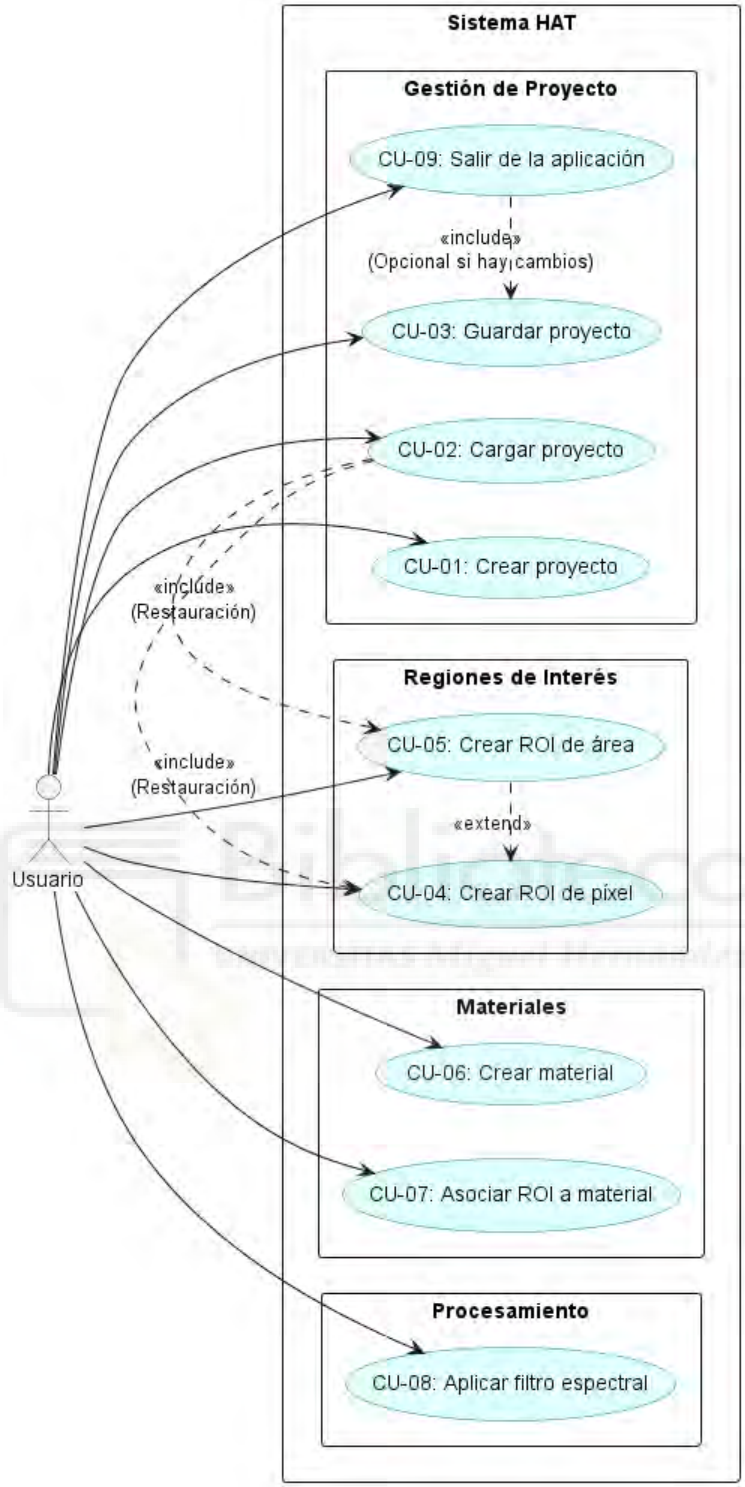

Figura 5.14: Diagrama de casos de uso general del sistema HAT.

Como se observa en el diagrama, existen relaciones de inclusión (`«include»`) entre casos de uso, como la necesidad implícita de cargar los datos auxiliares al restaurar un proyecto, o el guardado de seguridad opcional antes de salir de la aplicación.

Tabla 5.1: Caso de uso CU-01: Crear proyecto.

ID	CU-01
Nombre	Crear proyecto
Actor	Usuario
Precondiciones	Existe un archivo `.bin` de imagen hiperespectral accesible en disco.
Flujo principal	1. El usuario selecciona "Nuevo Proyecto" en el menú. 2. El sistema muestra un diálogo con campos para nombre del proyecto, selección del archivo `.bin` y directorio de destino. 3. El usuario completa los campos y confirma. 4. El sistema cierra el proyecto anterior (si existe). 5. `ProjectManager` crea la estructura de directorios en disco (`hypercubes/`, `meta/`, `processed/`). 6. `ProjectIO` copia el `.bin` original a `hypercubes/nombre.bhc`. 7. `NativeBridge.createProject()` carga el binario en C++ y retorna un `runtimeId`. 8. `HypercubeManager` registra el hipercubo en caché. 9. El sistema abre el hipercubo en una pestaña nueva.
Flujos alternativos	3a. El archivo `.bin` no es válido: el sistema muestra un error y cancela la operación. 5a. No hay espacio en disco: el sistema informa del error.
Postcondiciones	El proyecto está creado en disco con su archivo `.hc`, el hipercubo está cargado en memoria y visible en la interfaz.

Tabla 5.2: Caso de uso CU-02: Cargar proyecto.

ID	CU-02
Nombre	Cargar proyecto existente
Actor	Usuario
Precondiciones	Existe un archivo `.hc` válido en disco.
Flujo principal	1. El usuario selecciona "Abrir Proyecto" o arrastra un archivo `.hc` a la ventana. 2. El sistema cierra el proyecto anterior (si existe). 3. `HatSolutionParser` parsea el archivo `.hc` de forma *lazy* (sin cargar binarios). 4. `ProjectManager` crea el `ProjectContext` y puebla la lista de ítems de la sidebar. 5. El usuario hace doble clic sobre un ítem de la sidebar. 6. `HypercubeManager` carga el binario `.bhc` vía `NativeBridge.loadBinary()`. 7. `ProjectIO` restaura los metadatos auxiliares (ROIs, materiales, versiones) desde los archivos JSON. 8. El sistema abre el hipercubo en una pestaña.
Flujos alternativos	1a. El archivo `.hc` no se puede parsear: el sistema muestra un error. 6a. El archivo `.bhc` referenciado no existe: el sistema informa que falta el binario.
Postcondiciones	El proyecto está abierto con su estructura visible en la sidebar. Los hipercubos seleccionados están cargados y visibles.

Tabla 5.3: Caso de uso CU-03: Guardar proyecto.

ID	CU-03
Nombre	Guardar proyecto
Actor	Usuario
Precondiciones	Existe un proyecto abierto con al menos un hipercubo cargado.
Flujo principal	1. El usuario selecciona "Guardar" en el menú. 2. Si el proyecto no tiene archivo `.hc` asociado, el sistema muestra un diálogo "Guardar Como". 3. `ProjectManager` delega a `ProjectIO.saveProject()`. 4. El sistema serializa el archivo índice `.hc`. 5. Para cada hipercubo marcado como *dirty*, el sistema serializa los JSONs de metadatos (ROIs, materiales, transformaciones, versiones) en `meta/{uid}/`. 6. Se desactivan los flags *dirty* de todos los ítems.
Flujos alternativos	2a. El usuario cancela el diálogo: la operación se aborta sin cambios.
Postcondiciones	El proyecto completo está persistido en disco. Los flags de cambios sin guardar están desactivados.

Tabla 5.4: Caso de uso CU-04: Crear ROI de píxel.

ID	CU-04
Nombre	Crear ROI de píxel
Actor	Usuario
Precondiciones	Hay un hipercubo cargado y visible. El modo de selección está en `POINT`.
Flujo principal	1. El usuario hace clic sobre un píxel de la imagen. 2. `CoordinateUtils` convierte las coordenadas de pantalla a coordenadas de imagen (fila, columna). 3. `Hypercube.createPixelRoi()` invoca `NativeBridge.createPointRoi()`. 4. C++ crea un objeto `Roi` de tipo `POINT` y lo registra en `RoiManager` y en el hipercubo. 5. Kotlin recibe el `roiId` y crea un `RoiData` correspondiente. 6. El espectro del píxel se muestra en el gráfico espectral. 7. El hipercubo se marca como *dirty*.
Flujos alternativos	2a. Las coordenadas están fuera del rango: el sistema ignora el clic.
Postcondiciones	El ROI aparece en la lista de ROIs y su espectro se visualiza en el gráfico.

Tabla 5.5: Caso de uso CU-05: Crear ROI de área (rectángulo, polígono o mano alzada).

ID	CU-05
Nombre	Crear ROI de área
Actor	Usuario
Precondiciones	Hay un hipercubo cargado. El modo de selección está en `RECTANGLE`, `POLYGON` o `FREEHAND`.
Flujo principal	1. El usuario dibuja una región sobre la imagen (arrastre para rectángulo/mano alzada, clics sucesivos para polígono). 2. `RoiDrawingLayer` captura los puntos en coordenadas de pantalla. 3. Al finalizar el dibujo, `CoordinateUtils` convierte todos los puntos a coordenadas de imagen. 4. Los puntos se serializan como `IntArray` alternando (x, y). 5. `Hypercube.createRoi()` invoca `NativeBridge.createRoi()`. 6. C++ crea un `Roi` de tipo `AREA` con el contorno, calcula los píxeles internos mediante `cv::fillPoly` y lo registra. 7. Kotlin recibe el `roiId`, crea el `RoiData` y solicita el espectro medio. 8. El espectro medio se añade al gráfico espectral.
Flujos alternativos	6a. El contorno tiene menos de 3 puntos (para polígono): C++ retorna -1 y Kotlin muestra un aviso.
Postcondiciones	El ROI aparece en la lista, su contorno se superpone sobre la imagen y su espectro medio se visualiza.

Tabla 5.6: Caso de uso CU-06: Crear material.

ID	CU-06
Nombre	Crear material
Actor	Usuario
Precondiciones	Hay un hipercubo cargado.
Flujo principal	1. El usuario accede a la pestaña de materiales en el panel de análisis. 2. El usuario introduce nombre, descripción y selecciona un color. 3. `NativeBridge.createMaterial()` crea el objeto en C++ y retorna un `materialId`. 4. Kotlin registra el `MaterialData` en el hipercubo. 5. El material aparece en la lista de materiales disponibles.
Flujos alternativos	Ninguno.
Postcondiciones	El material existe tanto en C++ como en Kotlin y está disponible para asignación.

Tabla 5.7: Caso de uso CU-07: Asociar ROI a material.

ID	CU-07
Nombre	Asociar ROI a material
Actor	Usuario
Precondiciones	Existe al menos un ROI y un material creados.
Flujo principal	1. El usuario selecciona un ROI y pulsa "Asignar Material". 2. El sistema muestra un diálogo modal con la lista de materiales disponibles. 3. El usuario selecciona un material y confirma. 4. `NativeBridge.assignMaterialToRoi()` vincula el ROI al material en C++. 5. Kotlin actualiza el `materialId` del `RoiData`. 6. El color del ROI en la visualización cambia al color del material asignado.
Flujos alternativos	3a. El usuario cancela el diálogo: no se realiza ninguna asignación.
Postcondiciones	El ROI tiene un material asignado tanto en C++ como en Kotlin. La visualización refleja la asociación.

Tabla 5.8: Caso de uso CU-08: Aplicar función de filtro espectral.

ID	CU-08
Nombre	Aplicar función de filtro espectral
Actor	Usuario
Precondiciones	Hay un hipercubo cargado con al menos una versión de datos.
Flujo principal	1. El usuario accede a la pestaña "Funciones" del panel de análisis. 2. El sistema muestra la lista de funciones de procesamientos y filtros disponibles (obtenida de `FilterProvider`). 3. El usuario selecciona una funcion de procesamiento o filtro y configura sus parámetros. 4. El usuario pulsa "Aplicar". 5. `FilterViewModel` invoca `NativeBridge.applyFilter()` con el `hypercubeId`, índice de versión fuente, identificador del procesamiento o filtro y los parámetros como arrays paralelos de claves y valores. 6. C++ obtiene la versión fuente, invoca `FilterRegistry.applyFilter()` que ejecuta la función registrada, y añade el resultado como nueva versión procesada. 7. Kotlin ejecuta `Hypercube.syncAfterFilter()` para sincronizar la lista de versiones. 8. El sistema cambia automáticamente a la versión recién creada. 9. El hipercubo se marca como *dirty*.
Flujos alternativos	6a. El filtro falla (parámetros inválidos, error numérico): C++ retorna -1 y Kotlin muestra un error.
Postcondiciones	La nueva versión procesada existe en C++ y en Kotlin. El visor muestra el gráfico espectral de los rois transformados.

Tabla 5.9: Caso de uso CU-09: Salir de la aplicación.

ID	CU-09
Nombre	Salir de la aplicación
Actor	Usuario
Precondiciones	La aplicación está en ejecución.
Flujo principal	1. El usuario solicita salir (menú, atajo o botón de cierre de ventana). 2. El sistema comprueba si hay cambios sin guardar (`hasUnsavedChanges()`). 3. Si hay cambios, se muestra un diálogo de confirmación con opciones: Guardar, No guardar, Cancelar. 4. `AppState.dispose()` ejecuta el apagado ordenado: *a*) Cancela las corrutinas activas. *b*) Cierra pestañas y libera el estado de UI. *c*) `HypercubeManager.clearAll()` libera los objetos Kotlin. *d*) `NativeBridge.release()` libera todos los recursos nativos (memoria C++, objetos OpenCV). 5. La aplicación finaliza.
Flujos alternativos	3a. El usuario selecciona "Guardar": se ejecuta CU-03 antes de continuar. 3b. El usuario selecciona "Cancelar": la operación se aborta y la aplicación permanece abierta.
Postcondiciones	Todos los recursos (memoria nativa, hilos, archivos) están liberados. El proceso ha finalizado.

5.5. Diagramas de secuencia

Para documentar y comprender el comportamiento dinámico del sistema y la comunicación entre sus distintas capas arquitectónicas, se han elaborado diagramas de secuencia correspondientes a los casos de uso principales.

Estos diagramas respetan una convención de colores estructurada por capa tecnológica:

- **Azul:** Componentes de la interfaz de usuario y presentación gráfica (Compose).

- **Verde:** Lógica de dominio y gestión de estado en Kotlin (*Managers*).
- **Amarillo:** Capa de interoperabilidad y puente de comunicación (JNI).
- **Rojo:** Motor nativo de procesamiento y estructuras en C++.

A continuación, se describen los flujos de ejecución más relevantes.

5.5.1. CU-01: Creación de un nuevo proyecto (DS-01)

La creación de un proyecto implica configurar la estructura persistente en disco y reservar la memoria nativa. Como se aprecia en la Figura 5.15, el flujo se divide para no bloquear la interfaz: mientras el orquestador (`AppState`) muestra el estado de carga en el hilo principal, el `ProjectManager` opera en un hilo secundario (E/S) creando la jerarquía de carpetas mediante `ProjectIO`. Una vez copiado el binario original, se invoca a C++ a través de JNI para cargar los datos crudos (*RAW*) y devolver el identificador de ejecución a Kotlin.

Figura 5.15: Diagrama de secuencia DS-01

5.5.2. CU-02: Carga de un proyecto existente (DS-02)

La restauración de una sesión previa (`.hc`) requiere leer los descriptores JSON y reconstruir el estado de la aplicación (Figura 5.16). Inicialmente, se parsea el archivo de solución para cargar el árbol del proyecto en la interfaz. La memoria pesada en C++ no se reserva hasta que el usuario hace doble clic sobre un hipercubo específico, momento en el que el motor nativo lee el binario y `ProjectIO` se encarga de inyectar las Regiones de Interés (ROIs), materiales y versiones previas restauradas desde los metadatos JSON.

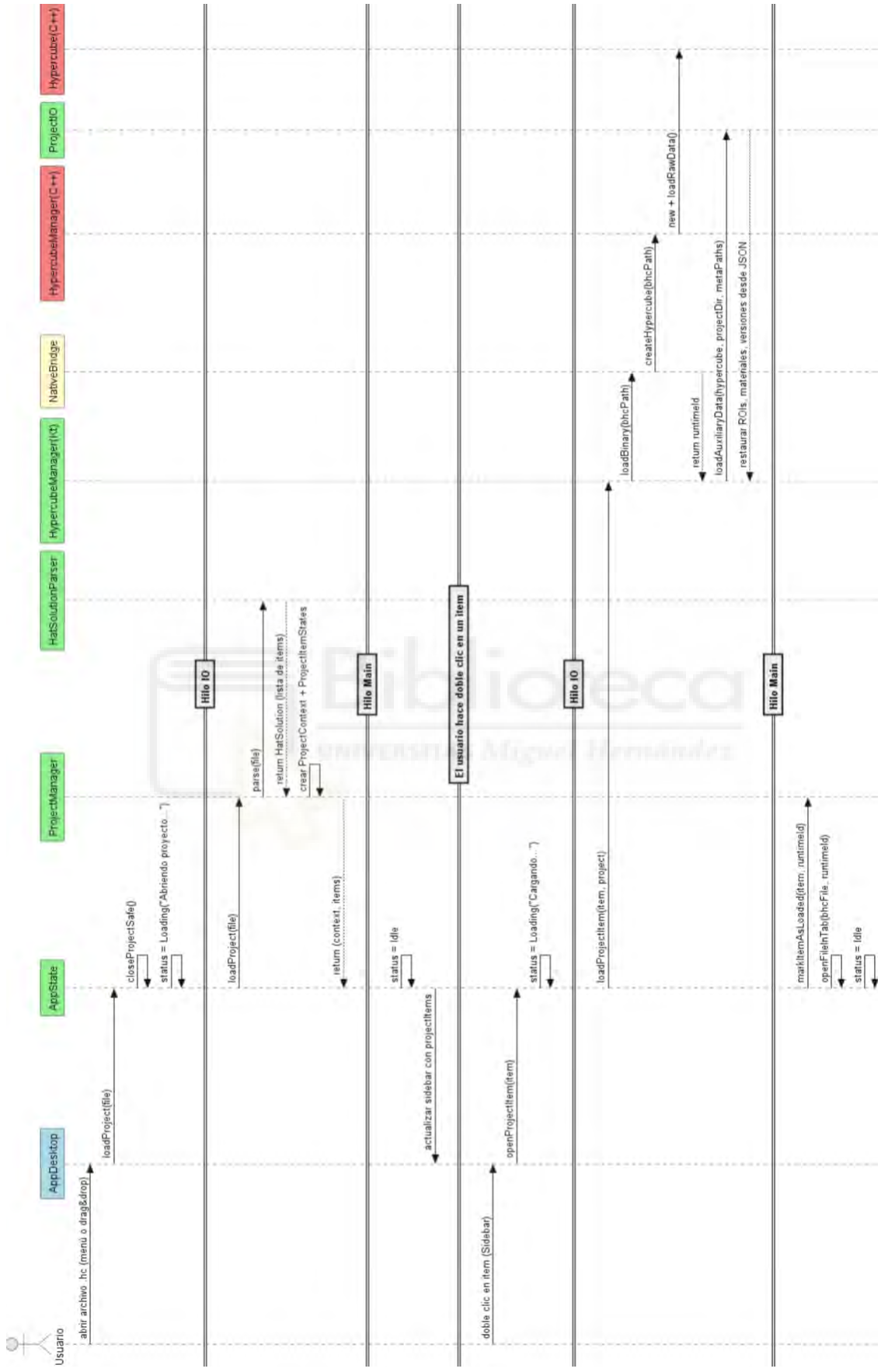

Figura 5.16: Diagrama de secuencia DS-02

5.5.3. CU-04: Creación de ROI de píxel (DS-03)

La selección de un píxel para análisis espectral (Figura 5.17) inicia con un evento táctil en `CentralImageViewer`. Las coordenadas de la pantalla (afectadas por *zoom* y *pan*) son traducidas a índices matriciales por `CoordinateUtils`. Estas coordenadas puras cruzan el puente JNI y C++ registra el punto, extrayendo instantáneamente la firma espectral. El vector numérico resultante se devuelve al Front-End para actualizar el gráfico `SpectrumChart`.

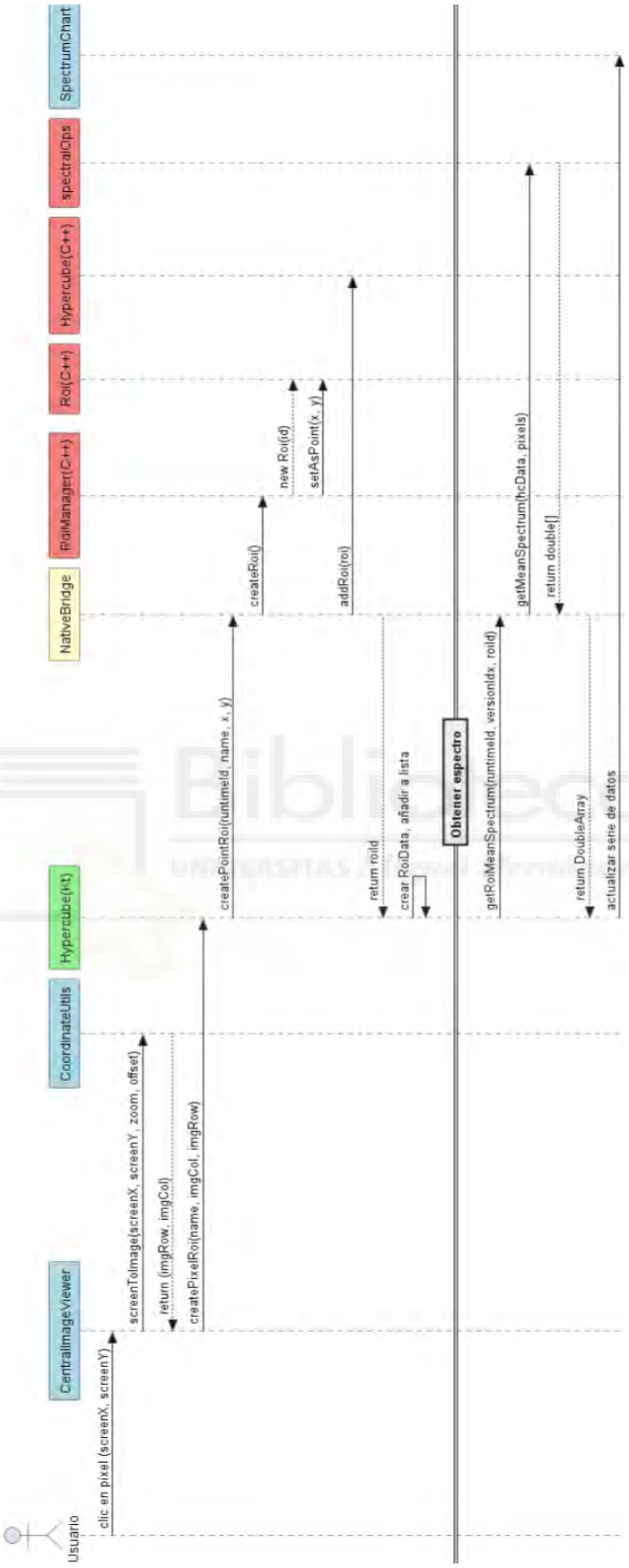

Figura 5.17: Diagrama de secuencia DS-03

5.5.4. CU-05: Creación de ROI de área (DS-04)

A diferencia del píxel individual, el trazado de un polígono (Figura 5.18) mantiene su estado transitorio en `RoiSelectionState` para renderizar un *overlay* visual mientras el usuario arrastra el ratón. Una vez finalizado el dibujo, el contorno entero se transfiere como un arreglo plano (`IntArray`) a C++. El motor nativo, utilizando funciones de OpenCV (`cv::fillPoly`), genera una máscara binaria interna y calcula estadísticamente la firma media de todos los píxeles abarcados.

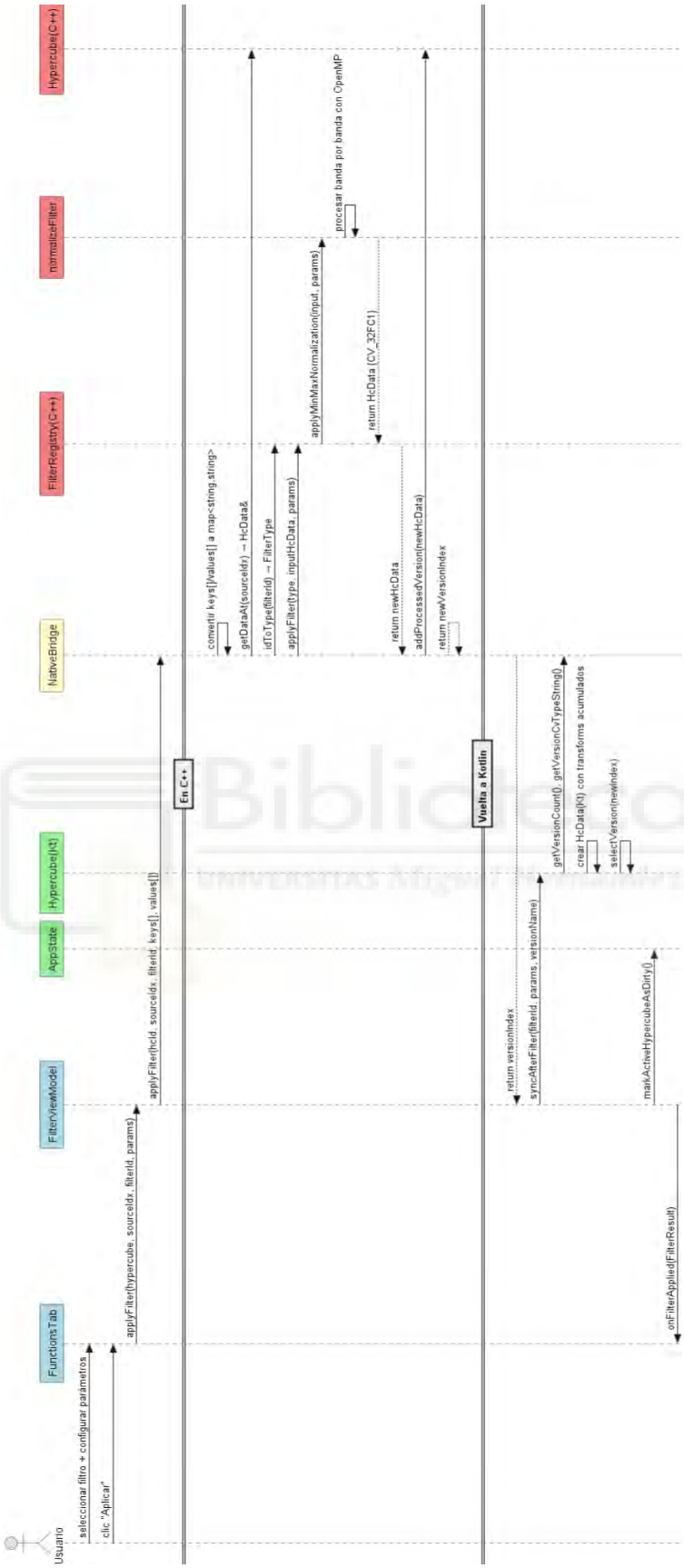

Figura 5.18: Diagrama de secuencia DS-04

5.5.5. CU-08: Aplicación de funciones de procesamientos de datos espectrales (DS-05)

El procesamiento de datos espectrales es el caso de uso más crítico en términos de rendimiento computacional. Como se detalla en la Figura 5.19, los parámetros configurados por el usuario se serializan en pares de arreglos (`keys[]` y `values[]`) hacia C++. El módulo `FilterRegistry` dirige la petición al algoritmo correspondiente (ej. normalización), el cual utiliza OpenMP para paralelizar iteraciones espaciales. Finalmente, se genera una nueva versión `HcData` sin destruir el original, y Kotlin actualiza la vista activa de la aplicación.

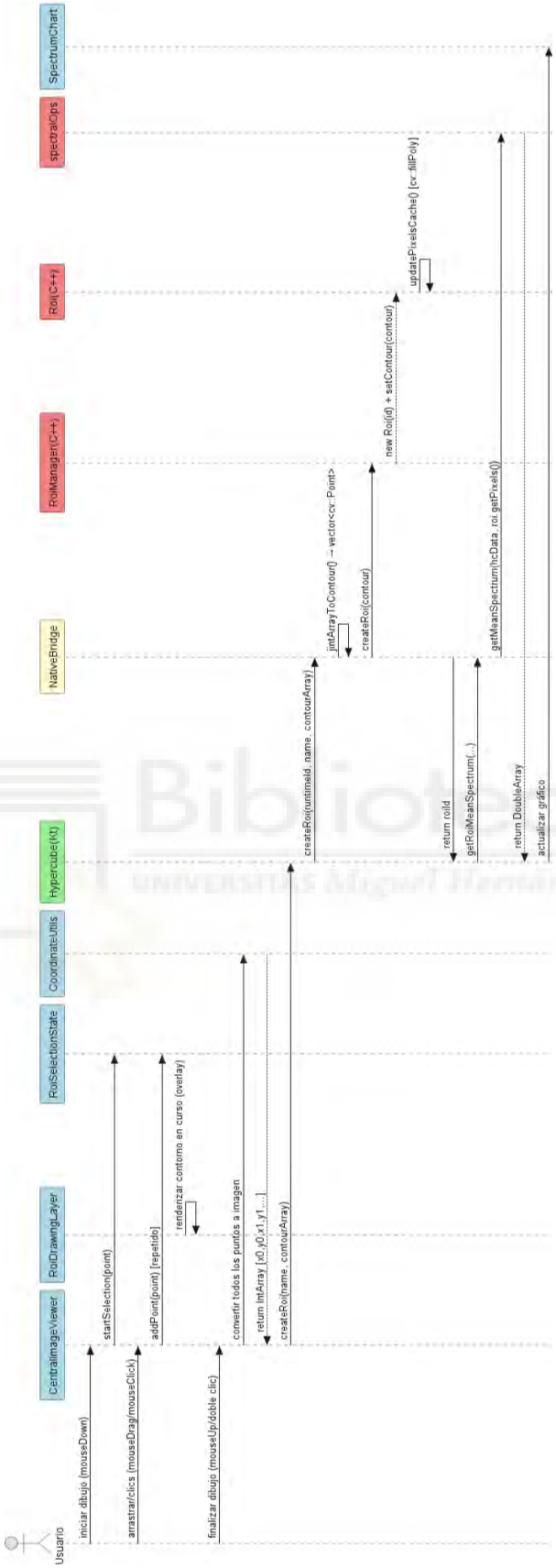

Figura 5.19: Diagrama de secuencia DS-05

5.5.6. CU-03: Guardado de proyecto (DS-06)

Garantizar la persistencia es fundamental para la reproducibilidad. En la Figura 5.20 se muestra cómo, mediante un bucle de comprobación sobre los hipercubos marcados como modificados (*dirty*), la aplicación serializa únicamente los metadatos y transformaciones (ROIs, historial de funciones de procesamientos y filtros aplicados, paleta de colores). El binario pesado jamás se reescribe, cumpliendo con la premisa de edición no destructiva.

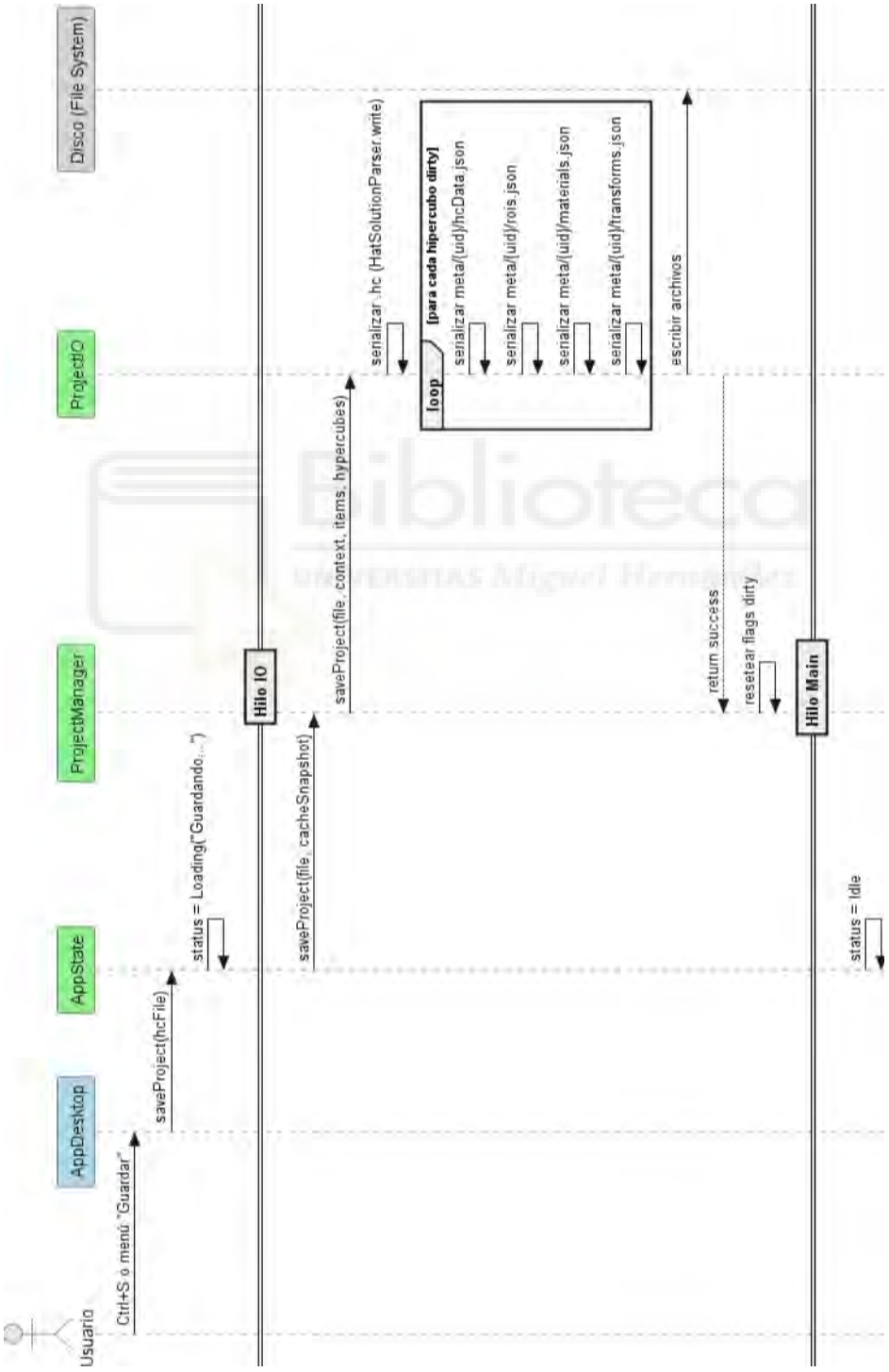

Figura 5.20: Diagrama de secuencia DS-06

5.5.7. CU-09: Salida del sistema (DS-07)

El apagado seguro de la aplicación (Figura 5.21) es crítico para prevenir fugas de memoria nativa. Si hay cambios pendientes, la UI solicita confirmación. En caso de continuar, `AppState.dispose()` interrumpe todas las corrutinas, ordena a cada `HypercubeManager` el borrado controlado de las instancias activas y notifica al puente JNI para que C++ destruya la memoria reservada por OpenCV antes de finalizar el proceso del sistema operativo.

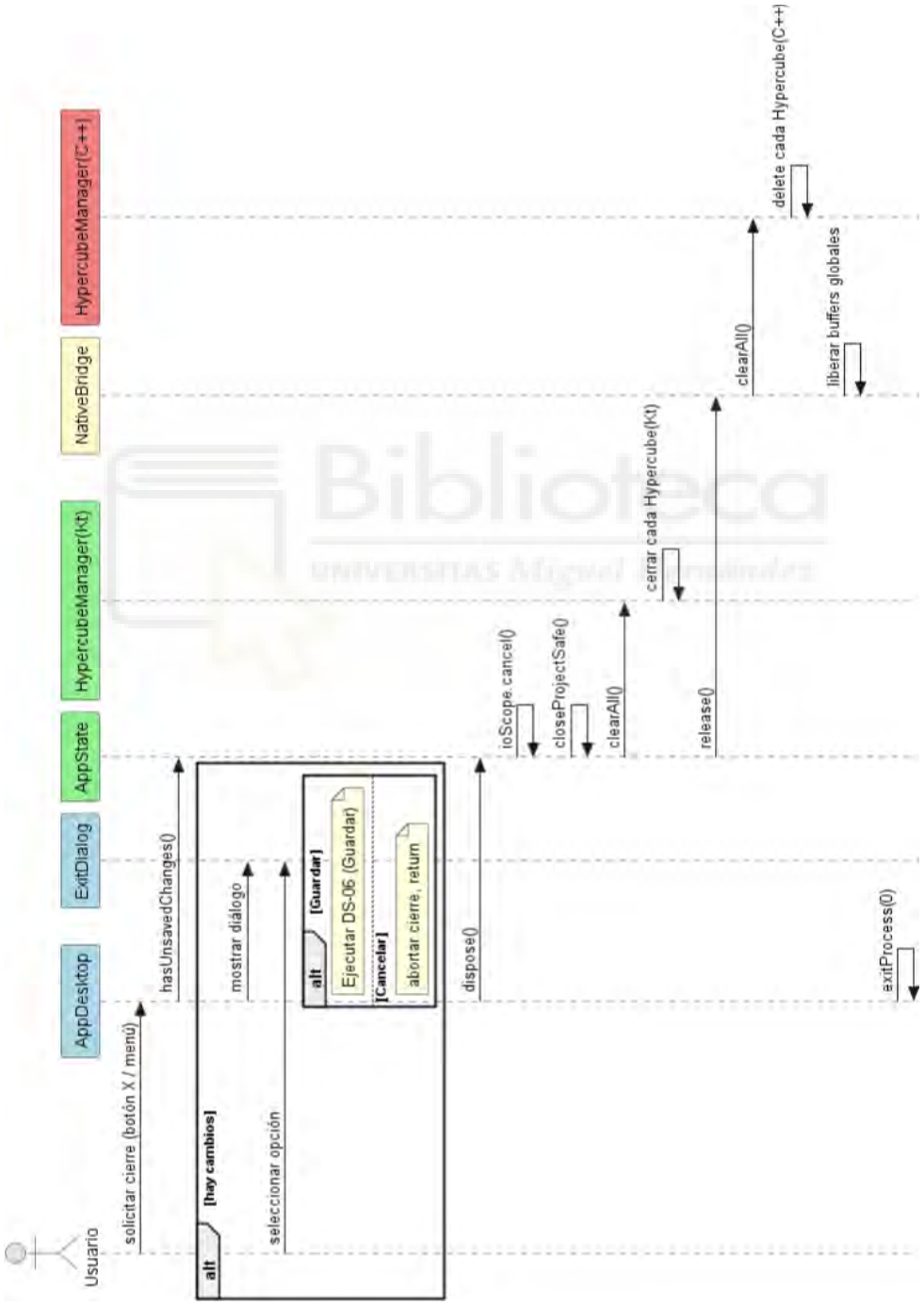

Figura 5.21: Diagrama de secuencia DS-07

6. Implementación

En este capítulo se detalla la implementación técnica del sistema, dejando a un lado la enumeración exhaustiva de los directorios del proyecto para centrarse en los retos arquitectónicos y las soluciones ingenieriles adoptadas. El desarrollo de una aplicación capaz de manejar hipercubos de datos masivos exige un control estricto de la memoria, una sincronización eficiente entre lenguajes de programación y un renderizado gráfico optimizado.

A continuación, se describen los mecanismos desarrollados para la orquestación del estado de la interfaz, el motor de procesamiento en C++, la interoperabilidad a través de JNI y el sistema de persistencia de proyectos.

6.1. Implementación del Front-End (Kotlin y Compose)

Esta capa gestiona la interfaz gráfica y la lógica de presentación. El reto principal ha sido mantener una interfaz fluida mientras se manejan grandes volúmenes de datos.

6.1.1. Gestión del Estado y Orquestación (Front-End)

El entorno de interfaz gráfica ha sido desarrollado en Kotlin empleando Compose Multiplatform. A diferencia de los paradigmas imperativos tradicionales, Compose utiliza un enfoque declarativo y reactivo. Por tanto, la arquitectura de la aplicación se ha diseñado basándose en el principio de *Single Source of Truth* (Única Fuente de la Verdad).

Para evitar el acoplamiento entre la lógica de negocio y las vistas, se ha diseñado un orquestador central denominado `AppState`, el cual actúa como el nodo raíz del árbol de dependencias. Este estado global delega las operaciones específicas en gestores de dominio o *Managers*:

- `ProjectManager:` Gestiona el ciclo de vida del proyecto actual (`ProjectContext`). Mantiene de forma reactiva el árbol de archivos disponibles para el explorador lateral y actúa como intermediario con la capa de persistencia en disco.
- `HypercubeManager:` Es el responsable del ciclo de vida de los datos pesados en memoria. Posee una caché que asocia los identificadores en tiempo de ejecución (*runtimeId*) con los objetos de dominio `Hypercube`. Su responsabilidad crítica es garantizar que, al cerrar un archivo o el proyecto entero, se emitan las órdenes de liberación de memoria correspondientes hacia el motor nativo para prevenir fugas de memoria (*memory leaks*).

Mediante el uso de propiedades delegadas como `mutableStateOf` y colecciones reactivas (`mutableStateListOf`), cualquier mutación en los modelos de datos producida

por el usuario o por la finalización de un algoritmo en C++, se propaga automáticamente desencadenando una recomposición únicamente en los componentes visuales afectados.

6.1.2. Interfaz de Usuario y Renderizado Avanzado

Uno de los principales desafíos técnicos en la visualización de imágenes hiperespectrales es la traducción eficiente de arreglos de datos puros y multidimensionales a formatos legibles por la unidad de procesamiento gráfico (GPU) sin bloquear el hilo principal de la interfaz de usuario.

En el desarrollo de interfaces de usuario modernas, existe un hilo de ejecución dedicado exclusivamente a capturar eventos táctiles y dibujar la pantalla, conocido como "Hilo Principal"(*Main Thread*) [29]. Si una operación matemática compleja bloquea este hilo, la aplicación se congela y deja de responder.

Para evitar este bloqueo al renderizar los hipercubos, la lectura de la memoria C++ se realiza de forma asíncrona. El *buffer* resultante se inyecta en un objeto `ByteBuffer` y se transforma en un `ImageBitmap` utilizando Skia [30], un motor gráfico 2D de alto rendimiento que actúa como base de renderizado para Compose Multiplatform.

6.1.3. Traducción de Memoria a Imagen (`ImageUtils`)

El motor nativo (C++) extrae la banda espectral seleccionada y la procesa reduciendo su profundidad mediante un escalado min-max, devolviendo un *buffer* continuo de 8 bits por píxel hacia Kotlin. Para renderizar estos datos en Compose sin sobrecargar el recolector de basura (*Garbage Collector*) de la Máquina Virtual de Java (JVM), se ha implementado la función `makeGrayImageFromBufferContinous`.

Esta función consume directamente la memoria nativa mediante un objeto `ByteBuffer` e inyecta los valores en un arreglo preasignado, configurando los canales RGBA donde el nivel de gris se replica en los canales RGB y el canal Alpha se satura. Posteriormente, este arreglo se transforma en un `ImageBitmap` de Skia (el motor de renderizado subyacente de Compose), garantizando una tasa de refresco óptima al cambiar de banda espectral.

6.1.4. Cálculo de Viewport e Interacción (`ViewportUtils`)

Para ofrecer herramientas analíticas precisas, como la selección de píxeles o regiones de interés (ROIs), el sistema debe ser capaz de traducir la coordenada de un clic del ratón en la pantalla a las coordenadas matriciales exactas del hipercubo.

Esta operación es compleja debido a que la imagen puede estar escalada (zoom) y desplazada (*pan*). A través del módulo `ViewportUtils`, la aplicación extrae el

desplazamiento *offset* y el factor de escala de la interfaz, aplicando una transformación afín inversa para proyectar el clic de la pantalla al *box* original y, finalmente, interpolar dicho punto al índice real del *array* de píxeles original del hipercubo.

6.1.5. Integración con el Sistema de Ventanas (Drag and Drop)

A pesar de las virtudes multiplataforma de Compose, la gestión de eventos de arrastrar y soltar archivos (*Drag and Drop*) desde el sistema operativo hacia la aplicación carece de madurez en plataformas de escritorio.

Para sortear esta limitación, se implementó el módulo `WindowDropTargetManager`, el cual realiza una inyección directa sobre la capa *Java AWT* de la ventana gráfica. Mediante el uso de un `GlassPane` invisible superpuesto sobre el marco de la aplicación, el sistema intercepta los eventos `DropTargetDropEvent` del sistema operativo (Windows o Linux), lee los descriptores URI de los archivos arrastrados, los valida y los canaliza de vuelta al entorno reactivo de Compose, logrando una experiencia de usuario fluida y nativa.

6.1.6. Capa de Dominio y Control de Memoria (Patrón Proxy)

Dado que los hipercubos pueden ocupar gigabytes de memoria RAM, cargar estos datos directamente en el *heap* de la Máquina Virtual de Java (JVM) provocaría paradas críticas por la recolección de basura (*Garbage Collection pause*) [31]. Por ello, la memoria pesada se mantiene estrictamente en la capa nativa (C++).

Para manipular estos datos desde Kotlin sin comprometer el rendimiento, se ha implementado el **Patrón Estructural Proxy** [7]. La clase `Hypercube.kt` actúa como un representante ligero del objeto real alojado en C++, enlazándose mediante un identificador de tiempo de ejecución (`runtimeId`).

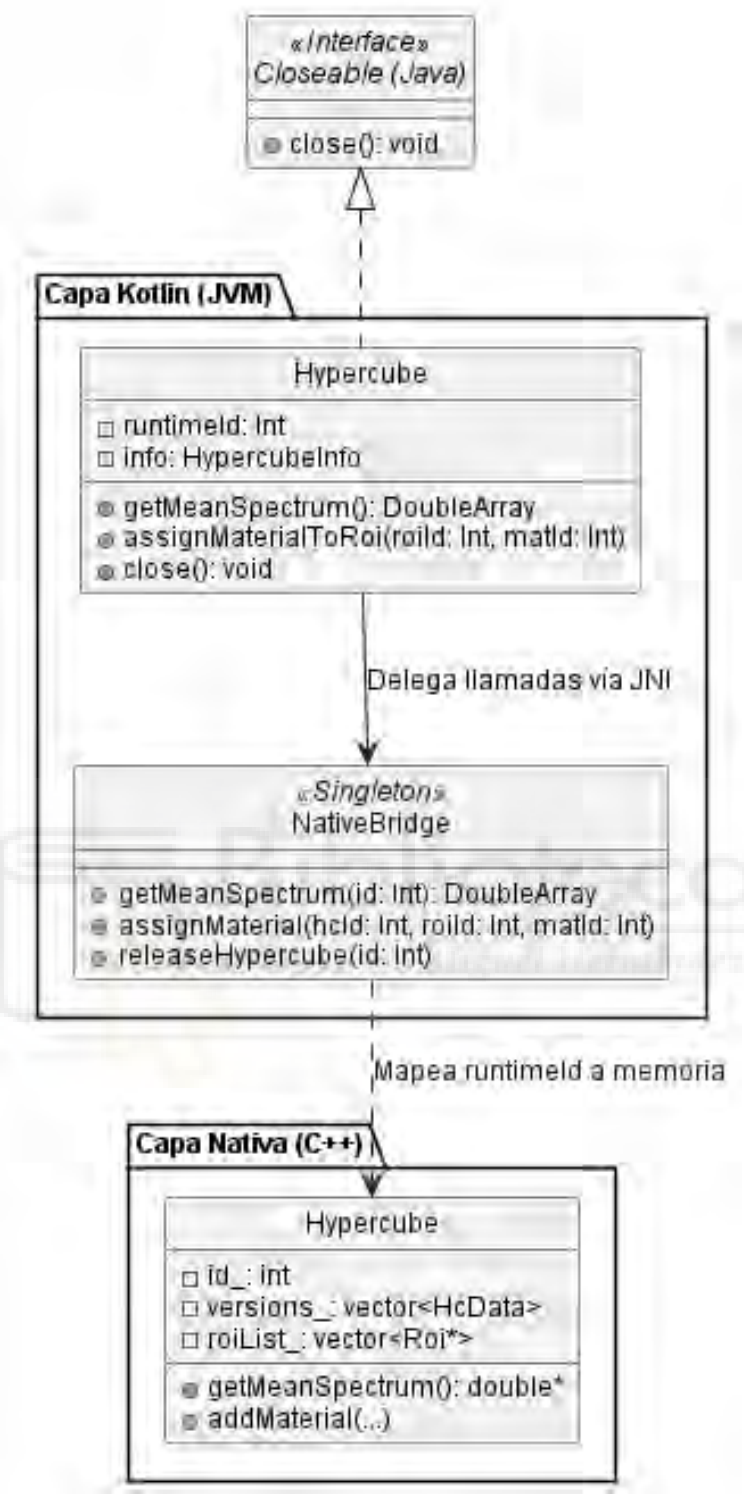

Figura 6.1: Diagrama de clases simplificado mostrando la implementación del Patrón Proxy.

Para garantizar la liberación de la memoria nativa y evitar fugas de memoria (*memory leaks*), la clase proxy implementa la interfaz estándar `Closeable`. Cuando el gestor decide descargar un hipercubo, invoca este método, el cual transmite la orden de destrucción a C++ a través de JNI, tal y como se observa en el Código 1.

```
override fun close() {
```

```
    if (isClosed) return
    isClosed = true
    try {
        NativeBridge.releaseHypercube(runtimeId)
        println("Hypercube cerrado: ${info.name}")
    } catch (e: Exception) {
        println("Error al cerrar Hypercube: ${e.message}")
    }
}
```

Listing 1: Implementación del cierre seguro en la clase proxy `Hypercube`.

6.2. Capa de Interoperabilidad (JNI)

La clase `NativeBridge` actúa como frontera estricta entre Kotlin y C++. JNI impone convenciones de nombrado muy específicas para que la Máquina Virtual de Java sea capaz de enlazar una función declarada en Kotlin con su implementación compilada en el binario nativo.

A modo de ejemplo, en el Código 2 se muestra cómo se declara la función encargada de eliminar un ROI en el archivo Kotlin. La palabra reservada `external` indica que su implementación no existe en el código fuente de la JVM.

```
object NativeBridge {
    // ...
    external fun deleteRoi(hypercubeId: Int, roiId: Int): Boolean
}
```

Listing 2: Declaración de una función nativa en Kotlin (`NativeBridge.kt`).

En contraparte, el Código 3 muestra la firma requerida en C++. El nombre de la función debe comenzar por `Java_`, seguido de la ruta completa del paquete, el nombre de la clase (`NativeBridge`) y el nombre del método (`deleteRoi`). Además, JNI inyecta automáticamente dos parámetros iniciales (`env` y `obj`) antes de los argumentos definidos por el usuario.

```
extern "C" JNIEXPORT jboolean JNICALL
Java_es_umh_tfg_luisdavid_core_bridge_NativeBridge_deleteRoi(
    JNIEnv* env, jobject /*obj*/, jint hypercubeId, jint roiId)
{
    // Lógica en C++ para eliminar el ROI de memoria
    return getRoiManager().deleteRoi(hypercubeId, roiId) ?
   ↪ JNI_TRUE : JNI_FALSE;
}
```

Listing 3: Implementación de la firma equivalente en C++ (`nativeBridge.cpp`).

Este embudo de comunicación también facilita la implementación del Patrón Proxy, donde las clases de dominio en Kotlin delegan la gestión de memoria pesada a sus contrapartes en C++ mediante identificadores enteros (`runtimeId`).

Uno de los principales hitos de esta capa es el **descubrimiento dinámico de funciones de procesamiento y filtros**. En lugar de codificar (*hardcodear*) los menús

de procesamiento de la interfaz, Kotlin consulta a C++ mediante `NativeBridge` qué funciones de procesamiento y filtros están disponibles y qué parámetros requieren (véase `FilterProvider.kt`). Esta aproximación respeta el **Principio Abierto/Cerrado (Open/Closed Principle)** [32]: se pueden añadir nuevos algoritmos en C++ sin necesidad de recompilar o alterar el código del Front-End.

La siguiente tabla muestra las conversiones de tipos de JNI en C++ a Kotlin, basados en la especificación oficial de Oracle [33]:

Tipo Kotlin	**Tipo JNI (C++)**	**Uso en el Proyecto HAT**
`Int`	`jint`	Identificadores (`runtimeId`, `versionIndex`).
`Long`	`jlong`	Punteros a estructuras nativas o handles de recursos del motor C++.
`Boolean`	`jboolean`	Activación/desactivación de flags de procesamiento y opciones de ejecución.
`Float`	`jfloat`	Parámetros de normalización, escalado y configuración de filtros.
`Double`	`jdouble`	Cálculos numéricos de alta precisión en operaciones espectrales.
`String`	`jstring`	Rutas de archivos, nombres de filtros, parámetros dinámicos.
`IntArray`	`jintArray`	Envío masivo de coordenadas (vértices de ROIs poligonales).
`FloatArray`	`jfloatArray`	Kernels de filtrado y matrices de transformación.
`ByteArray`	`jbyteArray`	Transferencia de buffers de imagen en bruto.
`ByteBuffer`	`jobject` (DirectBuffer)	Recepción eficiente de la banda renderizada de 8 bits sin copia intermedia.
`Any`	`jobject`	Referencia genérica a objetos Java/Kotlin desde código nativo.
`Class<T>`	`jclass`	Acceso a metadatos de clase y lookup dinámico de métodos.
`Throwable`	`jthrowable`	Propagación y manejo de excepciones entre la capa nativa y JVM.
`Array<T>`	`jobjectArray`	Paso de colecciones de objetos entre capas.

Tabla 6.1: Mapeo ampliado de tipos de datos en la interfaz JNI.

6.3. Motor Nativo y Procesamiento (C++ y OpenCV)

El núcleo algorítmico de la aplicación, denominado `Hypercube_native`, está implementado en C++ (estándar C++17). Emplea la librería OpenCV [34] no para el procesamiento de imágenes convencionales, sino como un contenedor matricial eficiente (`cv::Mat`) para manejar la tridimensionalidad del hipercubo.

6.3.1. Envoltorio `HcData` y Trazabilidad

Para implementar un flujo de trabajo no destructivo, el motor no sobrescribe el cubo original en memoria. En su lugar, emplea la estructura `HcData`, que encapsula un `cv::Mat` junto con un historial de las transformaciones aplicadas. Cuando el usuario aplica, por ejemplo, una corrección radiométrica, el motor genera un nuevo `HcData` y lo apila en el vector de versiones del objeto `Hypercube` de C++, lo que permite la navegación entre las distintas etapas del procesamiento desde un menú desplegable en la interfaz (`VersionSelectorDropdown`).

6.3.2. Paralelización con OpenMP

Las operaciones espectrales requieren recorrer secuencialmente cientos de bandas por cada uno de los millones de píxeles espaciales. Para acelerar estos procesos, se ha integrado OpenMP (*Open Multi-Processing*) [35]. En algoritmos como la normalización espacial (Código 4), se emplea la directiva `#pragma omp parallel for collapse(2)` para dividir el cálculo de las filas y columnas espaciales entre los distintos hilos de hardware disponibles.

```
#pragma omp parallel for collapse(2)
for (int line = 0; line < lines; ++line) {
    for (int sample = 0; sample < samples; ++sample) {
        double sum = 0.0;
        for (int band = 0; band < bands; ++band) {
            long long idx = idxLBS(line, band, sample, bands,
   ↪ samples);
            sum += inputData[idx];
        }
        // ... calculo y asignacion en outputData ...
    }
}
```

Listing 4: Uso de OpenMP para la paralelización del filtrado en `normalizeFilter.cpp`.

6.4. Sistema de Persistencia de Proyectos

Garantizar la reproducibilidad de los análisis exige que el usuario pueda guardar y retomar su sesión de trabajo. El sistema de almacenamiento se ha diseñado basán-

dose en la inmutabilidad del archivo crudo: el fichero binario original del hipercubo (`.bhc`) jamás se modifica.

El módulo `ProjectIO` se encarga de serializar a formato JSON toda la información auxiliar empleando la librería `kotlinx-serialization`. La estructura de archivos resultante adopta un enfoque empaquetado, análogo al de los entornos de desarrollo integrados modernos:

```
MiProyecto_HAT/
├── MiProyecto.hc ... Archivo índice del
│   proyecto
├── hypercubes/
│   └── cubo_muestra.bhc ... Datos binarios
│       inmutables
└── meta/
    └── {UID_del_cubo}/
        ├── hcData.json ... Metadatos y árbol
        │   de versiones
        ├── rois.json ... Definición vectorial
        │   de máscaras
        └── materials.json ... Paleta de
            colores y firmas base
```

Figura 6.2: Estructura de directorios generada por el sistema de persistencia.

Esta jerarquía permite separar los metadatos de los datos brutos, facilitando la portabilidad del proyecto y evitando la corrupción accidental de la captura espectral originaria.

7. Resultados: Hyperspectral Analysis Tool (HAT)

7.1. Flujo de Inicio y Gestión de Proyectos

7.1.1. Pantalla de Inicio y Carga de Hipercubos

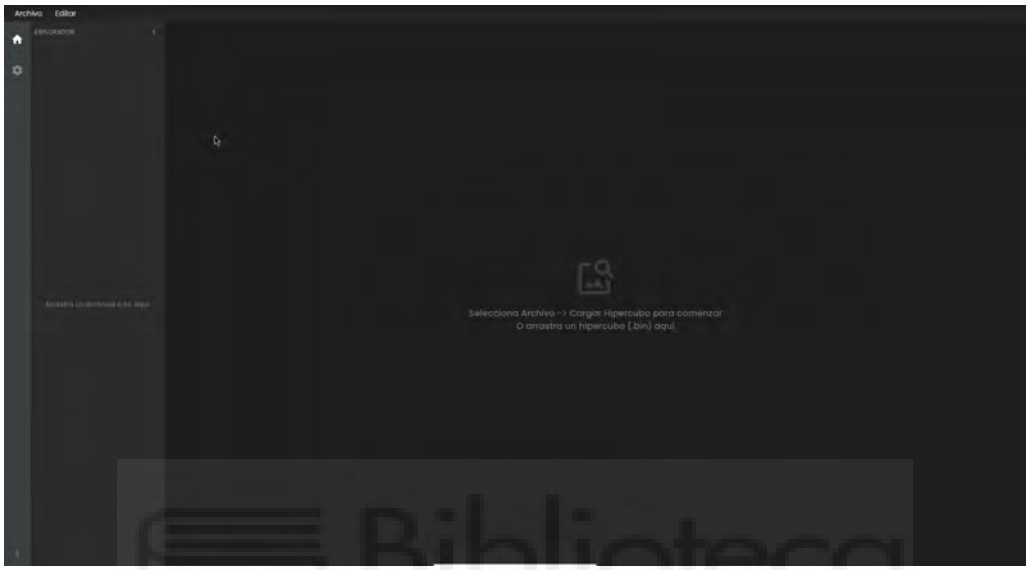

Figura 7.1: Pantalla de inicio de la aplicación. Se muestra el área central para arrastrar archivos y las indicaciones para cargar un hipercubo mediante el menú o drag&drop.

Figura 7.2: Menú desplegable “Archivo” mostrando las opciones principales: Nuevo Proyecto, Abrir, Guardar, Guardar Como y Salir.

7.1.2. Creación de Nuevo Proyecto

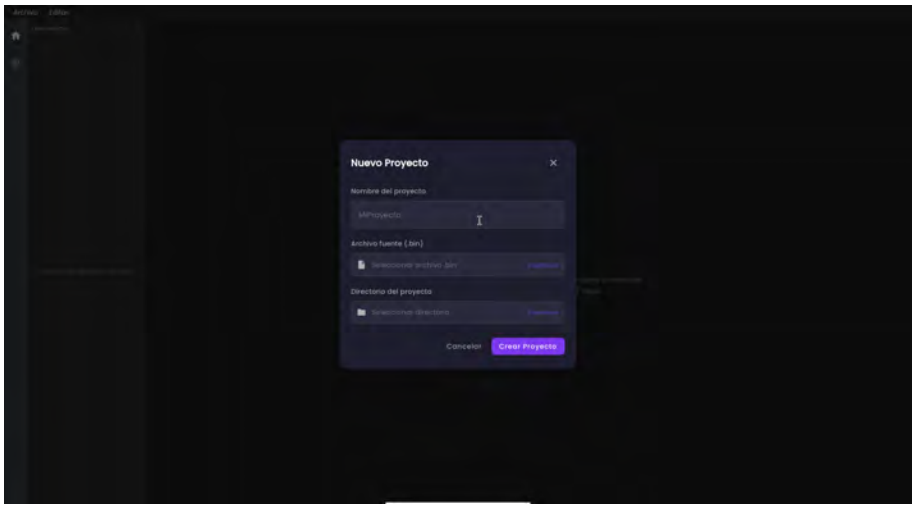

Figura 7.3: Modal de creación de nuevo proyecto. Permite especificar el nombre, seleccionar el archivo fuente (.bin) y el directorio de destino.

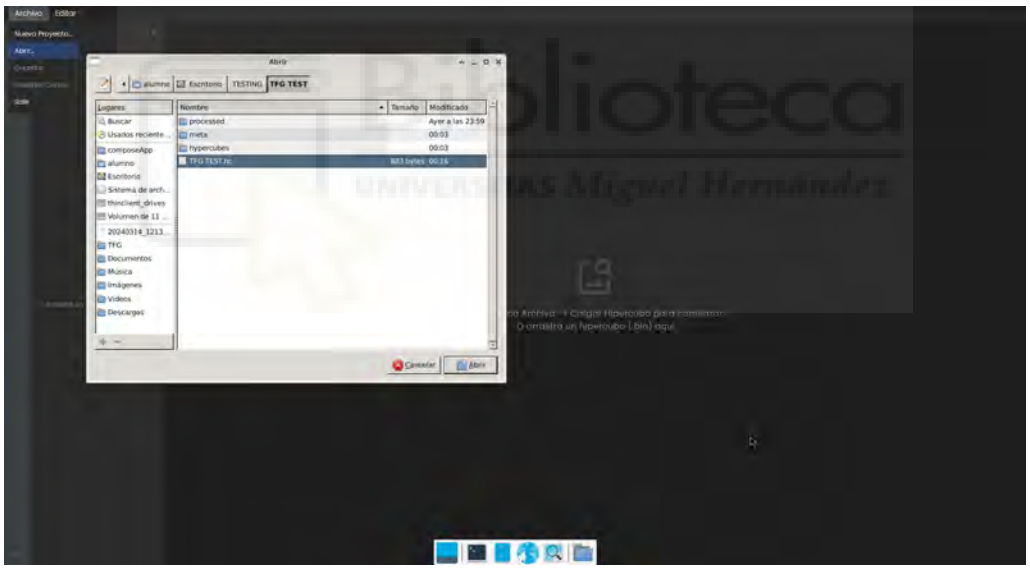

Figura 7.4: Diálogo para abrir un proyecto existente. Se muestra el explorador de archivos con la estructura de directorios del proyecto y la tabla con los archivos disponibles.

7.2. Interfaz Principal de Visualización

7.2.1. Vista General con un hipercubo con ROIs

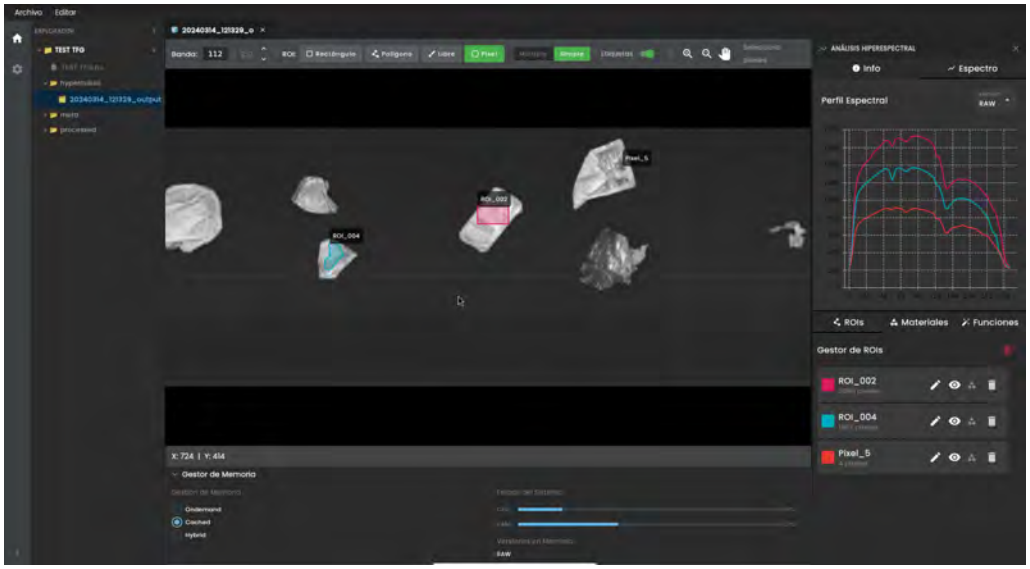

Figura 7.5: Se muestra el hipercubo "20240314_121329_o" cargado, con la banda 112 de 223 seleccionada y ROIs con *tags* creados

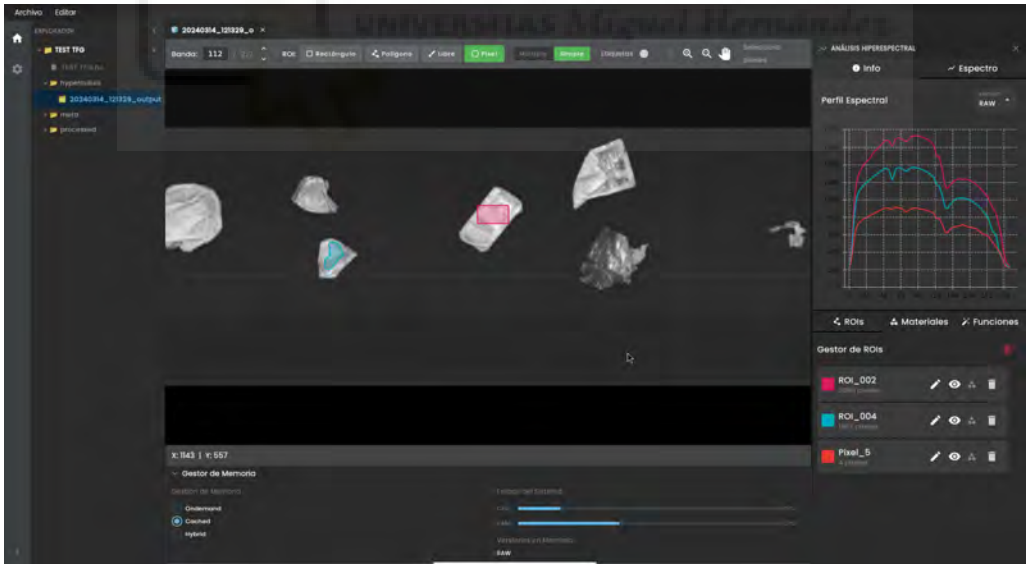

Figura 7.6: Se muestra el hipercubo "20240314_121329_o" cargado, con la banda 112 de 223 seleccionada y ROIs creados sin *tags* visibles

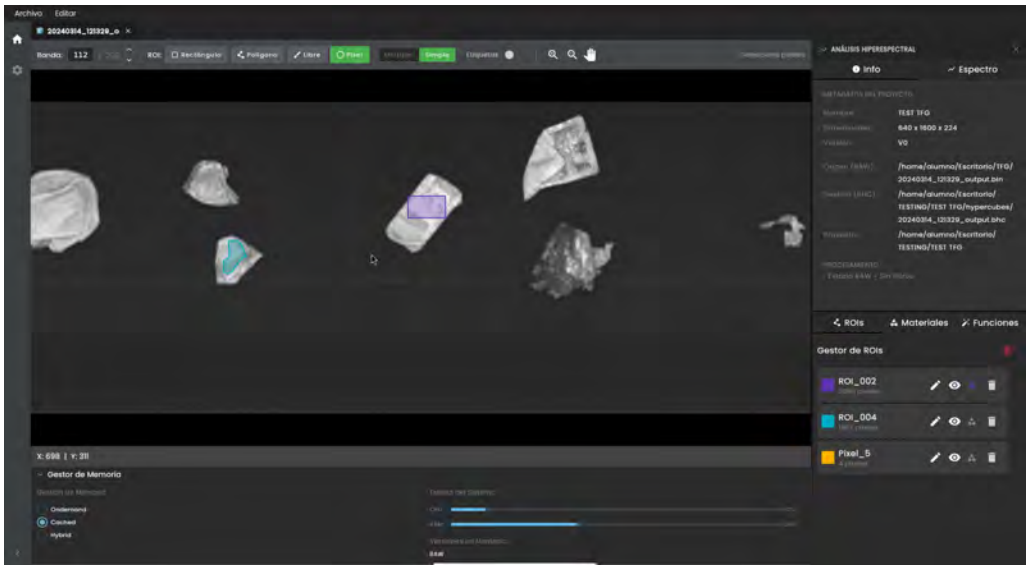

Figura 7.7: Vista de la interfaz con el panel de explorador de archivos contraido.

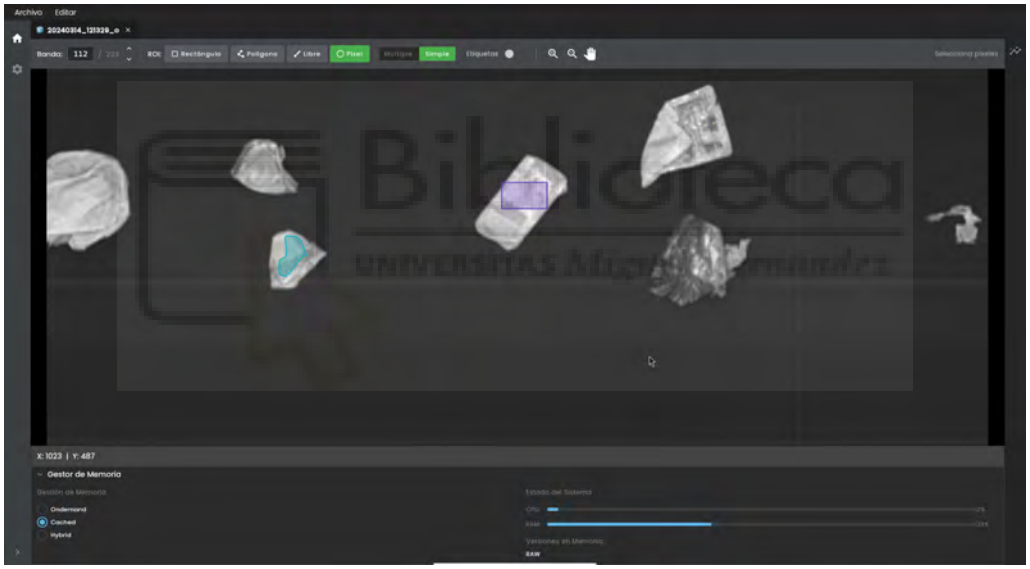

Figura 7.8: Vista de la interfaz con el panel de análisis contraido

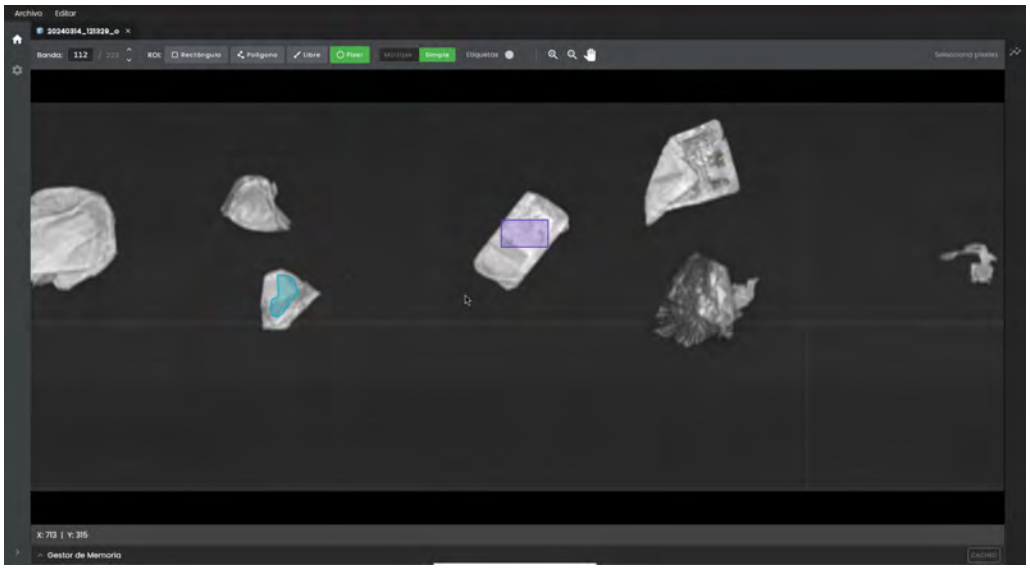

Figura 7.9: Panel del gestor de memoria colapsado, mostrando solo el título. Permite optimizar el espacio en la interfaz.

7.2.2. Exploración de la Imagen

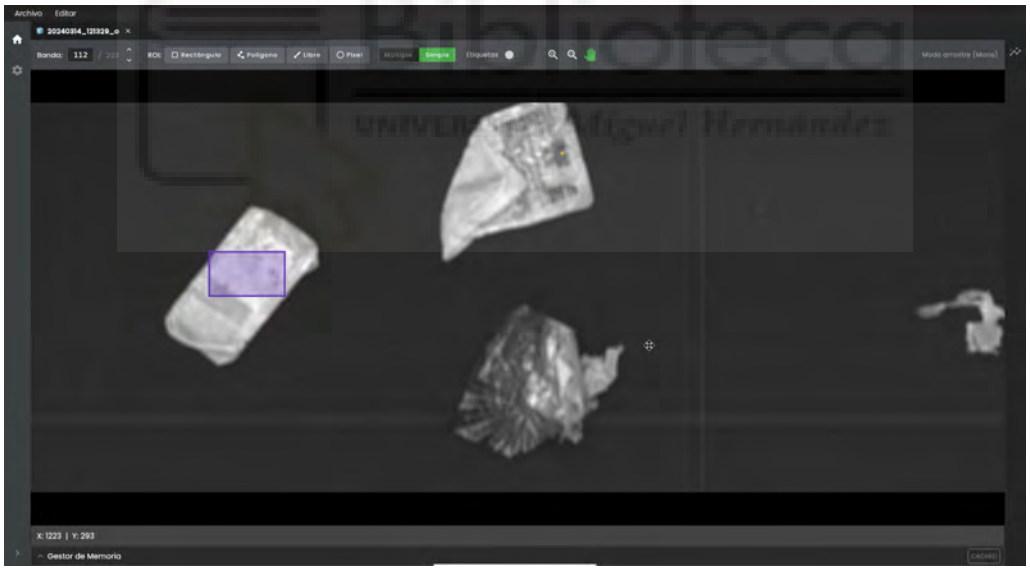

Figura 7.10: Vista ampliada (zoom) de la imagen hiperespectral. Se mantiene el modo arrastre para navegar por la zona ampliada. Coordenadas X:1223, Y:293.

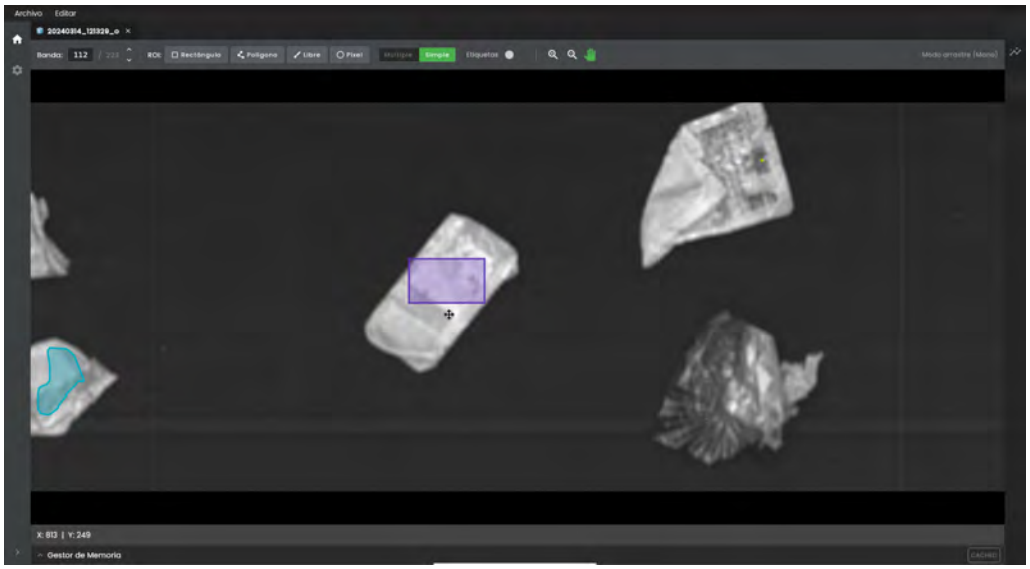

Figura 7.11: Modo de arrastre activado (icono de mano) para desplazar la imagen dentro del visor. Las coordenadas (X:813, Y:249) se actualizan en tiempo real. Se desplazó lateralmente la figura anterior 7.10

7.2.3. Selector despegable de versiones de hipercubos procesados

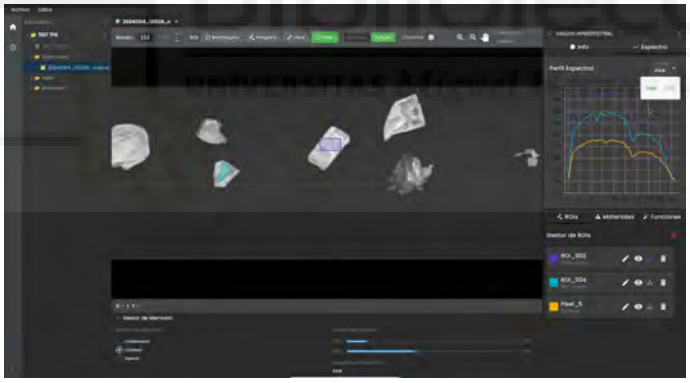

Figura 7.12: Gestor de memoria expandido con las opciones de gestión (Ondemand, Cached, Hybrid) y el estado del sistema mostrando CPU, RAM y las versiones en memoria (RAW).

7.3. Panel de información y metadatos del proyecto

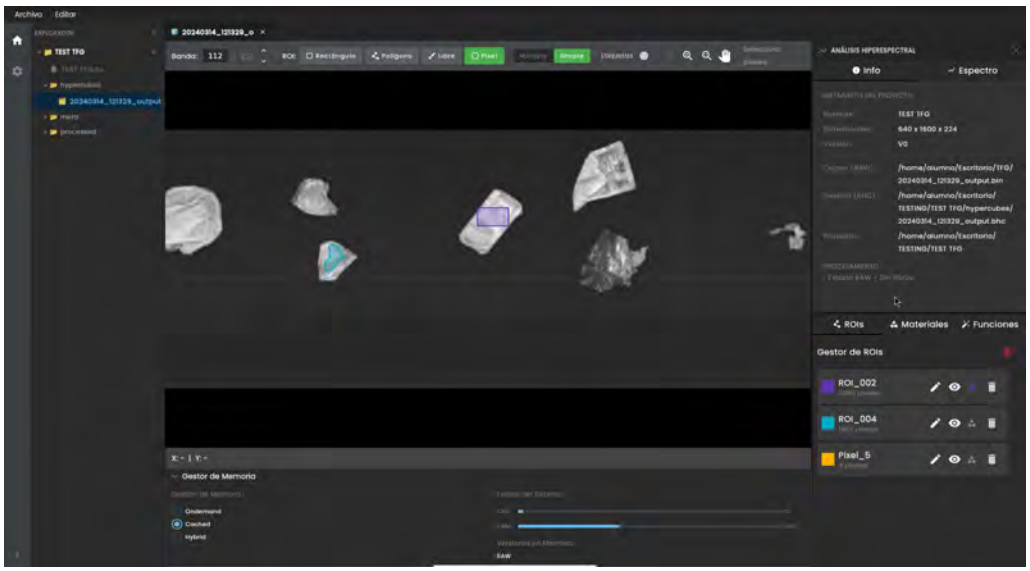

Figura 7.13: Pestaña de información mostrando los metadatos completos del proyecto: nombre (TEST TFG), dimensiones (640x1600x224), versión, rutas de origen y destino, y estado del procesamiento.

7.4. Selección y Gestión de ROIs

7.4.1. Herramientas de Selección

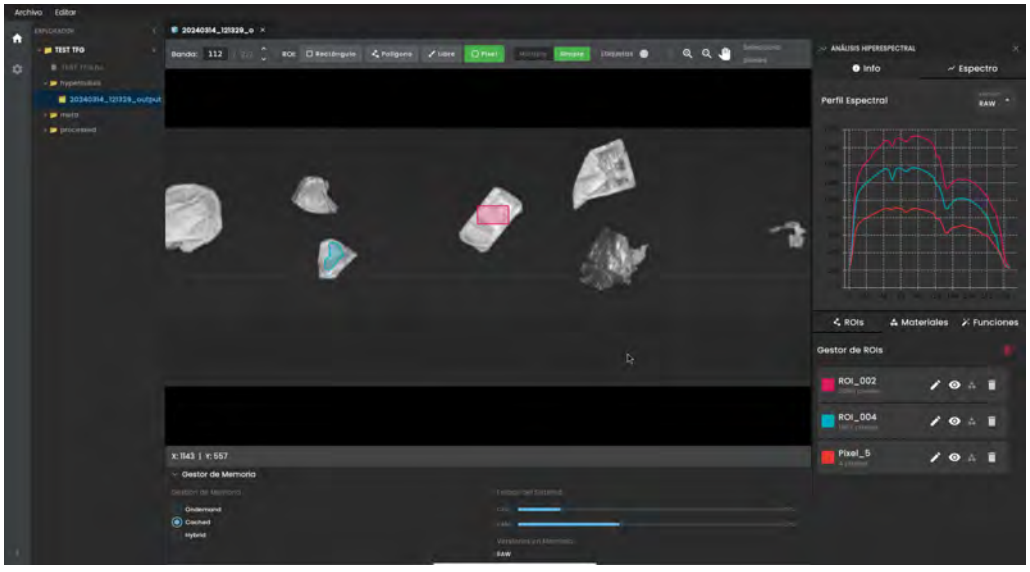

Figura 7.14: Panel de herramientas de selección de ROIs. Se muestran las opciones: Rectángulo, Polígono, Libre, Pixel, Múltiple, Simple y Etiquetas.

7.4.2. Gestor de ROIs

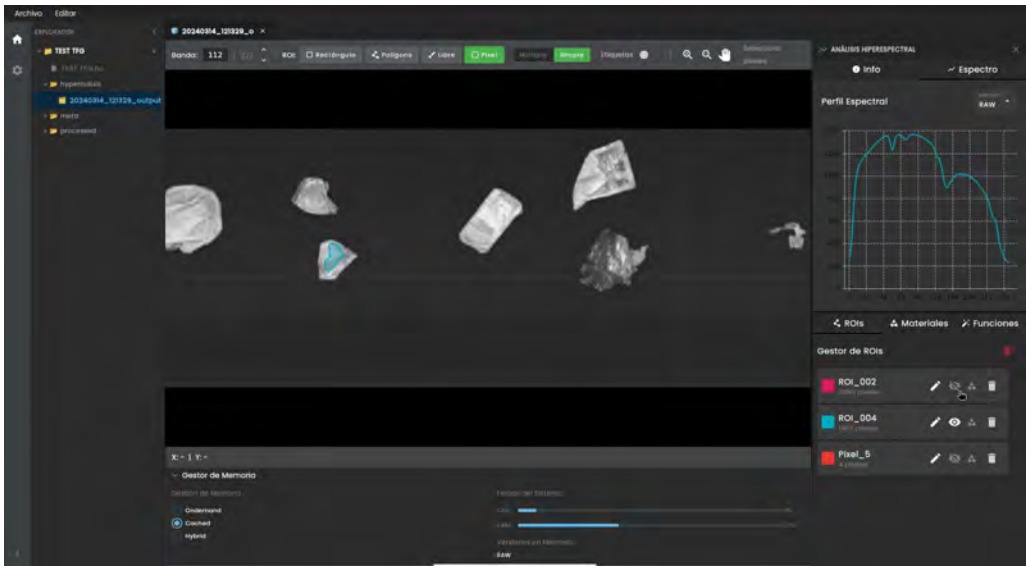

Figura 7.15: ROIs ocultas en la visualización. El gestor mantiene la lista pero las regiones no se dibujan sobre la imagen.

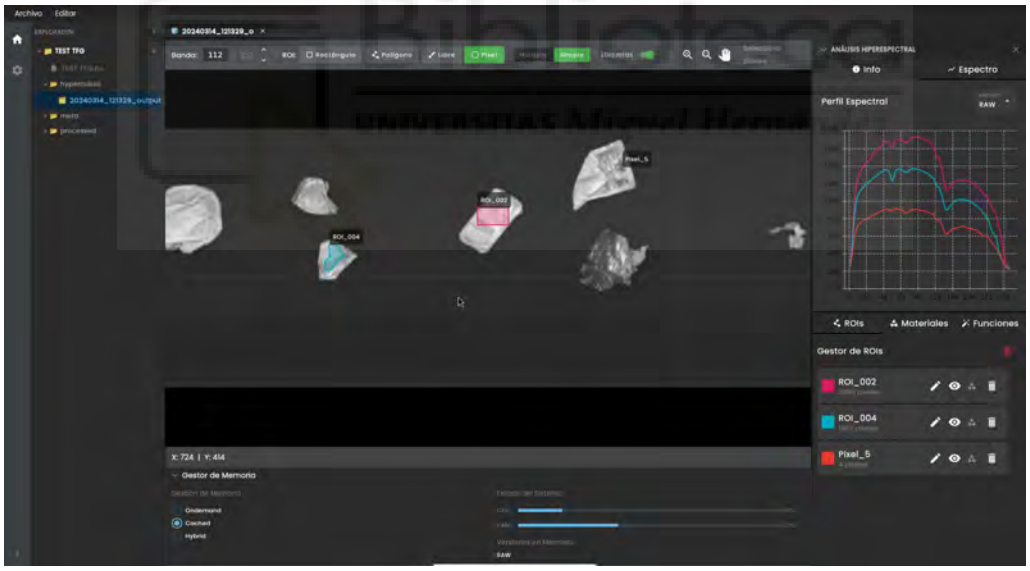

Figura 7.16: ROIs con materiales asignados. Se observa la diferenciación visual entre ROI_002 y ROI_004 tras la asignación.

7.4.3. Personalización de ROIs

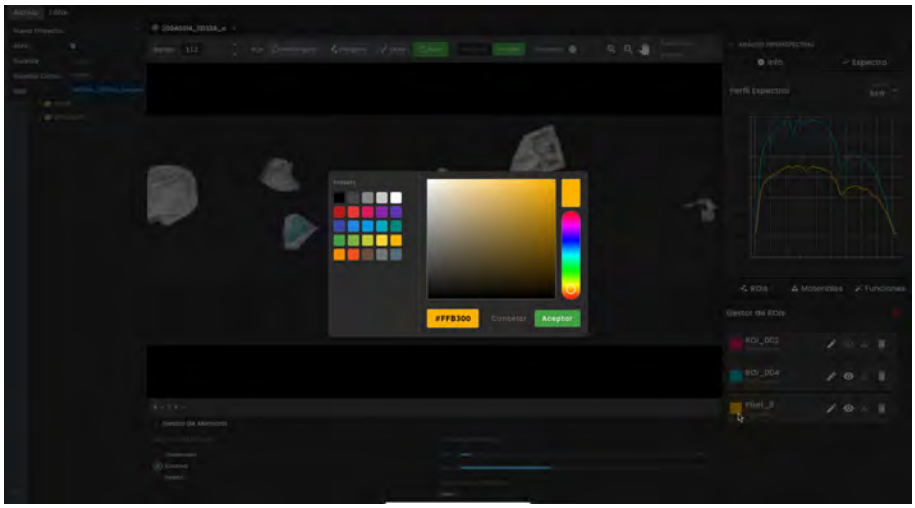

Figura 7.17: Selector de color para personalizar la visualización de una ROI. Se muestra el código hexadecimal #FFB300 seleccionado.

7.5. Gestión de Materiales y Espectros

7.5.1. Pestaña de Materiales

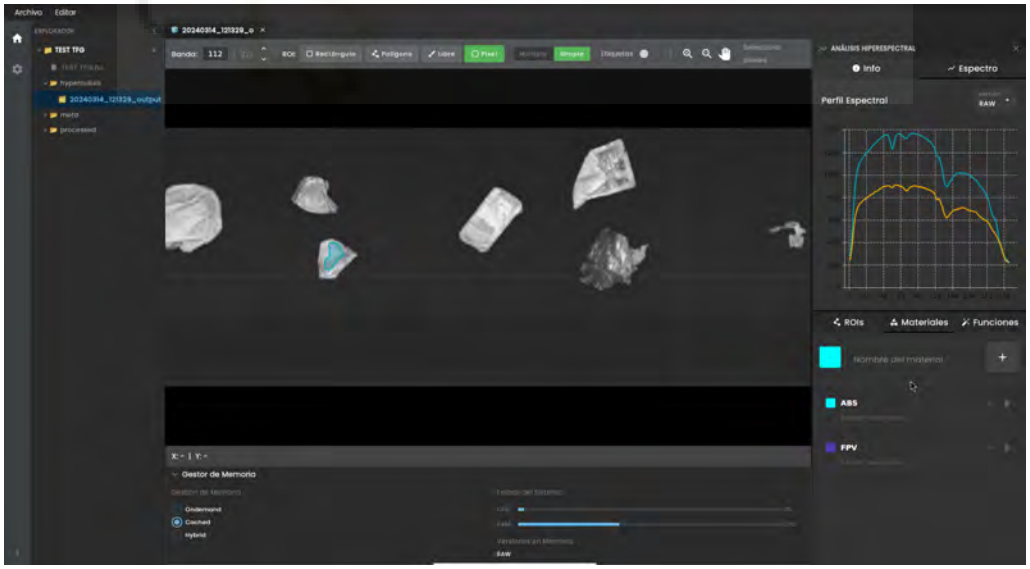

Figura 7.18: Pestaña de materiales mostrando los materiales definidos: ABS y FPV, cada uno con opción para añadir descripción.

7.5.2. Asignación de Materiales a ROIs

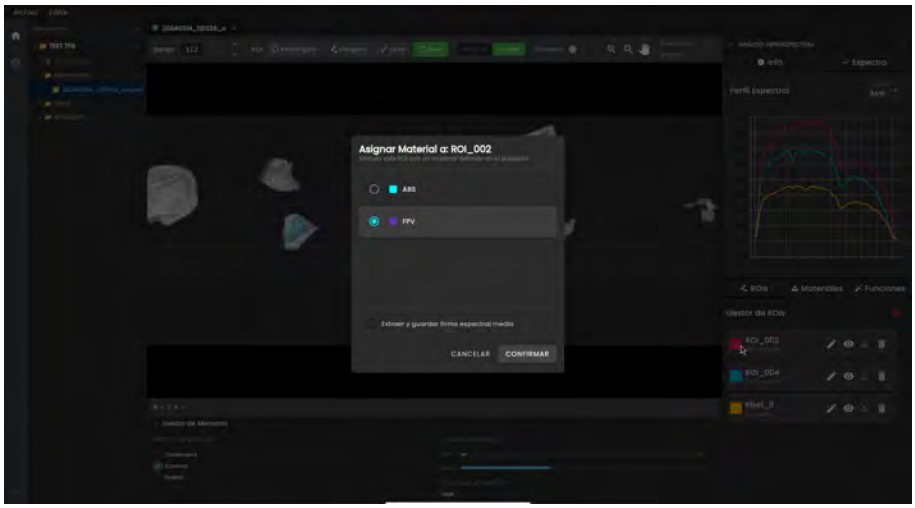

Figura 7.19: Modal de asignación de material a ROI_002. Permite vincular la región seleccionada con un material definido (ABS o FPV) y extraer la firma espectral media.

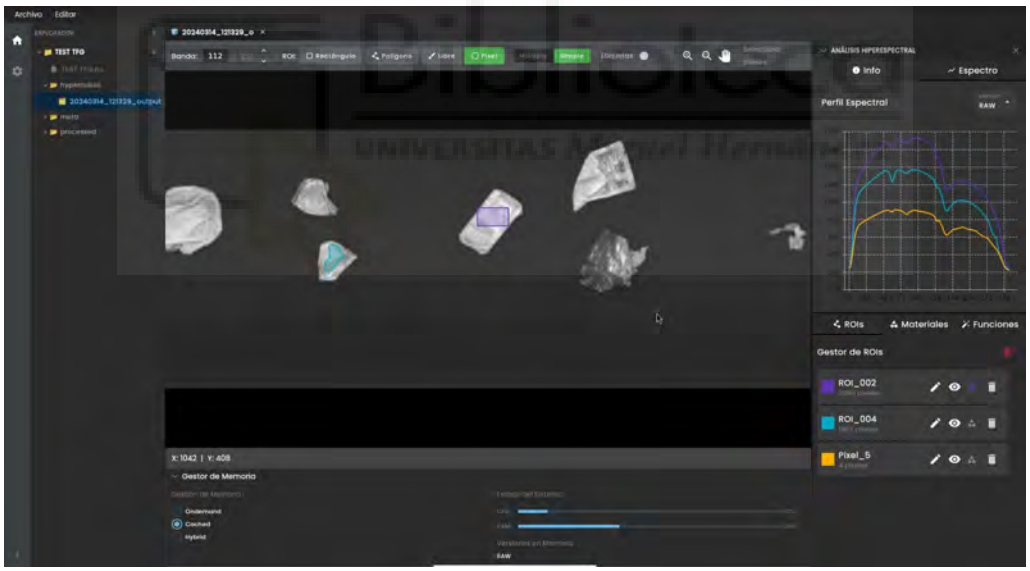

Figura 7.20: Gestor de ROIs mostrando las regiones creadas: ROI_002 (3588 píxeles), ROI_004 (1867 píxeles) y Pixel_5 (4 píxeles).

7.5.3. Panel de funciones

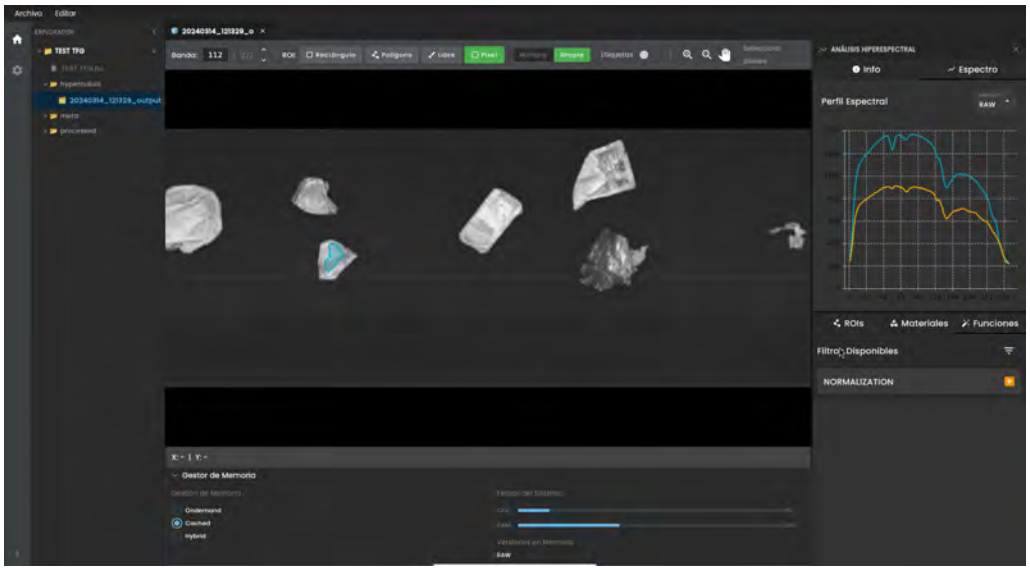

Figura 7.21: Pestaña de funciones mostrando el perfil espectral RAW y las opciones de análisis disponibles.

7.6. Modal de cierre

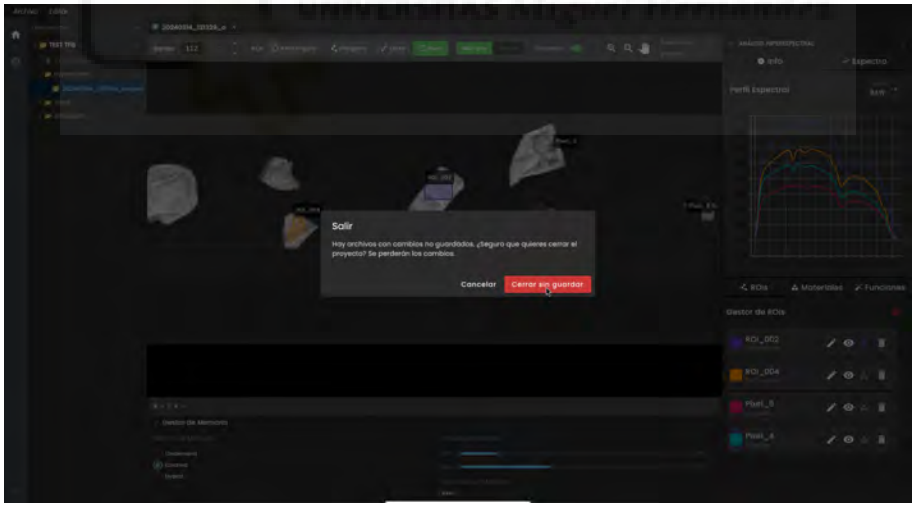

Figura 7.22: Modal de confirmación al salir de la aplicación. Advierte al usuario sobre cambios no guardados y ofrece las opciones de cancelar o cerrar sin guardar.

8. Conclusiones

Llegados a este capítulo del Trabajo de Fin de Grado, se puede afirmar que se han alcanzado satisfactoriamente los objetivos marcados al inicio del proyecto. Se ha logrado diseñar y construir una herramienta de software para el análisis de imágenes hiperespectrales que destaca por ser escalable, modular y altamente eficiente en la gestión de los recursos del sistema, usando la arquitectura propuesta.

El proceso de desarrollo ha resultado ser una experiencia sumamente valiosa. Más allá del hito curricular que supone la finalización de la etapa universitaria, el proyecto ha representado un importante crecimiento a nivel conceptual y profesional. Me ha permitido profundizar de forma práctica en el diseño de arquitecturas de software limpias y en el manejo de la memoria a bajo nivel, así como dominar el uso de nuevas herramientas, *frameworks* y librerías. Entre ellas, considero especialmente enriquecedoras para mi futuro profesional la adopción de *Compose Multiplatform* para el desarrollo de interfaces reactivas y *PlantUML* para la rigurosa documentación arquitectónica.

La sólida base de conocimientos de ingeniería del software adquirida durante mis años de estudio ha sido fundamental para abordar la complejidad de este sistema con solvencia. Gracias a ella, la asimilación de nuevos paradigmas y tecnologías no ha supuesto una barrera inalcanzable, sino más bien un reto apasionante y motivador al que he dedicado un gran esfuerzo y dedicación.

Asimismo, la integración de asistentes basados en inteligencia artificial durante el ciclo de desarrollo para optimizar la redacción, el diseño visual y la depuración del código, recalca una lección fundamental: la importancia de que el ingeniero moderno se adapte a los nuevos tiempos, aprendiendo a utilizar estas tecnologías emergentes para ser más eficiente y metódico en su trabajo.

En resumen, la herramienta desarrollada trasciende el propósito puramente académico y de evaluación de este Trabajo de Fin de Grado. Su enfoque modular y su rendimiento nativo demuestran que, con las adaptaciones y el desarrollo continuo pertinentes, este software tiene el potencial de ser utilizado en entornos industriales, agrícolas o de investigación científica real.

9. Líneas de Trabajo Futuro

El diseño modular y desacoplado de la aplicación, sustentado por la separación entre la interfaz gráfica y el motor nativo de procesamiento, sienta unas bases sólidas para la evolución del sistema. A continuación, se detallan las principales líneas de trabajo futuro que permitirían expandir las capacidades de la herramienta para su aplicación en entornos industriales o investigación.

9.1. Arquitectura Distribuida y Computación Remota

Tal y como se introdujo en las limitaciones del proyecto, el procesamiento de hipercubos masivos requiere de un hardware con gran capacidad de memoria RAM y múltiples núcleos de CPU. Para sortear esta restricción en equipos portátiles, móviles o de bajas prestaciones, el paso natural es migrar el motor C++ (`Hypercube_native`) hacia un servidor dedicado de alto rendimiento. En este escenario, la capa de aplicación en Kotlin actuaría como un cliente ligero (*Thin Client*).

Para implementar esta arquitectura cliente-servidor, se contemplan dos tecnologías principales: el uso de *Sockets* puros o el protocolo *gRPC*.

9.1.1. Enfoque basado en Sockets (TCP/WebSockets)

Los *sockets* proporcionan un canal de comunicación bidireccional de bajo nivel.

- **Ventajas:** Es una tecnología ubicua, con soporte nativo en prácticamente cualquier lenguaje, ideal para la transmisión continua de datos en bruto (*streaming* de la matriz de imagen).
- **Desventajas:** Carece de un formato de mensaje estandarizado. El desarrollador debe invertir tiempo en diseñar, serializar y parsear manualmente las peticiones (por ejemplo, definir cómo se envía un comando `applyFilter` y cómo se empaquetan los parámetros).

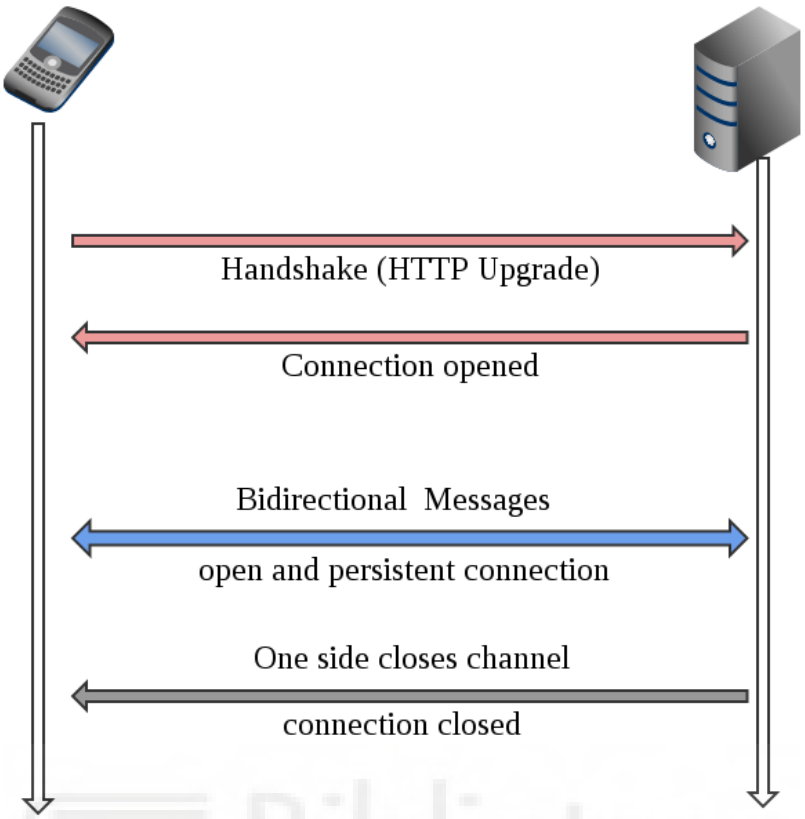

Figura 9.1: Esquema genérico de comunicación bidireccional mediante Sockets. Fuente: [36].

9.1.2. Enfoque basado en gRPC

gRPC es un *framework* moderno de Llamada a Procedimiento Remoto (RPC) de código abierto desarrollado por Google, que funciona sobre HTTP/2 y utiliza *Protocol Buffers* (Protobuf) [37].

- **Ventajas:** Permite definir fuertemente los servicios y estructuras de datos en un archivo `.proto`. A partir de este archivo, se genera automáticamente el código de red tanto para el cliente (Kotlin) como para el servidor (C++). Además, soporta *streaming* bidireccional nativo, ideal para enviar la renderización de la imagen por fragmentos.
- **Desventajas:** Curva de aprendizaje inicial más pronunciada y la necesidad de integrar el compilador de Protobuf en el flujo de construcción (*build system*).

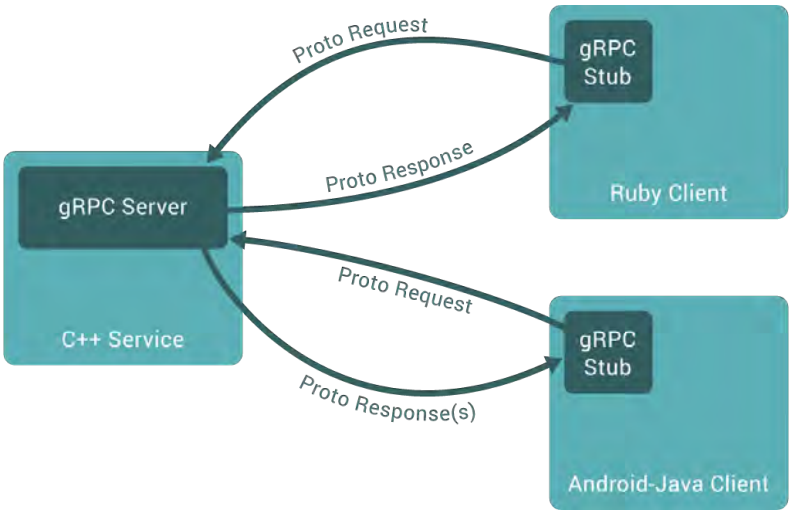

Figura 9.2: Arquitectura basada en gRPC comunicando clientes y servidores en distintos lenguajes. Fuente: [37]

9.1.3. Implementación Planteada

De cara a la escalabilidad del proyecto, **gRPC es la alternativa seleccionada**. La transición implicaría sustituir la capa `NativeBridge` actual (JNI) por un *gRPC Stub* en Kotlin,u otro enfoque sería permitir el modo de ejecución de la aplicación por el usuario. En el lado del servidor, el código en C++ actual se envolvería en un servicio gRPC capaz de recibir comandos, procesar los datos con OpenMP y devolver la banda espectral renderizada a través del canal HTTP/2, manteniendo la experiencia de usuario inalterada.

9.2. Implementación de un módulo avanzado de gestión de memoria

Actualmente, el sistema integra un panel de monitorización y diagnóstico que permite al usuario visualizar en tiempo real el consumo de recursos (uso de CPU y memoria RAM), así como auditar las distintas versiones de los hipercubos procesados que se encuentran alojados en la memoria nativa. Por defecto, el sistema emplea una estrategia de retención de memoria denominada *CACHED* (Cacheado). Bajo este paradigma, los datos en crudo y las versiones procesadas se mantienen persistentes en la memoria RAM hasta que el usuario cierra explícitamente el archivo o la aplicación. Esta configuración prioriza el rendimiento y la inmediatez, resultando ideal para flujos de trabajo intensivos que involucran múltiples preprocesamientos, filtrados o la creación y edición iterativa de Regiones de Interés (ROIs).

Para dotar al sistema de una mayor adaptabilidad y permitir su ejecución en equipos con recursos de hardware más limitados, se propone como trabajo futuro la expansión de este módulo mediante un selector de políticas de gestión de memoria que introduzca dos nuevos modos de operación:

- ***On-Demand* (Bajo demanda):** En este modo, el sistema operaría con una política estricta de asignación y liberación transitoria. Los datos masivos del hipercubo se cargarían en memoria exclusivamente durante la ejecución de una acción que los requiera y se liberarían inmediatamente al finalizar. Este enfoque es altamente eficiente para tareas de análisis puramente visual o comparativo, siendo viable gracias a la arquitectura actual del sistema, la cual ya genera y guarda *snapshots* (instantáneas ligeras) del espectro medio de los ROIs en cada versión, obviando la necesidad de mantener el volumen matricial tridimensional completo en RAM para renderizar las firmas espectrales.
- ***Hybrid* (Híbrido o basado en inactividad):** Esta política introduciría un mecanismo de gestión temporal similar a un *Time-To-Live* (TTL). Permitiría al usuario configurar un umbral de tiempo máximo de retención; de forma que, si un hipercubo o versión no es consultado durante dicho periodo de inactividad, el sistema emitiría la orden de descarga (*unload*) automáticamente. Este gestor inteligente actuaría de forma concurrente y segura (*thread-safe*), garantizando que la memoria jamás sea liberada si existe un hilo de ejecución o tarea de procesamiento espectral operando activamente sobre esos datos.

9.3. Módulo de Entrenamiento y Clasificación (Machine Learning)

Actualmente, el sistema permite asociar Regiones de Interés (ROIs) a Materiales, conformando el primer paso para el aprendizaje supervisado: el etiquetado (*Ground Truth*). El siguiente objetivo es desarrollar un módulo interno de clasificación espectral.

Este módulo extraería las firmas espectrales de los píxeles etiquetados y permitiría entrenar algoritmos clásicos de *Machine Learning* (como *Support Vector Machines* o *Random Forest*) directamente en la interfaz. Una vez entrenado, el modelo se aplicaría a la totalidad del hipercubo para clasificar y segmentar automáticamente cada píxel en la imagen, detectando contaminantes, plagas o materiales defectuosos sin intervención manual continua.

9.4. Herramienta de Composición y Creación de Hipercubos Sintéticos

Las cámaras hiperespectrales industriales (como los sensores *pushbroom*) capturan datos barriendo objetos a medida que estos se desplazan por una cinta transportadora. Para facilitar la creación de conjuntos de datos de prueba o la simulación de nuevos escenarios, se plantea la construcción de un módulo de Çomposición de Hipercubos".

Las características principales de este nuevo módulo serían:

1. **Extracción volumétrica:** Herramientas de recorte para seleccionar objetos en un hipercubo ya visualizado y extraer un "trozo.º sub-cubo tridimensional (conservando intactas todas las bandas espectrales de los píxeles seleccionados).

2. **Lienzo de escaneo:** Un editor visual donde el usuario pueda definir las dimensiones espaciales de un nuevo hipercubo virtual, asignando un espectro de fondo (el *background* o çinta transportadora").

3. **Posicionamiento interactivo:** Soporte para funcionalidad *Drag and Drop* (Arrastrar y Soltar) que permita importar los sub-cubos extraídos anteriormente e incrustarlos visualmente en el lienzo.

Esta herramienta generaría un archivo binario final que el sistema entendería como una captura real, resultando de enorme utilidad para proyectos de *Data Augmentation* aplicados a la inteligencia artificial en el sector agroalimentario o de reciclaje.

BIBLIOGRAFÍA

[1] A. Bhargava, A. Sachdeva, K. Sharma, M. H. Alsharif, P. Uthansakul y M. Uthansakul, «Hyperspectral Imaging and Its Applications: A Review,» *Heliyon*, vol. 10, 2024. DOI: 10.1016/j.heliyon.2024.e33208

[2] J. Kowalewski, J. Domaradzki, M. Zięba y M. Podgórski, «Hyperspectral imaging – a short review of methods and applications,» *Metrology and Measurement Systems*, vol. 30, n.º 4, págs. 785-804, 2023. DOI: 10.24425/mms.2023.147951

[3] Q. Zhang y M. B. Willmott, «Review of Hyperspectral Imaging in Environmental Monitoring: Progress and Applications,» *Academic Journal of Science and Technology*, vol. 6, n.º 2, págs. 9435-9454, 2025. DOI: 10.54097/ajst.v6i2.9435

[4] D. Wu y D.-W. Sun, «Advanced applications of hyperspectral imaging technology for food quality and safety analysis and assessment: A review — Part I: Fundamentals,» *Innovative Food Science and Emerging Technologies*, vol. 19, 2013. DOI: 10.1016/j.ifset.2013.04.014

[5] Q. Li, X. He, Y. Wang, H. Liu, D. Xu y F. Guo, «Review of spectral imaging technology in biomedical engineering: achievements and challenges,» *Journal of Biomedical Optics*, vol. 18, n.º 10, 2013. DOI: 10.1117/1.JBO.18.10.100901

[6] Library of Congress, *Band Interleaved by Line (BIL) Image File Format*, https://www.loc.gov/preservation/digital/formats/fdd/fdd000304.shtml, 2022.

[7] E. Gamma, R. Helm, R. Johnson y J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[8] JetBrains, *Kotlin Multiplatform Documentation*, https://kotlinlang.org/multiplatform/, 2023.

[9] W. H. Press, S. A. Teukolsky, W. T. Vetterling y B. P. Flannery, *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, 2007.

[10] OpenCV.org, *OpenCV Library*, https://opencv.org/, 2024.

[11] O. A. R. Board, *OpenMP Application Programming Interface*, https://www.openmp.org/, 2021.

[12] C.-I. Chang, *Hyperspectral Imaging: Techniques for Spectral Detection and Classification*. Kluwer Academic Publishers, 2003.

[13] Specim, Spectral Imaging Ltd., *Specim FX17 — Especificaciones de la cámara hiperespectral*, https://www.specim.com/products/specim-fx17/, 2026.

[14] Google, *Android Studio IDE*, https://developer.android.com/studio, 2024.

[15] Google, *Kotlin*, https://developer.android.com/kotlin/overview?hl=es-419, 2025.

[16] Oracle, *Java Native Interface Specification*, https://docs.oracle.com/javase/8/docs/technotes/guides/jni/, 2023.

[17] P. Goworowski y P. Michalik, *Vico: A light and extensible chart library for Android and Compose Multiplatform*, `https://github.com/patrykandpatrick/vico`, 2024.

[18] O. Team, *OSHI: Native Operating System and Hardware Information*, `https://github.com/oshi/oshi`, 2024.

[19] J. N. A. Team, *JNA: Java Native Access*, `https://github.com/java-native-access/jna`, 2024.

[20] G. Inc., *Gradle Build Tool User Manual*, https://docs.gradle.org/, 2024.

[21] K. Inc., *CMake Documentation*, https://cmake.org/cmake/help/latest/, 2024.

[22] S. Chacon y B. Straub, *Pro Git*. Apress, 2014.

[23] G. Inc., *GitHub Documentation*, https://docs.github.com/, 2024.

[24] PlantUML, *PlantUML: Open-source tool that uses simple textual descriptions to draw UML diagrams*, `https://plantuml.com/`, 2024.

[25] Google, *Gemini: A highly capable, multimodal AI model*, `https://deepmind.google/technologies/gemini/`, 2024.

[26] R. C. Martin, *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Prentice Hall, 2017, ISBN: 978-0-13-449416-6.

[27] G. Bradski y A. Kaehler, *Learning OpenCV*. O'Reilly Media, 2008.

[28] A. Cockburn, *Writing Effective Use Cases*. Addison-Wesley, 2001, ISBN: 978-0-201-70225-5.

[29] G. Developers, *Keep your app responsive (Main Thread)*, `https://developer.android.com/training/articles/perf-anr`, 2024.

[30] Google, *Skia Graphics Engine*, `https://skia.org/`, 2024.

[31] J. Bloch, *Effective Java*, 3rd. Addison-Wesley Professional, 2018, ISBN: 978-0134685991.

[32] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2008, ISBN: 978-0132350884.

[33] Oracle, *JNI Conversion Types*, https://docs.oracle.com/javase/8/docs/technotes/guides/jni/sp 2023.

[34] G. Bradski, «The OpenCV Library,» *Dr. Dobb's Journal of Software Tools*, 2000.

[35] L. Dagum y R. Menon, «OpenMP: An Industry-Standard API for Shared-Memory Programming,» *IEEE Computational Science and Engineering*, vol. 5, n.º 1, págs. 46-55, 1998. DOI: 10.1109/99.660313

[36] ResearchGate, *Web-socket architecture - Scientific Figure from: Financial stock application using websocket in real time application*, https://www.researchgate.net/figure/Web-socket-architecture_fig3_338553959, 2019.

[37] C. N. C. Foundation, *gRPC: A high performance, open source universal RPC framework*, https://grpc.io/, 2024.

ANEXO: MAPEO DE INTERFACES JNI

En el presente anexo se documenta el mapeo completo entre las funciones declaradas en Kotlin a través de JNI (Java Native Interface) y sus correspondientes implementaciones en C++. Esta tabla constituye una referencia esencial para comprender la interoperabilidad entre la capa de presentación desarrollada en Kotlin Multiplatform y el núcleo de procesamiento implementado en C++.

Tabla de correspondencia JNI

Tabla .1: Correspondencia entre funciones externas declaradas en Kotlin y sus implementaciones nativas en C++

#	Lenguaje	Firma / Descripción

1	Kotlin
	C++
		`external fun init()`
		`JNIEXPORT void JNICALL Java_es_umh_tfg_luisdavid_core_bridge_NativeBridge_init(JNIEnv*, jclass)`
		Inicializa la librería nativa y los gestores (HypercubeManager, RoiManager, MaterialManager, FilterRegistry)

2	Kotlin
	C++
		`external fun createProject(bhcPath: String): Int`
		`JNIEXPORT jint JNICALL Java_es_umh_tfg_luisdavid_core_bridge_NativeBridge_createProject(JNIEnv*, jclass, jstring)`
		Crea un nuevo proyecto cargando un hipercubo desde un archivo .bhc y retorna su ID

3	Kotlin
	C++
		`external fun loadBinary(path: String): Int`
		`JNIEXPORT jint JNICALL Java_es_umh_tfg_luisdavid_core_bridge_NativeBridge_loadBinary(JNIEnv*, jclass, jstring)`
		Carga un hipercubo desde un archivo binario (.bin, .bhc o .hdr) y retorna su ID

4	Kotlin
	C++
		`external fun releaseHypercube(hypercubeId: Int)`
		`JNIEXPORT void JNICALL Java_es_umh_tfg_luisdavid_core_bridge_NativeBridge_releaseHypercube(JNIEnv*, jclass, jint)`
		Libera los recursos asociados a un hipercubo específico

5	Kotlin
	C++
		`external fun release()`
		`JNIEXPORT void JNICALL Java_es_umh_tfg_luisdavid_core_bridge_NativeBridge_release(JNIEnv*, jclass)`
		Libera todos los recursos globales (hipercubos, ROIs, materiales y buffer de imagen)

Continúa en la siguiente página

#	Lenguaje	Firma / Descripción
6	Kotlin	`external fun getImageRows(hypercubeId: Int): Int`
	C++	`JNIEXPORT jint JNICALL Java_es_umh_tfg_luisdavid_core_bridge_NativeBridge_getImageRows(JNIEnv*, jclass, jint)` Retorna el número de filas (samples) del hipercubo
7	Kotlin	`external fun getImageCols(hypercubeId: Int): Int`
	C++	`JNIEXPORT jint JNICALL Java_es_umh_tfg_luisdavid_core_bridge_NativeBridge_getImageCols(JNIEnv*, jclass, jint)` Retorna el número de columnas (lines) del hipercubo
8	Kotlin	`external fun getImageBands(hypercubeId: Int): Int`
	C++	`JNIEXPORT jint JNICALL Java_es_umh_tfg_luisdavid_core_bridge_NativeBridge_getImageBands(JNIEnv*, jclass, jint)` Retorna el número de bandas espectrales del hipercubo
9	Kotlin	`external fun getPixelSpectrum(hypercubeId: Int, y: Int, x: Int): ShortArray?`
	C++	`JNIEXPORT jshortArray JNICALL Java_es_umh_tfg_luisdavid_core_bridge_NativeBridge_getPixelSpectrum(JNIEnv*, jclass, jint, jint, jint)` Obtiene el espectro completo de un píxel en coordenadas (y, x) como un array de shorts
10	Kotlin	`external fun getBandImage2DBufferPtr(hypercubeId: Int, bandIndex: Int): ByteBuffer?`
	C++	`JNIEXPORT jobject JNICALL Java_es_umh_tfg_luisdavid_core_bridge_NativeBridge_getBandImage2DBufferPtr(JNIEnv*, jclass, jint, jint)` Obtiene un buffer directo con los datos de una banda específica para renderizado
11	Kotlin	`external fun getVersionCount(hypercubeId: Int): Int`
	C++	`JNIEXPORT jint JNICALL Java_es_umh_tfg_luisdavid_core_bridge_NativeBridge_getVersionCount(JNIEnv*, jclass, jint)` Retorna el número de versiones procesadas del hipercubo
12	Kotlin	`external fun getVersionCvTypeString(hypercubeId: Int, versionIndex: Int): String`
	C++	`JNIEXPORT jstring JNICALL Java_es_umh_tfg_luisdavid_core_bridge_NativeBridge_getVersionCvTypeString(JNIEnv*, jclass, jint, jint)`

Continúa en la siguiente página

#	Lenguaje	Firma / Descripción
		Obtiene el tipo OpenCV (ej. ÇV_16UC1", ÇV_32FC1") de una versión específica

13	Kotlin
	C++
		`external fun getCvType(hypercubeId: Int): String`
		`JNIEXPORT jstring JNICALL Java_es_umh_tfg_luisdavid_core_bridge_NativeBridge_getCvType(JNIEnv*, jclass, jint)`
		Obtiene el tipo OpenCV de la versión raw (original) del hipercubo

14	Kotlin
	C++
		`external fun createRoi(hypercubeId: Int, contourPoints: IntArray): Int`
		`JNIEXPORT jint JNICALL Java_es_umh_tfg_luisdavid_core_bridge_NativeBridge_createRoi(JNIEnv*, jclass, jint, jintArray)`
		Crea un ROI a partir de un array de puntos que definen su contorno

15	Kotlin
	C++
		`external fun createPointRoi(hypercubeId: Int, x: Int, y: Int): Int`
		`JNIEXPORT jint JNICALL Java_es_umh_tfg_luisdavid_core_bridge_NativeBridge_createPointRoi(JNIEnv*, jclass, jint, jint, jint)`
		Crea un ROI de tipo punto en las coordenadas (x, y)

16	Kotlin
	C++
		`external fun getRoiPixelCount(hypercubeId: Int, roiId: Int): Int`
		`JNIEXPORT jint JNICALL Java_es_umh_tfg_luisdavid_core_bridge_NativeBridge_getRoiPixelCount(JNIEnv*, jclass, jint, jint)`
		Retorna el número de píxeles que contiene un ROI

17	Kotlin
	C++
		`external fun getRoiMeanSpectrum(hypercubeId: Int, versionIndex: Int, roiId: Int): DoubleArray?`
		`JNIEXPORT jdoubleArray JNICALL Java_es_umh_tfg_luisdavid_core_bridge_NativeBridge_getRoiMeanSpectrum(JNIEnv*, jclass, jint, jint, jint)`
		Calcula el espectro medio de los píxeles de un ROI para una versión específica

18	Kotlin
	C++
		`external fun deleteRoi(hypercubeId: Int, roiId: Int): Boolean`
		`JNIEXPORT jboolean JNICALL Java_es_umh_tfg_luisdavid_core_bridge_NativeBridge_deleteRoi(JNIEnv*, jclass, jint, jint)`
		Elimina un ROI del hipercubo y del gestor de ROIs

19	Kotlin	`external fun createMaterial(hypercubeId: Int, r: Int, g: Int, b: Int, a: Int): Int`

Continúa en la siguiente página

#	Lenguaje	Firma / Descripción
	C++	`JNIEXPORT jint JNICALL Java_es_umh_tfg_luisdavid_core_bridge_NativeBridge_createMaterial(JNIEnv*, jclass, jint, jint, jint, jint, jint)` *Discrepancia:* En C++ no se reciben name y description, solo color RGBA
20	Kotlin	`external fun updateMaterial(hypercubeId: Int, materialId: Int, r: Int, g: Int, b: Int, a: Int): Boolean`
	C++	`JNIEXPORT jboolean JNICALL Java_es_umh_tfg_luisdavid_core_bridge_NativeBridge_updateMaterial(JNIEnv*, jclass, jint, jint, jint, jint, jint, jint)` *Discrepancia:* En C++ solo se actualiza el color, no nombre/
21	Kotlin	`external fun restoreMaterialWithId(hypercubeId: Int, materialId: Int, r: Int, g: Int, b: Int, a: Int): Boolean`
	C++	`JNIEXPORT jboolean JNICALL Java_es_umh_tfg_luisdavid_core_bridge_NativeBridge_restoreMaterialWithId(JNIEnv*, jclass, jint, jint, jint, jint, jint, jint)` Restaura un material con ID específico al cargar un proyecto (solo color en C++)
22	Kotlin	`external fun assignMaterialToRoi(hypercubeId: Int, roiId: Int, materialId: Int): Boolean`
	C++	`JNIEXPORT jboolean JNICALL Java_es_umh_tfg_luisdavid_core_bridge_NativeBridge_assignMaterialToRoi(JNIEnv*, jclass, jint, jint, jint)` Asigna un material a un ROI específico
23	Kotlin	`external fun getRoiMaterialId(hypercubeId: Int, roiId: Int): Int`
	C++	`JNIEXPORT jint JNICALL Java_es_umh_tfg_luisdavid_core_bridge_NativeBridge_getRoiMaterialId(JNIEnv*, jclass, jint, jint)` Obtiene el ID del material asignado a un ROI
24	Kotlin	`external fun unassignMaterialFromRoi(hypercubeId: Int, roiId: Int): Boolean`
	C++	`JNIEXPORT jboolean JNICALL Java_es_umh_tfg_luisdavid_core_bridge_NativeBridge_unassignMaterialFromRoi(JNIEnv*, jclass, jint, jint)` Desvincula el material de un ROI
25	Kotlin	`external fun deleteMaterial(hypercubeId: Int, materialId: Int): Boolean`

Continúa en la siguiente página

#	Lenguaje	Firma / Descripción
	C++	`JNIEXPORT jboolean JNICALL Java_es_umh_tfg_ luisdavid_core_bridge_NativeBridge_ deleteMaterial(JNIEnv*, jclass, jint, jint)` Elimina un material del gestor y del hipercubo
26	Kotlin	`external fun getTransformCount(hypercubeId: Int): Int`
	C++	`JNIEXPORT jint JNICALL Java_es_umh_tfg_ luisdavid_core_bridge_NativeBridge_ getTransformCount(JNIEnv*, jclass, jint)` Retorna el número de transformaciones aplicadas a la versión activa
27	Kotlin	`external fun getTransformId(hypercubeId: Int, transformIndex: Int): String?`
	C++	`JNIEXPORT jstring JNICALL Java_es_umh_tfg_ luisdavid_core_bridge_NativeBridge_ getTransformId(JNIEnv*, jclass, jint, jint)` Obtiene el ID del filtro de una transformación específica
28	Kotlin	`external fun getTransformParamKeys(hypercubeId: Int, transformIndex: Int): Array<String>?`
	C++	`JNIEXPORT jobjectArray JNICALL Java_es_umh_tfg_ luisdavid_core_bridge_NativeBridge_ getTransformParamKeys(JNIEnv*, jclass, jint, jint)` Obtiene los nombres de los parámetros de una transformación
29	Kotlin	`external fun getTransformParamValues(hypercubeId: Int, transformIndex: Int): Array<String>?`
	C++	`JNIEXPORT jobjectArray JNICALL Java_es_umh_tfg_ luisdavid_core_bridge_NativeBridge_ getTransformParamValues(JNIEnv*, jclass, jint, jint)` Obtiene los valores de los parámetros de una transformación
30	Kotlin	`external fun getAvailableFilters(): Array<FilterInfo>`
	C++	`JNIEXPORT jobjectArray JNICALL Java_es_umh_tfg_ luisdavid_core_bridge_NativeBridge_ getAvailableFilters(JNIEnv*, jclass)` Obtiene todos los filtros disponibles registrados en FilterRegistry
31	Kotlin	`external fun getFiltersByCategory(category: String): Array<FilterInfo>`
	C++	`JNIEXPORT jobjectArray JNICALL Java_es_umh_tfg_ luisdavid_core_bridge_NativeBridge_ getFiltersByCategory(JNIEnv*, jclass, jstring)` Obtiene los filtros de una categoría específica

Continúa en la siguiente página

#	Lenguaje	Firma / Descripción
32	Kotlin	`external fun getFilterMetadata(filterId: String): FilterMetadata?`
	C++	`JNIEXPORT jobject JNICALL Java_es_umh_tfg_luisdavid_core_bridge_NativeBridge_getFilterMetadata(JNIEnv*, jclass, jstring)` Obtiene los metadatos completos de un filtro (parámetros, rangos, etc.)
33	Kotlin	`external fun applyFilter(hypercubeId: Int, versionIndex: Int, filterId: String, paramKeys: Array<String>, paramValues: Array<String>, createNewVersion: Boolean): Boolean`
	C++	`JNIEXPORT jboolean JNICALL Java_es_umh_tfg_luisdavid_core_bridge_NativeBridge_applyFilter(JNIEnv*, jclass, jint, jint, jstring, jobjectArray, jobjectArray, jboolean)` Aplica un filtro sobre una versión del hipercubo, creando nueva versión o sobrescribiendo