UNIVERSIDAD MIGUEL HERNÁNDEZ DE ELCHE

ESCUELA POLITÉCNICA SUPERIOR DE ELCHE

GRADO EN INGENIERÍA DE TECNOLOGÍAS DE TELECOMUNICACIÓN

"Aplicación Android para comprimir imágenes con el codificador PETW"

TRABAJO FIN DE GRADO

Febrero-2018

AUTOR: Andrés Ruiz García

DIRECTORES: Miguel Onofre Martínez Rach

Otoniel Mario López Granado

Objetivos y motivación del proyecto

Este proyecto consistirá en la realización de una aplicación sobre el sistema operativo Android que tendrá como objetivos realizar fotografías y comprimirlas. Para la correcta toma de estas, la app realizará los ajustes de enfoque y zoom de forma completamente automática, de forma que una vez realizadas, se comprimirán las imágenes perceptualmente, haciendo uso del algoritmo de compresión PETW.

Todos los días tomamos fotografías desde nuestro móvil para subirlas a redes sociales o simplemente almacenarlas, pero muchas veces nos encontramos con que nos falta memoria y debido a este problema, no podemos seguir guardando imágenes. Además, los algoritmos de compresión son de gran utilidad, puesto que nos permiten guardar en la memoria de nuestro teléfono la mayor cantidad de ficheros posibles. Pues bien, esta aplicación no solo permitirá al usuario tomar imágenes, sino que además le permitirá elegir la calidad de la compresión, guardando las imágenes en formato PETW, maximizando, de esta forma, la capacidad de memoria de almacenamiento.

Se ha elegido Android, puesto que lidera el mercado como sistema operativo en la telefonía móvil. La gestión de la cámara será implementada en java, haciendo uso de la librería *"camera2raw"*.Sin embargo, toda la parte correspondiente al compresor de imágenes será implementado directamente en código nativo C++. Se ha elegido esta opción y no la implementación directa en java, para maximizar el rendimiento del compresor y además para reutilizar gran parte del código C++ en el que está escrito el algoritmo de compresión.

A lo largo de este proyecto se explicará detalladamente el desarrollo de la aplicación, desde la interfaz gráfica, la realización de la cámara con la toma de imágenes en formato RAW con autoenfoque, hasta la implementación de código nativo en Android, con la implantación del compresor para la mayoría de las arquitecturas de teléfono que se usan en el mercado. Finalmente se mostrará el funcionamiento de la aplicación.

Palabras clave: Android, compresión, imagen, PETW

Agradecimientos

Quisiera agradecer en primer lugar, a mis directores de TFG por todo el apoyo y los conocimientos aportados, de los que he aprendido muchísimo, más concretamente a:

A Don Mario Otoniel López Granado, por todas las horas aportadas a este proyecto, por toda la experiencia, así como todo su apoyo.

A Don Miguel Onofre Martínez Rach, por todo su apoyo, su experiencia, todos los mensajes de ánimo cuando se presentaron dificultades y todas las horas de dedicadas.

A los dos, de verdad y de corazón, ¡Muchísimas gracias por todo!

También quiero agradecer a mis padres, todo su esfuerzo, el levantarse cada mañana, todo el apoyo, todos los ánimos y toda la fuerza que me han transmitido siempre para salir adelante, ¡Muchísimas gracias por todo, esta victoria es nuestra!

A Laura Parra Navarro por estar siempre ahí, por ser mi guía, mi luz, mi ánimo en todo momento, ¡Muchísimas gracias por todo, amor de mi vida, por existir y por estar siempre ahí!

A Pedro José Martínez Sánchez, amigos como tu ya quedan muy pocos, ¡Muchísimas gracias por haber esta siempre ahí, darme tu apoyo y enseñarme que siempre hay que seguir adelante!

A José María López Bonaque, ¡Muchísimas gracias por escucharme siempre, por estar siempre ahí, por todo tu apoyo!

A Agustín Navarro González, por estar siempre ahí, por todo tu apoyo, tu ánimo y por hacer de el último año de estancia en la carrera, algo increíble en lo que se refiere a la convivencia, ¡Gracias por todo!

A Fina Navarro, por todos sus consejos, por escucharme siempre, por enseñarme que pese a cualquier dificultad se lucha para salir adelante, y por transmitirme ánimo en todo momento en el que lo he perdido, ¡Muchísimas gracias por todo!

Y, en definitiva, a todos aquellos que habéis hecho posible la realización de este proyecto, os estaré eternamente agradecido.

Índice

Índice de figuras

1. Introducción

1.1. Fundamentos de la codificación de imágenes

Hoy en día los algoritmos de codificación de imagen son cada vez más utilizados y las cámaras fotográficas como las de los dispositivos móviles cada vez son de mayor calidad, generando imágenes con cada vez mayor resolución, lo que se traduce en menos espacio a la hora de almacenar dichas imágenes tanto en la memoria de una cámara como en nuestro teléfono. Es por estos motivos por lo que el uso de un algoritmo de codificación es fundamental, permitiéndonos aprovechar el espacio de memoria, de una manera más eficiente y, así poder guardar más imágenes en la misma capacidad de memoria. Esta es la ventaja que nos brindan las imágenes digitales, podemos eliminar información redundante, que no tiene por qué tener un impacto significativo de cara a la calidad de imagen percibida por el usuario. Además, en este proyecto, en la solución que se propone, será el usuario, el encargado de elegir la cantidad de información que está dispuesto a perder conforme a sus intereses.

Por otra parte, el sistema operativo Android con sus diferentes versiones de API, domina actualmente más del 86 % de la cuota del mercado mundial en telefonía móvil, por lo que cada vez está más presente la necesidad de generar soluciones de compresión de imagen que se adapten a este sistema operativo de forma que cualquier persona pueda en cualquier momento comprimir estas imágenes desde el mismo dispositivo móvil sin la necesidad de tener que operar en un ordenador portátil o de sobremesa.

Esta sección de introducción tiene como objetivo dar a conocer los conceptos que tienen que ver con la codificación de imágenes, cuales son los diferentes tipos de codificación, así como conocer los principales formatos de compresión de existen, teniendo en cuenta sus diferencias, ventajas y desventajas.

1.2. Algoritmos de compresión de imágenes

Los algoritmos de compresión de imágenes se pueden clasificar en dos grandes grupos generalmente:

- Algoritmos de compresión con pérdidas de información
- Algoritmos de compresión sin perdidas de información

En el primer tipo de algoritmos, una vez procesada la imagen, al reconstruirla se genera otra, la cual es diferente en mayor o menor medida a la original, mientras que, en el segundo tipo, la imagen reconstruida es idéntica a la original.

1.2.1. Algoritmos de compresión con pérdidas de información

En este tipo de algoritmos, las imágenes reconstruidas no son iguales a la imagen original. Se utilizan fundamentalmente cuando dentro de estas imágenes existe información redundante que puede ser eliminada.

La eliminación de esta información redundante se lleva a cabo mediante técnicas de codificación basadas en la fuente. Este tipo de técnicas se basan en las propiedades de la imagen, llegando generalmente a altas tasas de compresión.

Una de las técnicas más utilizadas para esta eliminación de información redundante, usada además en el algoritmo PETW es:

- Codificación por transformación

La codificación por transformación consiste en emplear transformadas matemáticas frecuenciales para realizar una correspondencia de la imagen con conjunto de coeficientes de dicha transformada. Sobre dichos coeficientes se aplica un proceso de cuantificación, donde generalmente un número considerable de los coeficientes que se corresponden con valores poco significativos, se puede suprimir, produciéndose una pérdida de información.

1.2.2. Algoritmos de compresión sin perdidas de información

Los algoritmos de compresión sin perdidas, son en realidad, algoritmos de codificación que tienen por objetivo realizar una representación de la información, con el fin de ocupar menos espacio en memoria de almacenamiento, pudiendo realizar una reconstrucción exacta de la imagen o los datos originales. Este tipo de algoritmos, se basan principalmente en dos conceptos:

- La codificación Huffman
- La entropía

Estos conceptos se suelen implementar generalmente mediante dos modelos fundamentales:

- Modelo estático
- Modelo basado en diccionario

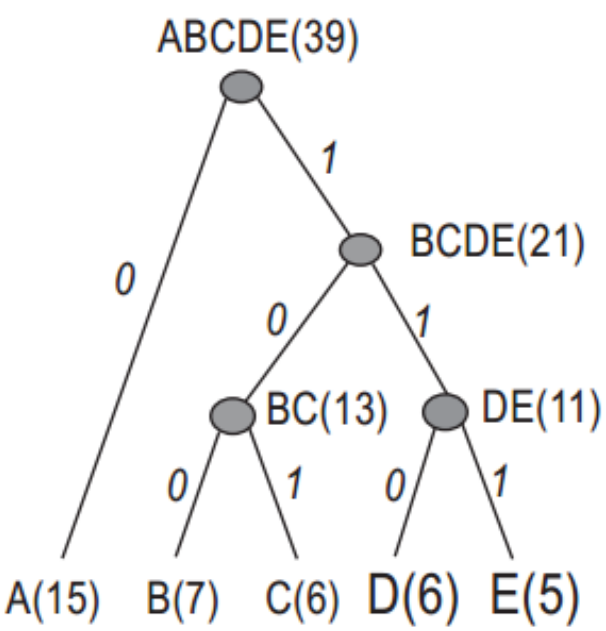
Por un lado, el modelo estático tiene como objetivo realizar una codificación de los datos en función de su probabilidad, tomando como base una tabla de probabilidades, la cual es estática. Sin embargo, construir un árbol Huffman a partir de los datos, tiene un coste computaciones elevado, por lo que generalmente se analizan solo ciertos bloques de datos, lo más representativos y con estos, se confecciona una tabla de frecuencias. A partir de dicha tabla, se genera el árbol de Huffman. No obstante, hacer uso de un modelo estático, tiene sus desventajas y es que si el flujo de datos de entrada, no se corresponde con la tabla de frecuencias, entonces tendremos como consecuencia una degradación de la relación de compresión.

Por otro lado, el modelo basado en diccionario utiliza un código simple que sustituye cadenas de símbolos. Este modelo lee los datos a la entrada y trabaja con grupos de símbolos pertenecientes a un diccionario. Si una cadena se corresponde, el código del símbolo es sustituido por un indicador.

La entropía es una técnica que realiza la codificación de los datos de la imagen sin tener precedentes de estos. Para los algoritmos de este tipo, la imagen que se reconstruye es exactamente igual que la imagen primigenia, es por esto que no hay perdidas de información. A continuación, se describirán algunas de las técnicas utilizadas, más importantes en este tipo de algoritmos:

- Codificación Huffman
- Codificación aritmética

Por un lado, la codificación Huffman [7] tiene como objetivo a los datos que aparecen con una mayor frecuencia, asignarles códigos de bits más cortos, mientras que a los que aparecen con menos frecuencia, códigos más largos. En realidad, no es más que un árbol de búsqueda binario que va desde abajo hasta arriba. En la figura 1 podemos observar un ejemplo de código binario generado para un alfabeto en código Huffman:



Símbolo	Código
A	0
B	100
C	101
D	110
E	111

Figura 1: Ejemplo de código binario generado para un alfabeto en código Huffman

A continuación se describen los pasos empleados para producir el código Huffman para un alfabeto y frecuencias determinadas:

- Tomamos un número n de símbolos. Cada símbolo tendrá una frecuencia de aparición adjunta.
- Los nodos del árbol binario los ordenaremos de mayor a menor frecuencia.
- Agruparemos en parejas los símbolos que aparezcan con una menor frecuencia y les asignaremos una suma de probabilidades al nodo padre, hasta que no queden nodos para unir a un nodo superior.

- Etiquetamos cada arista de las ramas del árbol, con unos a la derecha y ceros a la izquierda.
- Finalmente se produce el código Huffman para el alfabeto con sus frecuencias, recorriendo los bits desde la raíz hasta los nodos del árbol.

Por otro lado, la codificación aritmética[**11**] se trata de una técnica que tiene como objetivo codificar una secuencia de símbolos representando una secuencia binaria. Las secuencias de bits se obtienen de los intervalos cuyos valores reales oscilan entre uno y cero. Los pasos que se siguen a la hora de realizar esta técnica son los siguientes:

- Partimos de un alfabeto de número de símbolos n, teniendo en cuenta que cada símbolo tiene una frecuencia de aparición asociada.
- Calculamos la probabilidad acumulativa de los símbolos de cada secuencia.
- Asignamos un rango a cada símbolo, tomando como límite superior la probabilidad acumulativa del símbolo y como límite inferior la probabilidad del símbolo anterior de la secuencia.
- Finalmente traducimos el resultado obtenido a un código binario.

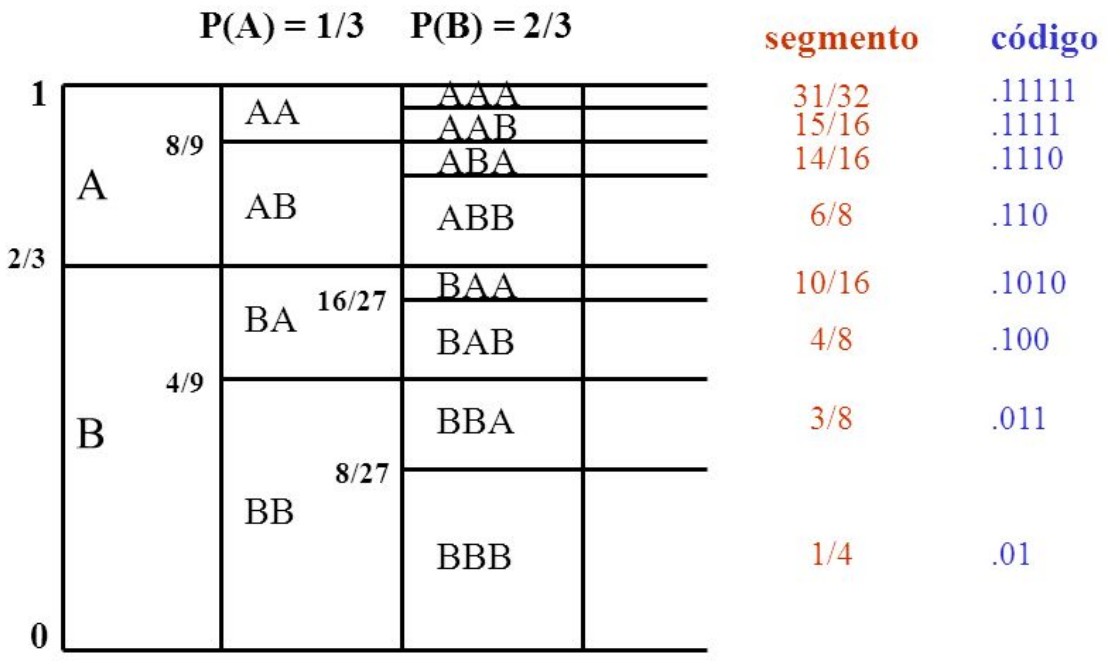


Figura 2: Ejemplo de codificación aritmética

El principal inconveniente de la codificación aritmética es que pese a tratarse de una técnica muy eficiente, algunos de sus algoritmos están patentados, como los algoritmos de Amir Said o Cabal.

1.3. Estándares de codificación

A continuación describiremos algunos de formatos de compresión más utilizados, así como sus diferencias:

1.3.1. Joint Photographic Experts Group(JPEG)

JPEG es un formato estándar ISO con pérdidas[**8**]. Este formato nos permite ajustar el grado de compresión, de manera que a niveles de compresión bajo, nos encontramos una imagen muy similar a la original, con un tamaño de archivo elevado. Sin embargo, si optamos por un grado de compresión alto, la imagen será de menor calidad que la original, pero con un tamaño de archivo bastante inferior que en el caso anterior. El proceso de codificación del formato JPEG, pasa por tres etapas[**2**], tal y como podemos apreciar en la figura 3.

Etapa 1: Preparación de la imagen La preparación de la imagen consta principalmente de tres pasos:

- Transformamos el espacio de color. Las imágenes de color son transformadas del espacio RGB al espacio YUV[**6**].
- Realizamos un submuestreo, que nos permite reducir el color para obtener ficheros en tamaño inferior a los originales, puesto que el ojo humano es menos sensible al color que al contraste.
- Dividimos la imagen en bloques de tamaño 8x8 pixeles.

Etapa 2: Codificación fuente

- Realizamos la transformación discreta de coseno(DCT) a cada bloque de 8x8 de la imagen, obteniendo un dominio de la frecuencia, es decir, una matriz de coeficientes que determina el valor de una señal a determinadas frecuencias.
- Realizamos la Cuantificación, donde cada coeficiente de la matriz se divide por una constante de una matriz de cuantificación y se redondea a su número entero más próximo. Eliminando los coeficientes menos representativos, introducimos, una pérdida de información de la imagen y, en consecuencia, se producirá una pérdida de calidad de la imagen reconstruida.

Etapa 3: Codificación entrópica

- Aplicamos la codificación Run-length[**9**], a todos los componentes del bloque.
- Aplicamos codificación Huffman al resultado obtenido en la codificación anterior.

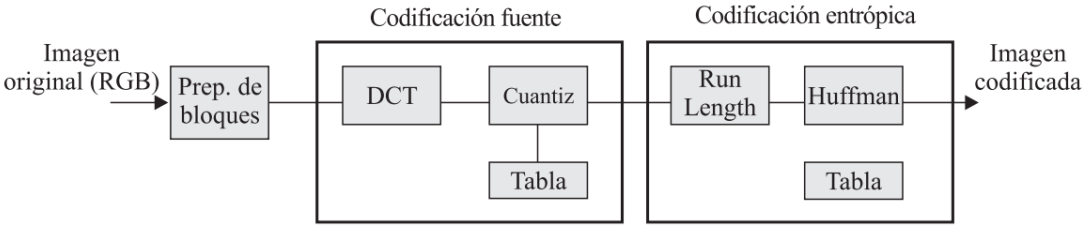


Figura 3: Esquema de compresión JPEG

1.3.2. JPEG2000

JPEG 2000[**5**] es un estándar de codificación de imágenes que usa técnicas de compresión de última generación basadas en la transformada Wavelet[**10**]. Su arquitectura es muy útil en infinidad de usos, desde cámaras digitales portátiles hasta avanzadas preimpresiones, imágenes médicas y muchos otros sectores.

Al igual que JPEG, fue creado por el Joint Photographic Experts Group en el año 2000, con el principal objetivo de ser el sustituto de JPEG, haciendo uso de la transformada Wavelet en vez de la transformada discreta de coseno que utilizaba el anterior formato JPEG.

Una de las mayores ventajas de JPEG2000 es que tiene la opción de que el usuario pueda seleccionar un área de interés pudiendo transmitir esta zona sin la necesidad de incluir en la transmisión el resto de la información de la imagen. En caso de imágenes en las que se necesita visualizar un área en concreto con más detalle, este estándar nos permite realizar una trasmisión muy eficiente de la imagen, minimizando el detalle de aquellas áreas que no son de importancia para el usuario. Por contra, esta selección de una determinada área de la imagen implica que tendremos que hacer uso de codificadores y, en consecuencia, decodificadores más complejos que a su vez, harán uso de una mayor utilización de recursos computacionalmente hablando.

Debido a que JPEG2000 está protegido por patentes y su implementación es más costosa en recursos que JPEG , no existen muchos navegadores actualmente que lo soporten, por lo que su uso no está muy extendido, excepto en caso de aplicaciones concretas, en las que por algún motivo técnico se requiera una mayor calidad de imagen o hacer uso de alguna funcionalidad concreta de JPEG2000, como es el caso de la industria del cine digital.

2. Algoritmo Perceptually Enhanced Lower Tree Wavelet (PETW)

2.1. Fundamentos teóricos del algoritmo PETW

El algoritmo PETW[12] es un algoritmo de compresión de imágenes cuya ventaja radica en la eficiencia a la hora de agrupar coeficientes y a su alta velocidad de codificación, puesto que al contrario de algoritmos como JPEG2000, PETW es menos complejo computacionalmente hablando.

Todas las imágenes digitales, a no ser que sean vectoriales, generalmente están compuestas por un grupo o conjunto de elementos llamados pixeles. Pues bien, de este conjunto o grupo de pixeles se obtiene un grupo de coeficientes(C) que se obtienen a partir de la transformada Wavelet. A continuación podemos observar de forma gráfica el proceso de codificación del algoritmo LTW[4] en la figura 4, teniendo en cuenta que la diferencia entre este y PETW reside en la elevación perceptual, concepto que sera explicado a lo largo del desarrollo de este proyecto.

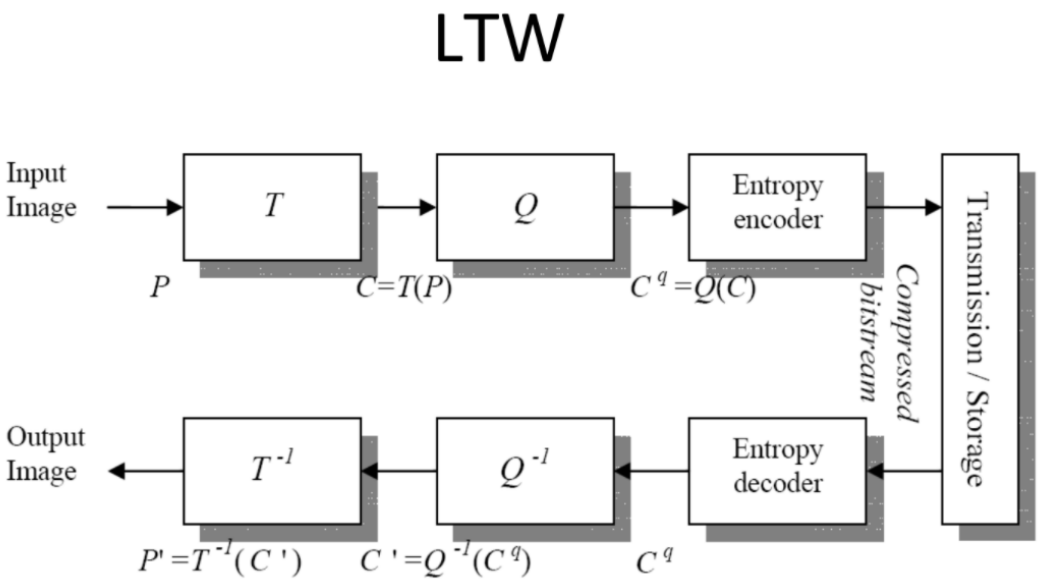


Figura 4: Ejemplo de las etapas de codificación y decodificación del algoritmo LTW

Como podemos observar una vez que la imagen es transformada, se obtienen los coeficientes Wawelet. Dichos coeficientes representan a diferentes versiones de la imagen a distintas frecuencias, tal y como podemos observar en la figura 5, la descomposición en dos niveles de la imagen Lena.

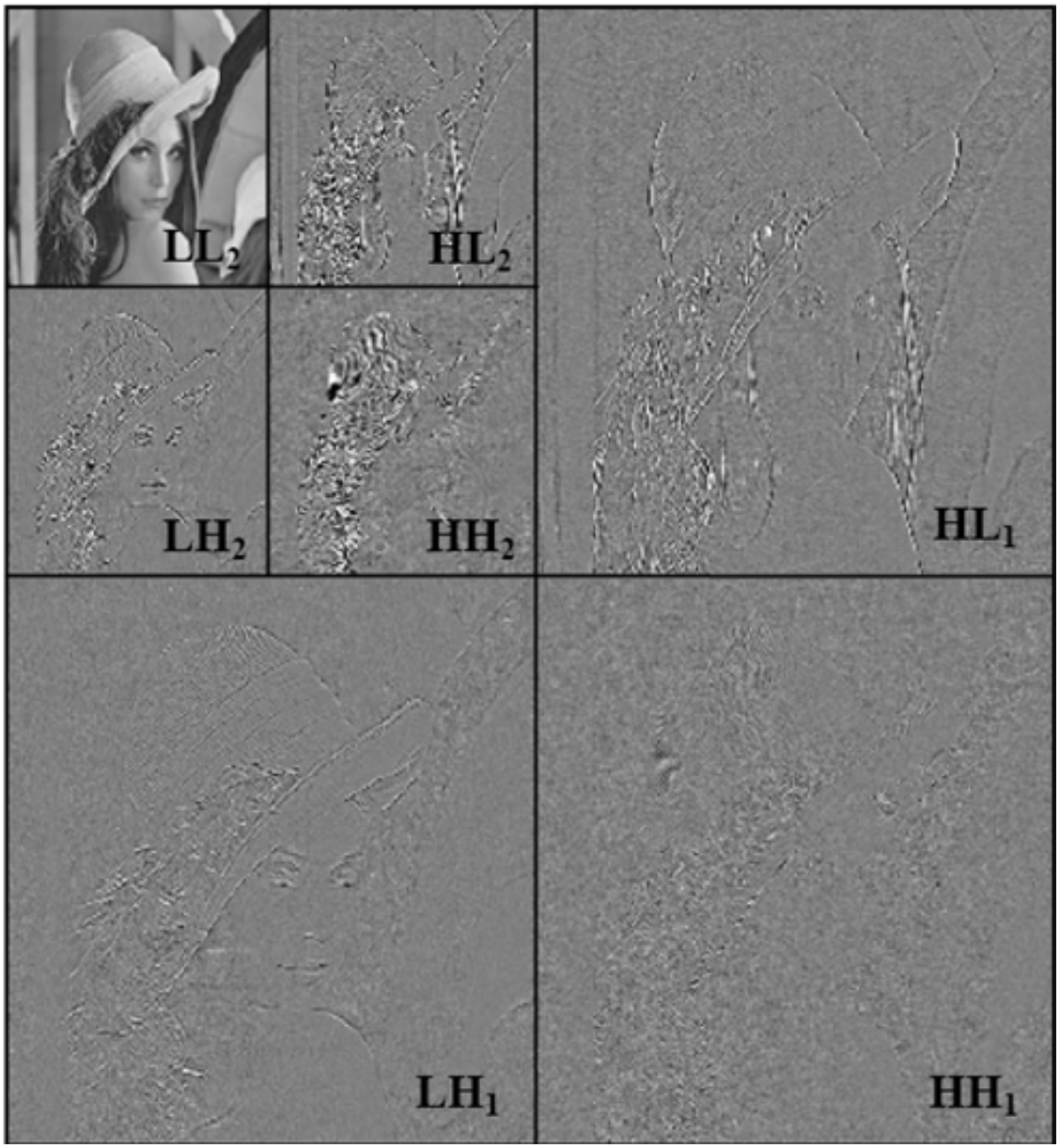


Figura 5: Ejemplo de descomposición de la imagen Lenna en distintos niveles

De los primeros niveles de descomposición a una escala determinada de la imagen LH_1,HH_1 y HL_1 se obtienen subbandas de frecuencia y el resto de niveles se obtienen, aplicando recursivamente la transformada Wavelet a la subbanda LL del nivel anterior hasta obtener el nivel de descomposición N deseado. En este proceso de descomposición se aplicara concretamente el filtro Daubechies 9/7F**[10]**, al igual que JPEG2000 en el caso de codificación con pérdidas.

La transformada Wavelet de Daubechies está basada en el trabajo de la física y matemática Ingrid Daubechies. Es en realidad una familia de Wavelets ortogonales que definen la transformada Wavelet discreta y están caracterizadas por un número máximo de momentos de desaparición para un soporte dado. En la figura 6, podemos ver un ejemplo gráfico de la transformada Wawelet aplicando el filtro de Daubechies.

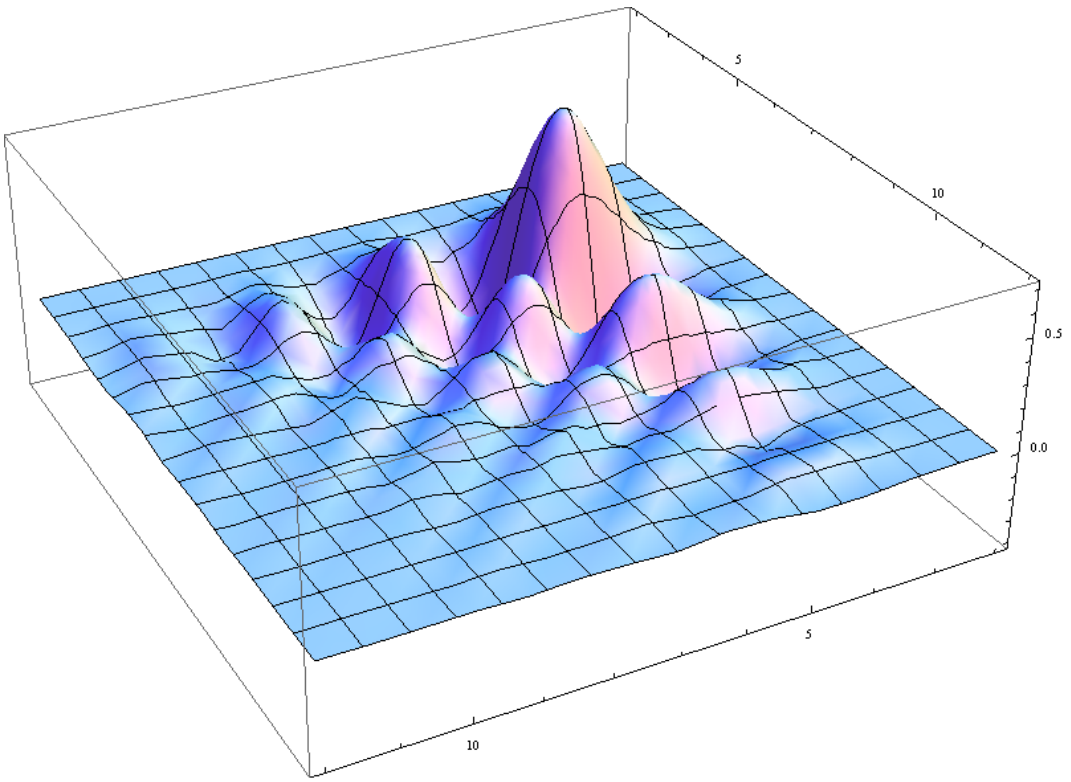


Figura 6: Ejemplo gráfico de la transformada de Wavelet aplicando el filtro de Daubechies

Una vez realizada la transformada Wavelet, entramos en la fase o proceso de cuantificación, en el que usan dos parámetros fundamentalmente en el algoritmo LTW:

- **Parámetro Q:** Se trata de una cuantización fina en la que se realiza una cuantización escalar uniforme que se aplica a todos los coeficientes Wavelet excepto a la subbanda de frecuencia LL.
- **Rplanes:** Se trata de una cuantización gruesa, en la se realiza una eliminación de los *"rplanes"* bits menos significativos de los coeficientes Wavelet. En la versión PETW, este parámetro siempre será 0, usando únicamente el parámetro Q.

Finalmente, la última etapa del codificador consiste en la aplicación de la codificación entrópica que no es más que un método de codificación sin perdidas que es independiente a las características concretas de un medio dado, tras realizar un proceso de agrupación de coeficientes en un árbol de 4 nodos *"quadtree"*. A continuación, explicaremos de forma más detallada el funcionamiento del algoritmo en pseudocódigo con el objetivo de comprender mejor su funcionamiento, tal y como podemos observar en la figura 7.

function LowerTreeWaveletCoding( )

1) Initialization:

output *rplanes*

output $maxplane = \max_{\forall c_{i,j} \in C} \left\{ \left\lceil \log_2 \left(\left| c_{i,j} \right| \right) \right\rceil \right\}$

2) First image pass:

LTWCalculateSymbols( )

3) Second image pass:

LTWOutputCoefficients( )

end of fuction

Figura 7: pseudocódigo del algoritmo LTW

Tras la transformada y la cuantificación de los coeficientes Wavelet, se produce la inicialización en la que se define el maxplane, que no es más que el número máximo de bits del valor mayor de todos los coeficientes Wavelet de la imagen. Posteriormente se llama al método **LTWCalculateSymbols()** que podemos observar en la figura 8.

subfunction LTWCalculateSymbols( )
Scan the first level subbands (HH_1, LH_1 and HL_1) in 2×2 blocks.
For each block B_n
  if $|c_{i,j}| < 2^{rplanes} \ \forall c_{i,j} \in B_n$
    set $c_{i,j} = LOWER\_COMPONENT \ \forall c_{i,j} \in B_n$
  else
    for each $c_{i,j} \in B_n$
      if $|c_{i,j}| < 2^{rplanes}$
        set $c_{i,j} = LOWER$
Scan the rest of subbands (from level 2 to N) in 2×2 blocks.
For each block B_n
  if ($|c_{i,j}| < 2^{rplanes} \wedge$ descendant($c_{i,j}$)=$LOWER\_COMPONENT$) $\forall c_{i,j} \in B_n$
    set $c_{i,j} = LOWER\_COMPONENT \ \forall c_{i,j} \in B_n$
  else
  for each $c_{i,j} \in B_n$
    if $|c_{i,j}| < 2^{rplanes} \wedge$ descendant($c_{i,j}$)=$LOWER\_COMPONENT$
      set $c_{i,j} = LOWER$
    if $|c_{i,j}| < 2^{rplanes} \wedge$ descendant($c_{i,j}$)$\neq LOWER\_COMPONENT$
      set $c_{i,j} = ISOLATED\_LOWER$
end of subfuction

Figura 8: pseudocódigo la función LTWCalculateSymbols()

Como podemos observar en primer lugar al ver el pseudocódigo de este método, es que esta diseñado de forma recursiva, formando una estructura de árbol(lower-tree) que va desde los nodos hojas hasta la raíz. Como podemos observar en la figura 8, las subbandas de frecuencia del primer nivel son analizadas por bloques, de tamaño de 2x2 y en caso de que los 4 coeficientes no sean significativos es decir, sean menores de $2^{rplanes}$, se consideran parte del mismo árbol inferior, etiquetado como **LOWER_COMPONENT**.Al escanear el nivel superior en las subbandas, si un bloque 2x2 tiene cuatro coeficientes no significativos y todos sus descendientes, serán etiquetados también con **LOWER_COMPONENT**, aumentando el tamaño del árbol inferior. Sin embargo, cuando al menos un coeficiente en un boque es significativo, el árbol inferior no puede seguir creciendo. En ese caso, cada coeficiente no significativo se etiquetara como LOWER, si todos sus descendientes son **LOWER_COMPONENT**, de lo contrario, los coeficientes no significativos se etiquetan como **ISOLATED_LOWER.**

Una vez que este método ha terminado, tal y como podemos observar en la figura 7, se llama al método **LTWOutputCoefficients()**, cuyo pseudocódigo podemos observar a continuación:

subfunction LTWOutputCoefficients( )
Scan the subbands (from N to 1, in 2×2 blocks)
For each $c_{i,j}$ in a subband
if $c_{i,j} \neq LOWER\_COMPONENT$
if $c_{i,j} = LOWER$
arithmetic_output *LOWER*
else if $c_{i,j} = ISOLATED\_LOWER$
arithmetic_output *ISOLATED_LOWER*
else
$nbits_{i,j} = \lceil \log_2(|c_{i,j}|) \rceil$
if descendant($c_{i,j}$) $\neq LOWER\_COMPONENT$
arithmetic_output $nbits_{i,j}$
else
arithmetic_output $nbits_{i,j}^{LOWER}$
output $\text{bit}_{nbits_{i,j}-1}(|c_{i,j}|) \ldots \text{bit}_{rplane+1}(|c_{i,j}|)$
output sign($c_{i,j}$)
end of subfuction

Note: $\text{bit}_n(c)$ is a function that returns the n^{th} bit of c.

Figura 9: pseudocódigo la función LTWOutputCoefficients()

Como podemos observar en este método, se exploran todas las subbandas de frecuencia desde el nivel N hasta el nivel 1. Este es el momento en el que entra en juego las etiquetas del método descrito anteriormente, puesto que cada coeficiente se codifica de una forma en función de la etiqueta que tenga. Para el caso de los coeficientes etiquetados con **LOWER_COMPONENT**, no tendrá salida, puesto que dichos coeficientes ya son representados en el árbol correspondiente por un símbolo de subbanda del nivel anterior. Como podemos observar en la nota cuando un coeficiente es significativo, se puede codificar de dos formas distintas:

- Con un entero que represente el número de bits necesarios($nbits_{i,j}$) para representarlo.
- Con el símbolo $nbits_{i,j}^{LOWER}$ que aparte de representar el número de bits necesarios, indica que todos sus descendientes son no-significativos quedando todos ellos representados por este valor.

2.2. Elevación perceptual

La principal diferencia entre LTW y PETW es la elevación perceptual[**12**].En un esquema de codificación adaptativo, los coeficientes se cuantifican usando un cuantificador como expuesto, y los coeficientes cuantificados son generalmente codificados por entropía para reducir la redundancia. La mayoría de los codificadores independientemente de la transformada matemática que usen, determinan el valor umbral para que los coeficientes sean inferiores a cero. A medida que se establecen más coeficientes a cero, la calidad de la imagen reconstruida se deteriora. Sin embargo, la forma en la que la calidad de la imagen se ve afectada no solo depende del número de coeficientes no nulos retenidos sino también sobre qué coeficientes se descartan, es decir, algunos coeficientes son perceptualmente más importantes que otros.

Otra indicación es determinar el tamaño de paso de cuantificación apropiado en cada caso. Una vez que los coeficientes han sido anulados, el resto de coeficientes no nulos se cuantifican para reducir el número de bits. La sobrecuantización de los coeficientes correspondientes a diferentes frecuencias espaciales afecta a la imagen reconstruida de diferentes maneras. Por ejemplo, la sobrecuantización de los coeficientes DCT de baja frecuencia causan bloqueo, mientras que los pasos de cuantificación grandes a frecuencias más altas hacen que el ruido aleatorio se vuelva visible.

2.3. Prueba del codificador PETW en el sistema operativo WINDOWS

A continuación vamos a comprobar el funcionamiento del codificador PETW sobre el sistema operativo WINDOWS 7, para ello trabajaremos en una carpeta que contendrá los siguientes elementos:

- **PETW.exe:** Es el ejecutable del algoritmo encargado de procesar la imagen
- **Comprimir.bat:** Fichero batch encargado de pasarle los parametros y la ubicación de la imagen a PETW.exe
- **Barbara.x:** Imagen de prueba en formato x, pudiendo ser x, bmp,raw, jpeg...
- **Foto.dng:** Imagen de prueba en formato dng

En la siguiente captura podemos observar el espacio de trabajo:

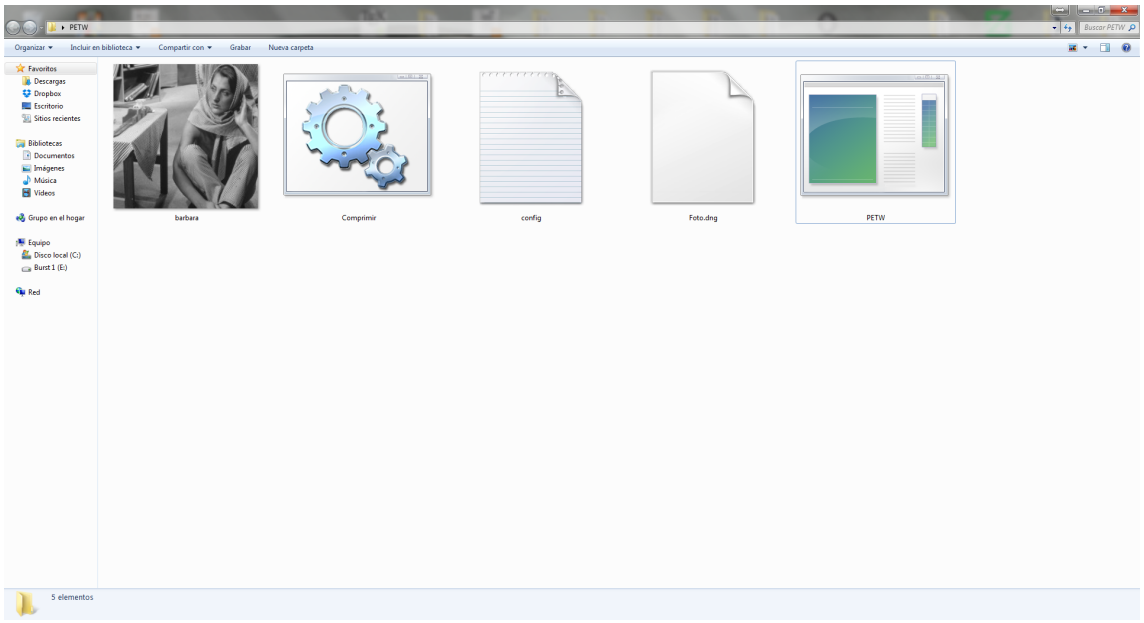


Figura 10: Espacio de trabajo para las pruebas del algoritmo PETW en WINDOWS

Las pruebas se realizaran con dos fotos barbara.bmp y Foto.raw(Formato RAW), para demostrar que el algoritmo es funcional y detallar su funcionamiento, en primer lugar tenemos que configurar el fichero batch con los siguientes parámetros:

```
PETW.exe -C -i Foto.raw -o salida.bts -w 5296 -h 2072  -Q 0.8 -Xi 0.375
    -d 0.6
```

A continuación detallaremos cada parámetro:

- **-C:** Parámetro encargado de establecer el algoritmo en modo codificación, para decodificación se usaría **-D**
- **-i:** Especifica el fichero imagen de entrada, en este caso foto.raw

- **-o:** Especifica el fichero de salida(bitstream), en este caso salida.bts
- **-w:** Establece el ancho de la imagen
- **-h:** Establece el alto de la imagen
- **-Q:** Parámetro que establece la tasa de compresión
- **-d:** Parámetro delta que establece el punto de recuperación por defecto, su valor debe estar comprendido siempre entre 0 y 1

NOTA: Es indispensable que el tamaño de la altura y anchura de la imagen será correctos o por el contrario el algoritmo no procesara la imagen.

Una vez que tenemos el bach listo y la imagen que vamos a procesar en el directorio, simplemente ejecutamos Comprimir.bat, en este caso para Foto.dng:

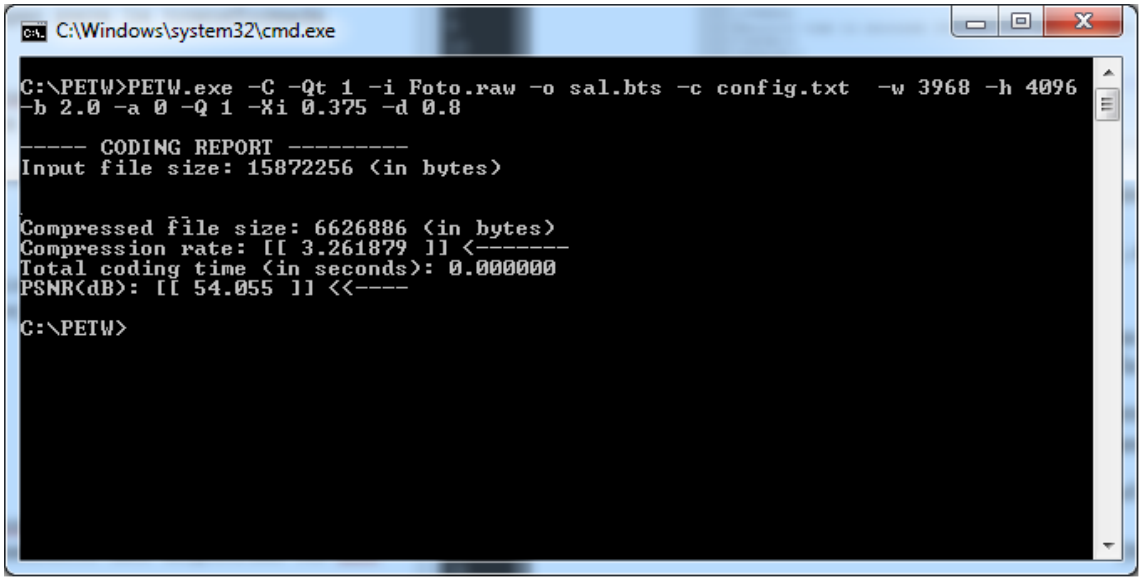


Figura 11: Salida por consola del procesamiento del algoritmo PETW sobre Foto.raw

3. Kit de desarrollo nativo(NDK)

3.1. Introducción

NDK[**1**] es una herramienta que nos permite utilizar código C/C++ a través de Java Native Interface(JNI) concepto que en la siguiente sección. El uso de código nativo C/C++ tiene como consecuencia en una aplicación escrita en lenguaje JAVA, una ejecución más rápida y un mejor uso de los recursos, puesto que, a diferencia de JAVA, el código nativo no es ejecutado en una máquina virtual JAVA(JVM) sino que es interpretado directamente por el procesador. Debido a este hecho NDK suele ser más usado en aplicaciones que realizan un alto uso del microprocesador, como pueden ser videojuegos o aplicaciones que tienen un elevado coste computacional. Existen numerosas aplicaciones muy famosas en el mercado que hacen uso de NDK como por ejemplo WhatsApp o Skype.

Si realizamos un proyecto desde cero no es recomendable el uso de NDK, a no ser que queramos un aumento de rendimiento o queramos reutilizar código/librerías que ya tenemos escritas en C/C++ y queremos utilizarlas sin necesidad de volver a reescribirlas en java, puesto que NDK de manera inevitable sumara complejidad a cualquier proyecto.

3.2. Java Native Interface(JNI)

3.2.1. Introducción

JNI[**14**] es una herramienta que nos permite comunicar de forma bidireccional código java con código nativo ya sea C, C++. Se utilizan para aplicaciones de bajo nivel como, por ejemplo, operaciones de escritura y la lectura de ficheros.

A través de JNI, podemos confeccionar una biblioteca de enlace dinámico a través de métodos nativos de forma que una o varias aplicaciones pueden compartir la misma biblioteca haciendo uso del código escrito en C/C++, lo cual es muy potente, puesto que podemos reutilizar el mismo código una vez generada la biblioteca para las aplicaciones que queramos. Es necesario remarcar que cada sistema operativo tiene su propia arquitectura y configuraciones determinadas por que el uso y manejo de estas bibliotecas, así como la eficiencia y rendimiento que aportan se verán afectados por el sistema operativo en cuestión. Dichas bibliotecas han de ser encontradas en directorios determinados en función del sistema operativo, por lo que no será la misma configuración de una biblioteca para UNIX que para WINDOWS.

Una de las ventajas más importantes de JNI es que no impone ningún tipo de restricción sobre la implementación de la máquina virtual JVM subyacente. De modo que los desarrolladores de JVM puede añadir soporte sin afectar a otras partes de la misma, además los programadores pueden lanzar una aplicación o biblioteca nativa y posteriormente ser utilizada como hemos explicado anteriormente por cualquier tipo JVM que tenga soporte para JNI.

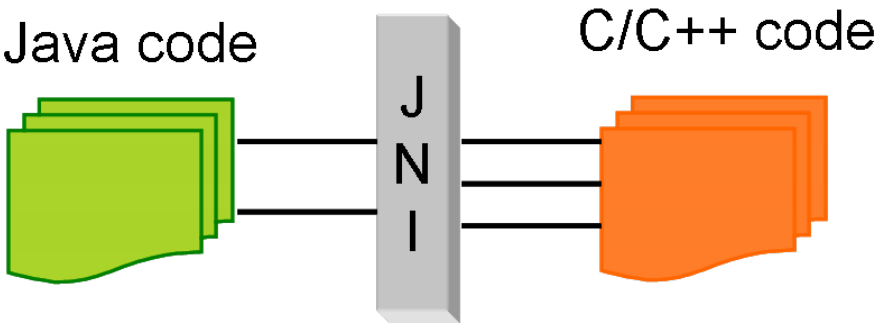


Figura 12: Ejemplo gráfico de aplicación de JNI

3.2.2. Situaciones en las que es necesario el uso de JNI

Aunque todas las aplicaciones puede ser escritas completamente en JAVA, siempre podemos encontrarnos con situaciones en las que JAVA no satisfaga nuestras necesidades, ya sea en rendimiento o en reutilización de código o librerías, A continuación se muestran algunos ejemplos en los cuales puede ser muy útil hacer uso de JNI:

- Ya tenemos una biblioteca escrita en otro lenguaje y queremos hacer uso de ella en una aplicación JAVA a través de JNI
- Nos encontramos con la obligación o la necesidad de implementar una parte del código de forma nativa ya sea por tiempo o porque necesitamos hacer uso de un lenguaje de bajo nivel como puede ser ensamblador.
- Nos encontramos con que las bibliotecas JAVA estándar no admiten funciones que son dependientes de la plataforma para la que estamos desarrollando la aplicación.

Poniéndonos desde el punto de vista del código nativo, podemos usar JNI por las siguientes razones:

- Llamar desde el código nativo a métodos java e interactuar con ellos.
- Capturas y lanzar excepción para controlar los errores y poder localizarlos
- Cargar clases JAVA y así como obtener información de ellas.
- Crear, inspeccionar y actualizar objetos JAVA de todo tipo.

3.2.3. Objetivos

Los objetivos que pretende conseguir JNI son los siguientes:

- Que los desarrolladores de herramientas no necesiten hacer uso de distintos tipos de interfaces de métodos nativos
- Que los desarrolladores de aplicaciones puedan escribir dicha aplicación en código nativa y esta pueda ser ejecutada en diferentes máquinas virtuales java.

- Que las máquinas virtuales java puedan contener cada más código nativo.
- Compatibilidad binaria: Conseguir la compatibilidad de binaria de todas las bibliotecas dinámicas de enlace de métodos nativa para todas las implementaciones de la máquina virtual java posibles.
- Buscar el mínimo impacto sobre el rendimiento y el uso de recursos
- Ser capaz de conseguir que los métodos nativos puedan ser capaces de desarrollar cualquier tipo de tarea sin ningún tipo de limitación por parte de JNI

3.3. Compilación y estructura del proyecto

A la hora de incluir en nuestro proyecto Android, código C/C++ mediante el uso de NDK, tenemos que ser conscientes de que dicha implementación tendrá como consecuencia un cambio en la estructura de nuestro proyecto. Para poder comprender cuales son dichos cambios y cuales la estructura, así como familiarizarnos con los conceptos de NDK, a continuación, describiremos la estructura del proyecto:

- Crear un proyecto en Android Studio
- Crear una carpeta JNI dentro de la carpeta app del proyecto, donde ira incluido todo el código C/C++ con extensión .cpp en nuestro caso, junto a la librerías en caso de ser necesarias.
- Haciendo uso de javah y javac crearemos el fichero de cabecera .h necesario para poder crear la biblioteca de enlace dinámica a no ser que añadamos la línea **extern** *"C"* , en cuyo caso no sera necesario el uso de javah
- Crear el código C/C++ o incluirlo en la carpeta JNI
- Ejecutar ndk-build para compilar el código nativo y producir las bibliotecas finales.

En primer lugar creamos el proyecto en Android Studio, teniendo la siguiente estructura, como podemos observar en la figura 14.
Una vez creado el proyecto, crearemos una clase java que se llamara PETW de forma que será esa clase la encargada de llamar la función compresión escrita en código nativo, dicha función nos devolverá la cadena de caracteres que mostraremos por pantalla, indicándonos que el código nativo ha sido ejecutado con éxito.
Tal y como podemos observar en la figura 15, cargamos en la Biblioteca de enlace dinámica en este caso la llamaremos librería, la cual contiene el método. Como podemos observar la estructura de la clase es sencilla, puesto que simplemente cargaremos la librería **ndkmain** y declararemos el método compresión que nos devolverá un string como hemos descrito anteriormente comunicándonos que el código nativo se ha ejecutado correctamente y recibirá como único argumento un array de strings, esto es así, puesto que emularemos dentro de NDK la consola de comandos, de cualquier sistema operativo como puede ser WINDOWS o LINUX

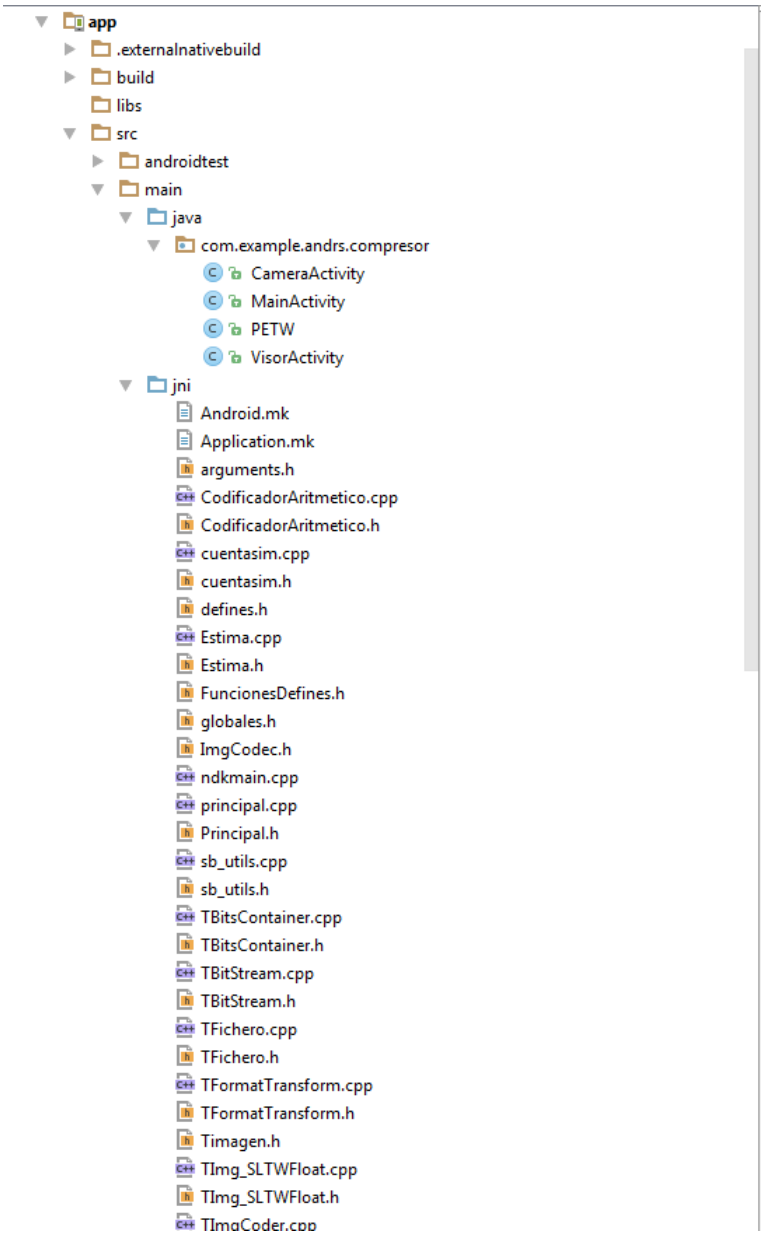


Figura 13: Estructura inicial del proyecto compresor PETW en NDK

```
package com.example.andrs.compresor;


public class PETW  {

    static {

        System.loadLibrary("ndkmain");
    }

    public native String compresion(String[] argv,int[] array, int anchomod, int altomod);

    public  native String descompresion(String[] argv);

}
```

Figura 14: Estructura de la clase PETW

de forma que no alteraremos la estructura del proyecto primigenio escrito en C++ y, además mantendremos todas las funcionalidades sin alterar apenas, la parte de recepción de argumentos del código.

Para ello, crearemos un fichero llamado ndkmain.cpp que simulara la función main de C de forma que los argumentos será introducidos a través de una interfaz gráfica, que recibirá la clase MainActivity de forma que dichos argumentos serán cargados en un array de strings y serán pasados directamente al código nativo, tal y como podemos observar:

```
String[] arguments = new String[]{"-C", "-Qt", "1", "-i", path + "/"+
    m_entrada+".petw", "-o", path+ "/"+m_salida+".raw", "-c", path + "/
    config.txt", "-w", "3840", "-h", "2048", "-b","2.0","-a","0","-Q","
    0.873","-Xi","0.375","-d","0.8"};

texto.setText(petw.descompresion(arguments));
```

Dentro del método oncreate() de Android, definiremos un TextView y un Button, de forma que cuando el usuario le dé al botón de compresión, se comprimirá perceptualmente la imagen tomada en formato RAW mediante la cámara de la misma.

Nota: Es importante destacar que a la hora de pasar el path o ruta al código nativo ya no sirven las rutas de tipo *"/sdcard/barbara.raw"*, puesto que debido a las actualizaciones de las apis de Android ya no puede escribirse la ruta directa y esta debe obtenerse a través del método *getExternalFilesDir*(*null*) que nos proporcionara la ruta correspondiente a nuestra aplicación cuyo nombre corresponde con el nombre de package o paquete, de la misma.

Una vez obtenida la ruta, crearemos un objeto de la clase PETW y en la variable *arguments*, introduciremos los argumentos necesarios.En este caso a modo de ejemplo, introducimos los parámetros directamente de forma manual, pero en el caso real dichos parámetros serán tomados de una interfaz gráfica, como podemos observar en la figura 16.

3.3.1. Emulación de la consola de comandos en código nativo C++ para Android

Para la emulación de los comandos de consola, nos creamos un archivo ndkmain.cpp que será nuestro módulo principal, tal y como podemos ver a continuación:

```
extern "C"
JNIEXPORT jstring JNICALL Java_com_example_andrs_compresor_PETW_compresion(JNIEnv* env, jobject thiz , jobjectArray jargv,jintArray arr,
                                                                           int anchomod,int altomod){

    // initializations, declarations, etc
    int *c_array;
    int l=env->GetArrayLength(arr);

    // get a pointer to the array
    c_array = env->GetIntArrayElements(arr, NULL);

    // do some exception checking
    if (c_array == NULL) {
        LOGI("Error en la obtencion del array");
    }

        //jargv is a Java array of Java strings
        int argc = env->GetArrayLength(jargv);
        typedef char *pchar;
        pchar *argv = new pchar[argc];
        int i;

        for (i = 0; i < argc; i++) {
            jstring js = (jstring) env->GetObjectArrayElement(jargv, i); //A Java string
            const char *pjc = env->GetStringUTFChars(js,
                                                     NULL); //A pointer to a Java-managed char buffer
            size_t jslen = strlen(pjc);
            argv[i] = new char[jslen + 1]; //Extra char for the terminating null
            strcpy(argv[i],
                   pjc); //Copy to *our* buffer. We could omit that, but IMHO this is cleaner. Also, const correctness.
            env->ReleaseStringUTFChars(js, pjc);
        }

        //Call Principal main
        Principal *pa;
        pa = Principal::CreateInstance(argc, argv);
        pa->arguments.pcarray = c_array;
        pa->Run();
        pa->FreeInstance();

        //Now free the array
        for (i = 0; i < argc; i++)

            delete[] argv[i];

        delete[] argv;

        return env->NewStringUTF("El código nativo se ha ejecutado correctamente");
```

Figura 15: Estructura del fichero ndkmain.cpp

En primer lugar, una vez recibimos los parámetros necesarios para la codificación, nos creamos la variable entera argc, que tendrá como objetivo, simular el argc de C++, dicho parámetro será en realidad, el número total de paramentos que estamos introduciendo. Una vez definido argc, crearemos un estructura de tipo puntero a char, que posteriormente apuntara a un array que contendrá los parámetros y se rellenara mediante un bucle for, simulando el argv de Android, de forma que al final llamamos a la función $Principal :: CreateInstance$, esta función, crea una instancia/objeto del compresor PETW, pasándole los parámetros argc y argv. Este será nuestro módulo principal, de forma que será el encargado de llamar al código nativo C++ del algoritmo de compresión, pasándole los parámetros correspondientes y simulando a su vez la consola de comandos.

3.3.2. Escritura de ficheros en Android NDK desde código nativo

La escritura de ficheros es un tema fundamental a la hora implementar el algoritmo de compresión sobre código nativo, puesto que estamos trabajando sobre un sistema operativo(SO) distinto(Android)a WINDOWS, sin embargo, realmente no hay diferencia entre escribir un fichero en C++ y en Java, puesto que el problema de la escritura y lectura de ficheros se reduce en NDK a establecer la ruta correcta para dichas operaciones, así como tener los permisos necesarios. En nuestro caso tendremos escribir un fichero *"salida.bts"* que contendrá la imagen comprimida, para ello y como hemos descrito en secciones anteriores haremos uso del método $getExternalFilesDir(null)$ que nos devolverá la siguiente ruta *"/storage/.../Android/data/nombrepaquete.compresor/files/"*. Esta ruta nos servirá tanto para leer como para escribir ficheros.

3.3.3. Depuración de código nativo C/C++ en un proyecto Android

A la hora depurar un proyecto Android, se nos presenta un gran problema que es la depuración del código nativo en tiempo de ejecución con el fin de encontrar los problemas que puedan aparecer. En el entorno de desarrollo Android Studio existe el depurador de código nativo LLDB pero este debugger es completamente experimental y no funciona bien en proyectos complejos cuando hacemos uso de la secuencia de comandos ndk-build para compilar nuestro proyecto, y esto nos supone un gran problema puesto que *"logcat"* de Android, que es donde aparecen todos los problemas y logs que tienen que ver con la depuración , no aparecerá nada referente a nuestro código C++.La solución que se propone en este proyecto para la depuración del código nativo es la generación de una libreria escrita en C++ que haga uso de la librería ya existente *"android/log.h"* con el fin de poder realizar logs de cualquier tipo de variable a modo de printf de C o cout de C++ con el objetivo de poder ver en *"logcat"* de Android Studio el valor de estas variables y poder detectar así de una forma muy rudimentaria donde se encuentra los problemas en tiempo de ejecución. A continuación, en la siguiente captura podemos observar la estructura de esta librería a la que llamaremos trace.h:

```
#include <android/log.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>


#ifdef __ANDROID__
#define LOG_TAG "MyNative"
#define STRINGIFY(x) #x
#define LOG_TAG    __FILE__ ":" STRINGIFY(__MyNative__)
#define LOGI(...)  __android_log_print(ANDROID_LOG_INFO,LOG_TAG,__VA_ARGS__)
#define LOGD(...)  __android_log_print(ANDROID_LOG_DEBUG,LOG_TAG,__VA_ARGS__)
#define LOGE(...)  __android_log_print(ANDROID_LOG_ERROR,LOG_TAG,__VA_ARGS__)


#endif
```

Figura 16: Estructura de la Liberia trace.h

A la hora de realizar la depuración, lo que haremos será usar la palabra reservada *"LOGI"* a modo de printf para poder visualizar en el logcat de Android Studio si realmente todas las variables tienen una asignación correcta, así como encontrar y solucionar los errores que vayan apareciendo en tiempo de ejecución. A continuación, realizaremos un ejemplo con el objetivo de verificar si el entero argc y el vector argv son correctos, en el ejemplo hemos pasado 13 parámetros, por lo que argc tiene que valer 13 y argv, en sus distintos valores de array tienen que mostrarnos uno por uno, todos los parámetros que hemos introducido:
Para ello, escribiremos el siguiente código justo en la recepción de los parámetros por el código nativo, es decir, dentro del fichero ndkmain.cpp:

```
LOGI("argc dentro de ndkmain.cpp vale %d",argc);
    for(int i=0;i<13;i++) {

        LOGI("argv[%d] ahora dentro de ndkmain.cpp vale %s",i,argv[
            i]);
```

Como podemos observar en la siguiente captura este método es totalmente valido y funciona correctamente.

```
__MyNative__: argc dentro de ndkmain.cpp vale 13
__MyNative__: argv[0] ahora dentro de ndkmain.cpp vale -C
__MyNative__: argv[1] ahora dentro de ndkmain.cpp vale -Qt
__MyNative__: argv[2] ahora dentro de ndkmain.cpp vale 1
__MyNative__: argv[3] ahora dentro de ndkmain.cpp vale -i
__MyNative__: argv[4] ahora dentro de ndkmain.cpp vale /stora
__MyNative__: argv[5] ahora dentro de ndkmain.cpp vale -o
__MyNative__: argv[6] ahora dentro de ndkmain.cpp vale /stora
__MyNative__: argv[7] ahora dentro de ndkmain.cpp vale -c
__MyNative__: argv[8] ahora dentro de ndkmain.cpp vale /stora
__MyNative__: argv[9] ahora dentro de ndkmain.cpp vale -w
__MyNative__: argv[10] ahora dentro de ndkmain.cpp vale 512
__MyNative__: argv[11] ahora dentro de ndkmain.cpp vale -h
__MyNative__: argv[12] ahora dentro de ndkmain.cpp vale 512
```

Figura 17: Logcat de Android Studio mostrando los valores de argc y argv

Nota: Para poder hacer uso la librería *"android/log.h"*, es necesario declarar el flag correspondiente en el archivo Android.mk, en este caso *" – llog"*, en caso contrario no veremos nada en el logcat de Android Studio.

```
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)
LOCAL_MODULE := ndkmain
LOCAL_SRC_FILES :=ndkmain.cpp CodificadorAritmetico.cpp cuentasim.cpp
LOCAL_CPP_FEATURES += exceptions
LOCAL_LDLIBS := -llog -landroid
LOCAL_LDFLAGS := -Wl,--allow-multiple-definition
LOCAL_CFLAGS := -DANDROID_NDK  -Wno-psabi -DGL_GLEXT_PROTOTYPES=1
include $(BUILD_SHARED_LIBRARY)
```

Figura 18: Declaración del flag llog en el fichero Android.mk

3.3.4. Fichero Android.mk

Android.mk[15] es un pequeño fragmento del Makefile de GNU que sera analizado durante varias veces por el sistema en tiempo de compilación. Es importante que en él, minimizamos las variables que declaramos y no utilicemos ninguna que no sea estrictamente necesaria.

Este fichero se encuentra en un subdirectorio de nuestro proyecto, y describe los ficheros de origen y bibliotecas compartidas en el sistema de compilación. El archivo Android.mk es muy útil para definir configuraciones en todo el proyecto que Application.mk, el sistema de compilación y las variables de entorno dejan por defecto sin definir. También podemos anular configuraciones del proyecto para módulos concretos.

La sintaxis del fichero Android.mk nos permite agrupar los archivos de origen en módulos. Un módulo es una biblioteca estática, una biblioteca compartida o un ejecutable independiente. Podemos definir uno o más módulos en cada archivo Android.mk y usar el mismo archivo de origen en varios módulos. El sistema de compilación solo dispone bibliotecas compartidas en el paquete de la aplicación.

Además de empaquetar bibliotecas, el sistema de compilación es el encargado de muchos otros asuntos. Por ejemplo, no necesitamos indicar archivos de encabezados o dependencias específicas entre los archivos generados en nuestro fichero Android.mk. El sistema de compilación del NDK genera estas relaciones automáticamente por nosotros.

La sintaxis de este fichero es muy similar a la que se usa en los archivos Android.mk distribuidos con el Proyecto de código libre de Android completo. Si bien la implementación del sistema de compilación que los usa difiere, la similitud se encuentra en una decisión de diseño intencional orientada a hacer que los desarrolladores de aplicaciones puedan reutilizar, de manera más sencilla, el código fuente de bibliotecas externas.

La sintaxis de este fichero está diseñada con el fin de poder agrupar sus fuentes en módulos. Un módulo puede ser una biblioteca compartida o una biblioteca estática. Las bibliotecas estáticas pueden utilizarse a su vez para generar bibliotecas compartidas. Dentro de Andorid.mk, podemos definir uno o más módulos, pudiendo utilizar, si se desea, el mismo fichero Android.mk para todos los módulos. El sistema de compilación hace que no sea necesario listar de forma manual los archivos e cabecera ni las dependencias explicitas entre los archivos generado en Android.mk, puesto que el sistema de compilación de NDK lo realizar de forma automática. Esto a su vez hace que, si actualizamos a una versión más reciente de NDK, podamos utilizar las nuevas herramientas si tener que modificar la base de nuestro fichero Android.mk. A continuación, en la siguiente captura, vamos a visualizar el fichero Android.mk de nuestro proyecto en Android Studio con el fin de comprender su estructura y el significado de toda la sintaxis declarada:

```
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)
LOCAL_MODULE := ndkmain
LOCAL_SRC_FILES :=ndkmain.cpp CodificadorAritmetico.cpp cuentasim.cpp
LOCAL_CPP_FEATURES += exceptions
LOCAL_LDLIBS := -llog -landroid
LOCAL_LDFLAGS := -Wl,--allow-multiple-definition
LOCAL_CFLAGS := -DANDROID_NDK  -Wno-psabi -DGL_GLEXT_PROTOTYPES=1
include $(BUILD_SHARED_LIBRARY)
```

Figura 19: Estructura del fichero Android.mk

A continuación, vamos explicar las líneas de código que forman el fichero:

```
LOCAL_PATH := $(call my-dir)
```

Cualquier fichero Android.mk siempre debe comenzar con la definición de la variable LOCAL_PATH, esta variable se utiliza para localizar los archivos de origen en el árbol de desarrollo, en nuestro hemos utilizado la macro *"my-dir"*, proporcionada por el sistema de compilación, es utilizada para devolver la ruta del directorio actual, es decir, el directorio que contiene al propio fichero Android.mk

```
include $(CLEAR_VARS)
```

La variable CLEAR_VARS es proporcionada por el sistema de compilación y apunta a un Makefile GNU especial que borrará muchas variables LOCAL_XXX (por ejemplo, LOCAL_MODULE, LOCAL_SRC_FILES, LOCAL_STATIC_LIBRARIES, etc ...), con la excepción de LOCAL_PATH. Esto es necesario porque todos los archivos de control se analizan en un solo contexto de ejecución de GNU Make donde todas las variables son globales.

```
LOCAL_MODULE := ndkmain
```

La variable LOCAL_MODULE debe definirse para identificar cada módulo que describe en el fichero Android.mk. El nombre debe ser único y no contener espacios. Hay que tener en cuenta que el sistema de compilación agregará automáticamente los prefijos y sufijos adecuados al archivo generado correspondiente. es decir, un módulo de biblioteca compartida llamado 'ndkmain' generará 'libndkmain.so'.

```
LOCAL_SRC_FILES :=ndkmain.cpp CodificadorAritmetico.cpp cuentasim.cpp
    ...
```

Las variables LOCAL_SRC_FILES deben contener una lista de archivos de origen C/C++ que constituirán un módulo. Hay que tener en cuenta que no se deben listar los encabezados y los archivos incluidos aquí, porque el sistema de compilación calculará sus dependencias de forma automática.

```
LOCAL_CPP_FEATURES += exceptions
```

LOCAL_CPP_FEATURES es una variable opcional que se puede definir para indicar la extensión de archivo de C++. El valor predeterminado es .cpp, pero puede cambiarse. Utilizamos la palabra reservada exceptions para indicar que nuestro código utiliza excepciones de C++.

```
LOCAL_LDLIBS := -llog -landroid
```

LOCAL_LDLIBS es una variable que nos permite cargar librerías NDK ya existentes con el prefijo -l, en nuestro caso se trata de dos librerías más concretamente:

- -llog
- -landroid

llog es una librería que nos permite usar logs a modo de printf para poder mostrarlos en el logcat de Android. Por otra parte, landroid es una librería que nos permite indicar que el proyecto será específicamente compilado para el sistema operativo Android.

```
LOCAL_LDFLAGS := -Wl,--allow-multiple-definition
```

La variable LOCAL_LDFLAGS Puede pasar indicadores adicionales al vinculador. Hay que tener en cuenta que el orden de los parámetros es muy importante.

```
LOCAL_CFLAGS := -DANDROID_NDK -Wno-psabi -DGL_GLEXT_PROTOTYPES=1
```

La variable LOCAL_CFLAGS es útil para especificar definiciones de macros adicionales u opciones de compilación específicas.

```
include $(BUILD_SHARED_LIBRARY)
```

BUILD_SHARED_LIBRARY es una variable proporcionada por el sistema de compilación que apunta a un script de Makefile de GNU que se encarga de recoger toda la información que se definió en variables LOCAL_XXX desde el último 'include(CLEAR_VARS)' y determinar qué construir y cómo hacerlo exactamente. También podemos utilizar BUILD_STATIC_LIBRARY para generar una biblioteca estática.

3.3.5. Fichero Application.mk

El fichero Application.mk[**16**] es otro pequeño fragmento de Makefile de GNU que se encarga de definir las variables necesarias en el proceso de compilación. A continuación, podemos ver una captura, en la que podemos observar como se ha definido dicho fichero para este proyecto, así como explicar las líneas de código que lo forman:

```
APP_MODULES:=ndkmain
APP_ABI := all
APP_STL=stlport_static
```

Figura 20: Estructura del fichero Application.mk

APP_MODULES:=ndkmain

Si definimos esta variable, le estamos diciendo a la secuencia de comandos ndk-build que se construyan sólo los módulos correspondientes, compilando solo aquellos ficheros de los que depende el módulo. Los nombres de los módulos deben estar separados por espacios, en nuestro caso como podemos observar solo se va a compilar el módulo ndkmain.

Si esta variable no se define, la secuencia de comandos ndk-build buscara la lista de todos los módulos de nivel superior instalables, es decir, aquellos que se definen en el fichero Android.mk y cualquier archivo que vaya ligado a este directamente.

Un módulo instalable puede ser biblioteca compartida o ejecutable, que generará un archivo en el subdirectorio de liberias(libs). Si esta variable no está definida, y no hay módulos instalables de nivel superior en nuestro proyecto, ndk-build creara todas las bibliotecas estáticas de nivel superior y sus dependencias en su lugar. Estas bibliotecas se colocan en el subdirectorio de objetos(obj).

APP_ABI := all

Por defecto, el sistema de compilación de NDK generara código máquina para el armeabiABI. Este código máquina corresponde a una CPU basada en ARMv5TE con operaciones de punto flotante de software. Podemos utilizar esta variable para seleccionar un ABI diferente. En resumen, con esta variable definimos la arquitectura para la que será compilado nuestro módulo, con la palabra reservada *"all"*, estamos indicando que dicho módulo será compilado para todas las arquitecturas. A continuación se adjunta una tabla, sacada de la pagina web oficial de Android developers:

Conjunto de instrucciones	Valor
Instrucciones de Hardware FPU en dispositivos basados en ARMv7	`APP_ABI := armeabi-v7a`
ARMv8 AArch64	`APP_ABI := arm64-v8a`
IA-32	`APP_ABI := x86`
Intel64	`APP_ABI := x86_64`
MIPS32	`APP_ABI := mips`
MIPS64 (r6)	`APP_ABI := mips64`
Todos los conjuntos de instrucciones soportados	`APP_ABI := all`

Figura 21: Conjunto de instrucciones en función del valor asignado a la variable APP_ABI

APP_STL=stlport_static

Por defecto, el sistema de compilación de NDK nos proporcionara encabezados C++ para la biblioteca de tiempo de ejecución C++ mínima proporcionada por el sistema Android. Además, viene con implementaciones alternativas de C++ que podemos usar o enlazar en nuestras propias aplicaciones. Usamos la palabra reservada stlport_static para añadir bibliotecas precompiladas que nos permitan trabajar con excepciones de C++ y RTTI.

4. Herramientas de desarrollo

4.1. Introducción

En esta sección, vamos a explicar las herramientas de desarrollo que hemos usado para la construcción de este proyecto, así como una descripción del sistema operativo Android, con el fin de comprender como funciona, así como la integración de NDK en el entorno de desarrollo Android Studio.

4.2. Sistema operativo Android

4.2.1. Introducción

Android es un sistema operativo desarrollado por la compañía GOOGLE con el objetivo de liderar con él, el sector de la telefonía móvil, cosa que ha conseguido con creces, con actualmente más de un 86 % de la cuota de mercado. Numerosas marcas importantes como Samsung, HTC, LG etc. utilizan este sistema operativo en sus dispositivos ya sea en Tablets, Smartphone, Smartwatchs etc. Android es un sistema operativo de código abierto y aunque con cuenta con numerosas APIS oficiales también existen APIS de terceros e incluso nosotros mismos podemos tener acceso al código y modificarlo a nuestro gusto, si contamos claro está, con los conocimientos de programación necesarios. La programación Android esta basa principalmente en el lenguaje java, de forma que podemos incluir en nuestros proyectos sobre este sistema operativo cualquier librería o proyecto java de terceros o nuestro propio sin apenas problemas en su implementación. Al estar basado en java como es normal, Android utiliza algo muy similar a la máquina virtual de java, la máquina virtual Dalvik a diferencia de la máquina virtual java se pierde la capacidad de portabilidad a otras plataformas, con la ventaja de que se ejecutan con un mayor rendimiento y además con un menor consumo de energía, aunque dependiendo de la aplicación esto no siempre es posible. Aunque actualmente la máquina virtual Dalvik ha sido sustituida por Android Runtime, explicada más adelante.

Además desde la existencia de NDK , podemos incluir código nativo C/C++ en nuestro proyecto Android pudiendo llamar a los métodos nativos desde nuestras clases java, lo que nos permite usar numerosas librerías y código ya escrito en C, puesto que aunque la implementación de código nativo aumenta irremediablemente la complejidad de la aplicación a nivel de programación, nos proporciona una enorme cantidad de posibilidades que poco sistemas operativos tienen con la capacidad poder utilizarla en cualquier sitio, puesto que hoy en día el móvil siempre lo llevamos en el bolsillo.

Por otra parte, el sistema operativo Android está en constante actualización y aunque si bien esto puede ser un inconveniente para los desarrolladores en ciertos momentos porque pueden ser cambios muy grandes, le permite Android estar en la posición en la que se encuentra actualmente, siendo uno de los sistemas operativos que mayor adaptación tiene a las necesidades del mercado. Además de todo esto tenemos una amplia cantidad de recursos y de documentación en la red, por lo que es mucho más factible resolver cualquier problema a nivel de desarrollo que pueda surgir en Android que en cualquier otro sistema operativo como puede ser IOS de la compañia APPLE.

Para este proyecto se ha escogido el sistema operativo Android, porque el objetivo es que pueda ser usado por la mayoría de los dispositivos posibles, además de cómo hemos mencionado anteriormente la gran documentación de la que dispone.

4.2.2. Arquitectura

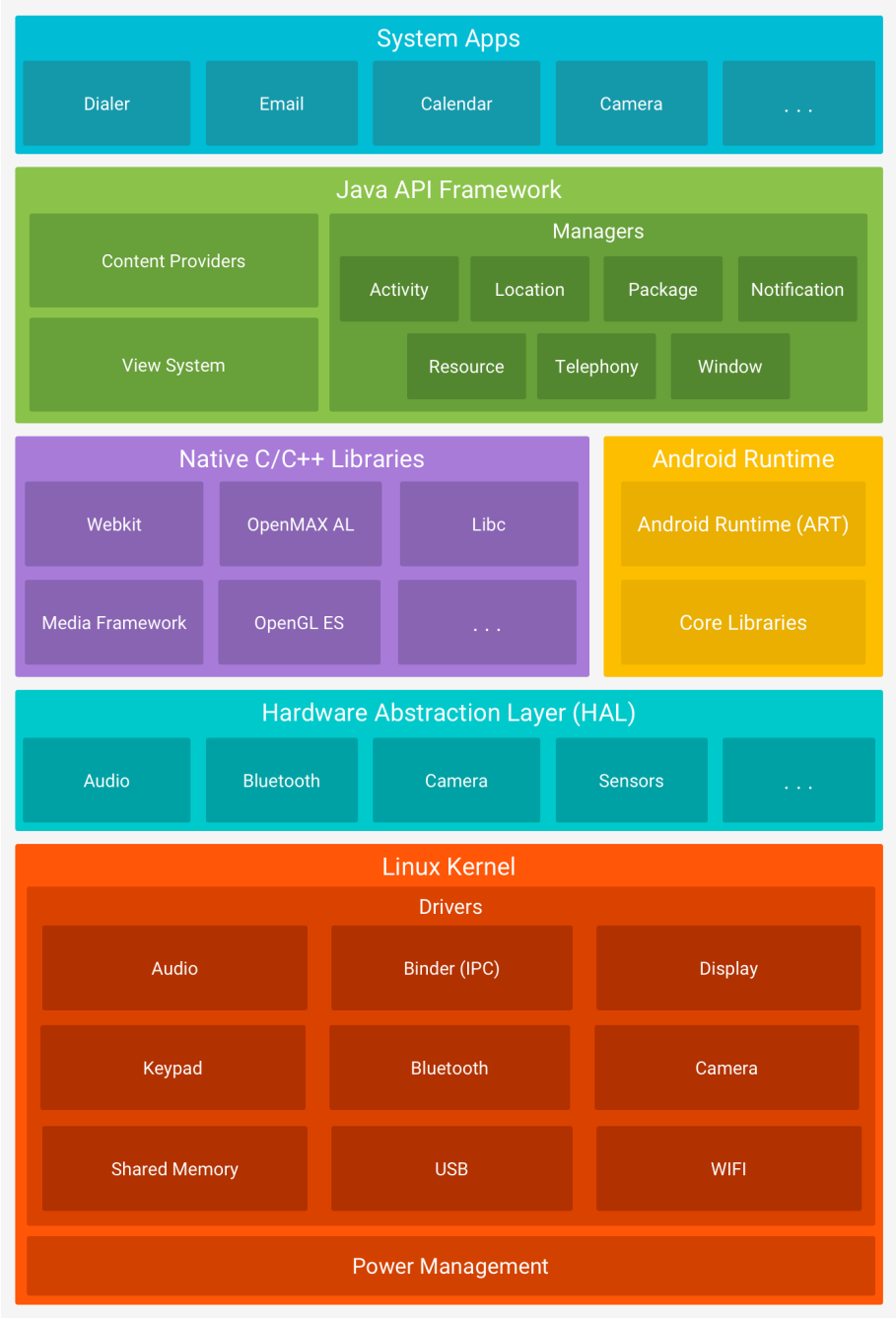


Figura 22: Arquitectura del sistema operativo Android

A la hora de describir bajo mi punto de vista, la arquitectura de Android, lo mejor es empezar desde abajo hacia arriba, como podemos observar en la imagen, la arquitectura de Android se divide en capas, en primer lugar, tenemos la capa del kernel que es la encargada de proporcionar el acceso a los dispositivos físicos del terminal(hardware) a través de drivers, hay que destacar que tal y como podemos observar, Android está basado en el kernel de linux. Además de esto, esta capa es la encarga de la gestión de energía del dispositivo, algo altamente importante, puesto que los terminales móviles tienen una capacidad de batería limitada y es importante el uso y gestión de esta de cara a que sea lo más duradera posible.

Mas arriba la capa Hardware Abstraction Layer(HAL) es la capa encargada de que las capas superiores se puedan comunicar con la capa inferior sin necesidad de modificar estas.

Como podemos observar, más arriba de esta capa nos entramos con librerías nativas de C/C++ y con Android Runtime. Android Runtime es el entorno de ejecución de aplicaciones que utilizan las versiones actuales del sistema operativo Android y que ha sustituido a la máquina virtual Dalvik, a diferencia de esta, Android Runtime maneja muchísimo mejor la CPU del dispositivo, mejorando además duración de la batería de este, al hacer un uso más óptimo de los recursos. A continuación, en la siguiente captura podremos ver con más detalle la diferencias entre la máquina virtual Dalvik y Android Runtime:

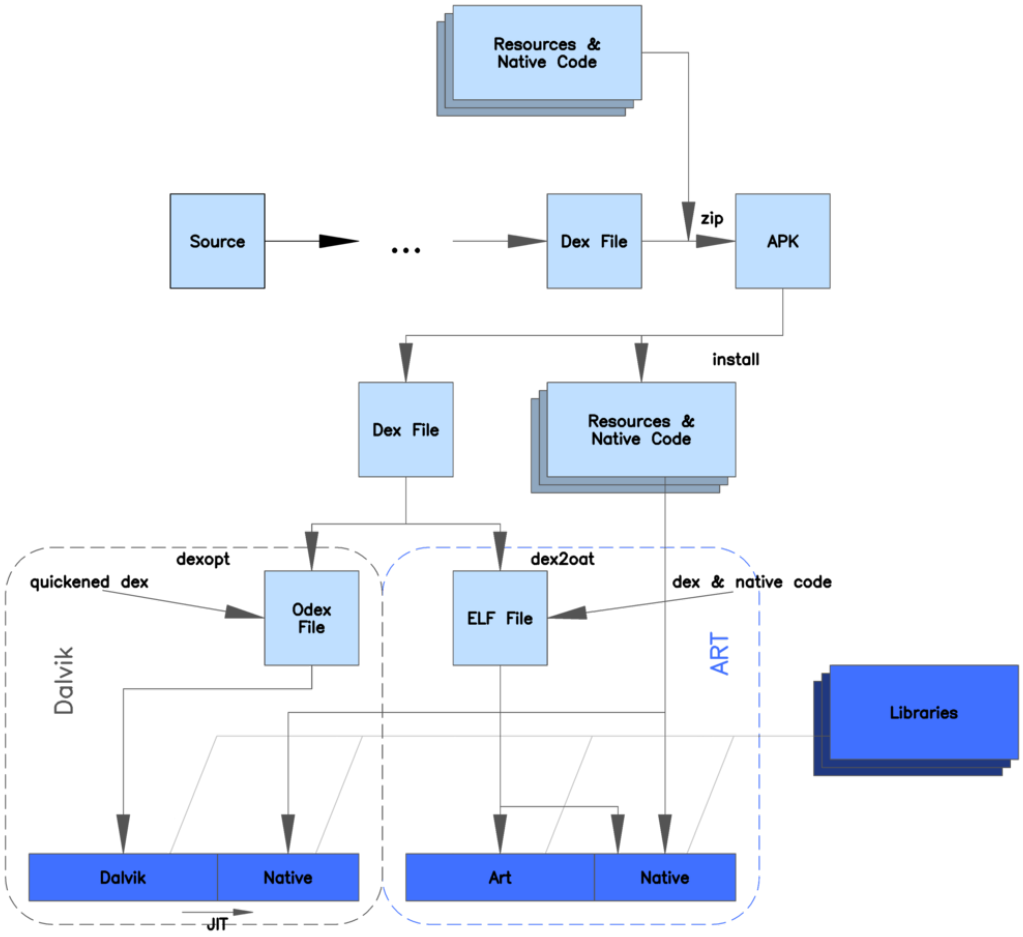


Figura 23: Diferencias entre la máquina virtual de Dalvik y Android Runtime

La siguiente capa es el API de java, que es el lenguaje en él se programa la mayor parte de cualquier aplicación en Android. De esta capa manejaremos las distintas activity de Android así como la interacción con el sistema operativo, como puede ser notificaciones, acceder a la agenda escribir en memoria etc. Finalmente la última capa que nos encontramos es la de aplicaciones que es lo que ve el usuario final , cualquier aplicación como puede ser Email, el calendario etc.

4.3. Entorno de desarrollo Android Studio

En la actualidad, existen principalmente dos entornos de desarrollo para trabajar con Android, por un lado tenemos Eclipse que es un compilador genérico que no está destinado a un lenguaje en concreto, aunque principalmente es usa para aplicaciones java, también se puede desarrollar aplicaciones Android, pero al no estar este entorno optimizado para este sistema operativo en ciertas situaciones puede presentar inestabilidad e incompatibilidades, por lo que este proyecto se va a trabajar con el entorno Android Studio en todo momento, ya que es un entorno que está destinado específicamente a Android y además es el que más compatibilidad y documentación dispone en este sentido, como en nuestro caso utilizaremos Android NDK para implementar el compresor de imagen, vamos a explicar paso a paso, las acciones necesarias para poder trabajar con NDK dentro de nuestro proyecto:

En primer lugar tenemos que tener claro que NDK explicado en secciones anteriores, lo que nos permite principalmente es poder utilizar JNI dentro de nuestro proyecto para ello nos creamos una carpeta llamada jni, para ello tenemos dos opciones , bien creamos la carpeta manualmente dentro de la raíz del proyecto o bien la creamos a partir del propio entorno de desarrollo, la mejor opción es esta última , puesto que Android Studio la reconocerá de forma automática y además nos permitirá ver los archivos de C++ en cualquier modo de visión o estructura del proyecto, para ello realizamos lo siguiente, tal y como podemos observar en la captura:

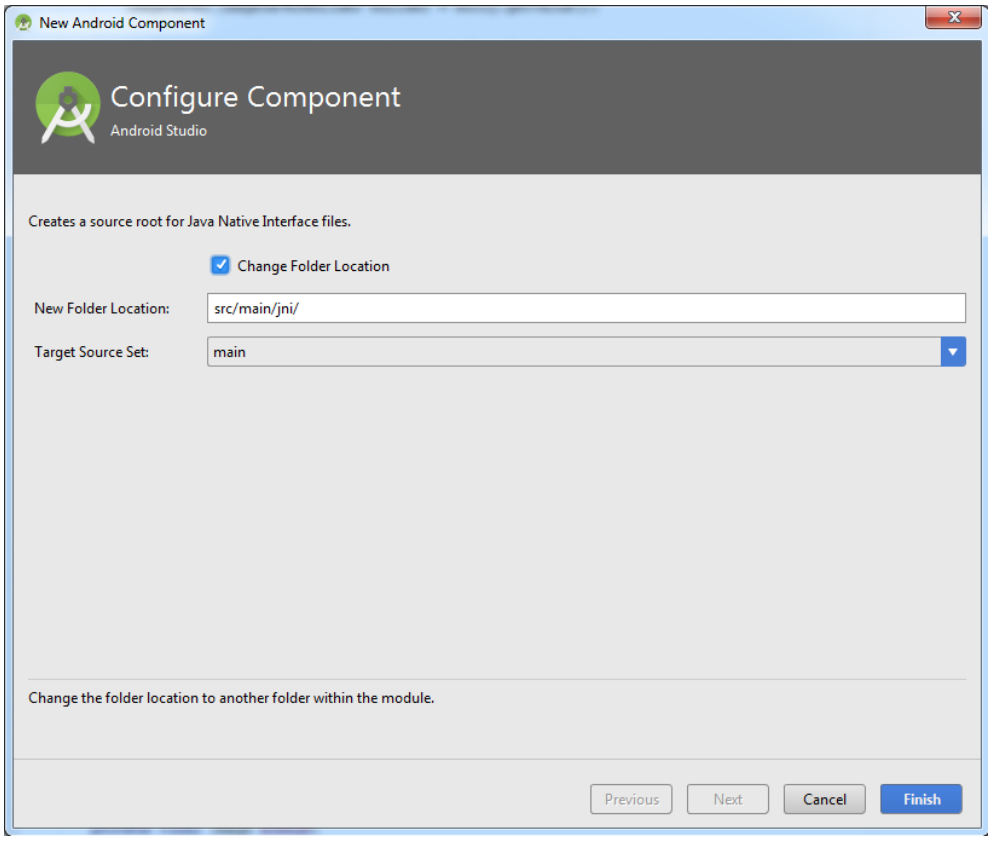


Figura 24: Creación de la carpeta jni en Android Studio

Una vez creada la carpeta jni, tenemos que meter dentro todos los archivos que compongan nuestro proyecto en lenguaje C++ junto a sus librerías en extensión .h, como podemos observar una vez hecho esto y al haber creado la carpeta desde el mismo entorno de desarrollo se nos muestran de forma ordenada todos los ficheros C++, metidos dentro de la carpeta cpp , que es creada automáticamente por el entorno, tal y como podemos observar:

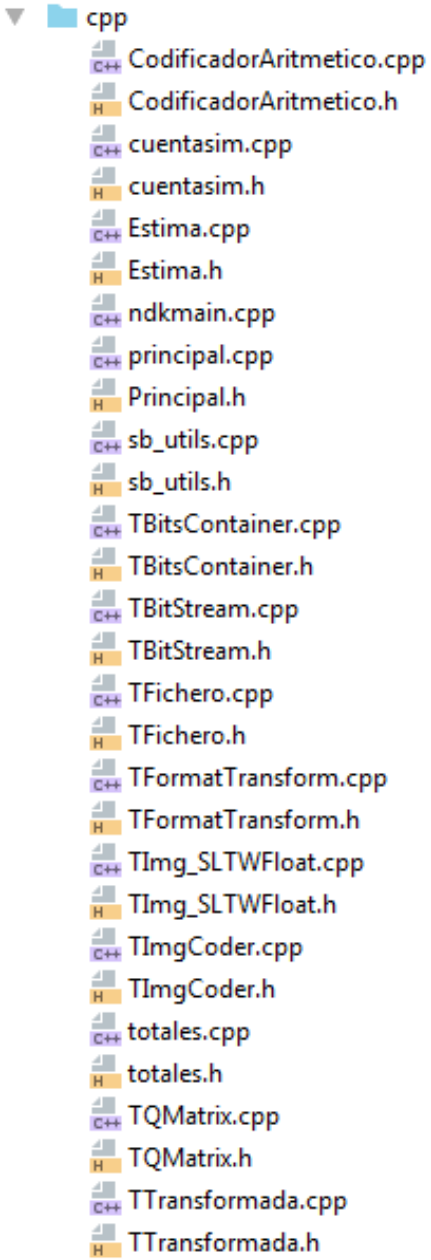


Figura 25: Carpeta cpp que nos muestra todos los ficheros escritos en lenguaje C++ del proyecto

Una vez hecho esto, tenemos dos opciones o bien compilamos nuestro directamente desde la secuencia de comandos ndk-build o bien lo hacemos integrando esta secuencia de comandos desde el mismo Android Studio, aunque lo más como es lo segundo, vamos a explicar los dos métodos:

4.3.1. Compilación del código C++ con ndk-build

Para realizar la compilación con este método, lo primero que tenemos que hacer es descargamos el NDK de la pagina oficial de Android developers:

Descargas de NDK

Plataforma	Paquete	Tamaño (bytes)	Suma de comprobación de SHA1
Windows 32-bit	android-ndk-r15b-windows-x86.zip	783838327	74e45891d0cc99b077b3951aeba87d9c91df20a8
Windows 64-bit	android-ndk-r15b-windows-x86_64.zip	848796389	126a1cd8985132c0383ab96579feed09ba402e22
Mac OS X	android-ndk-r15b-darwin-x86_64.zip	959321525	05e3eec7e9ce1d09bb5401b41cf778a2ec19c819
Linux 64-bit (x86)	android-ndk-r15b-linux-x86_64.zip	974035125	2690d416e54f88f7fa52d0dcb5f539056a357b3b

Figura 26: Descarga de Android NDK de la pagina oficial de Android developers

Una vez descargado, añadiremos el directorio de ndk-build a las variables de entorno del sistema operativo, en nuestro caso windows:

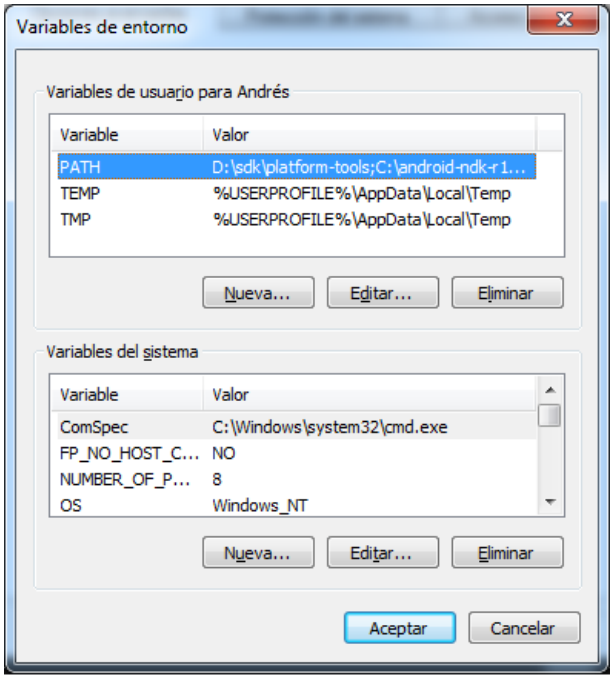


Figura 27: Agregación de ndk-build a la variables de entorno de windows

Finalmente ,nos dirigimos al directorio de nuestro en el que tenemos la carpeta jni y ejecutamos ndk-build:

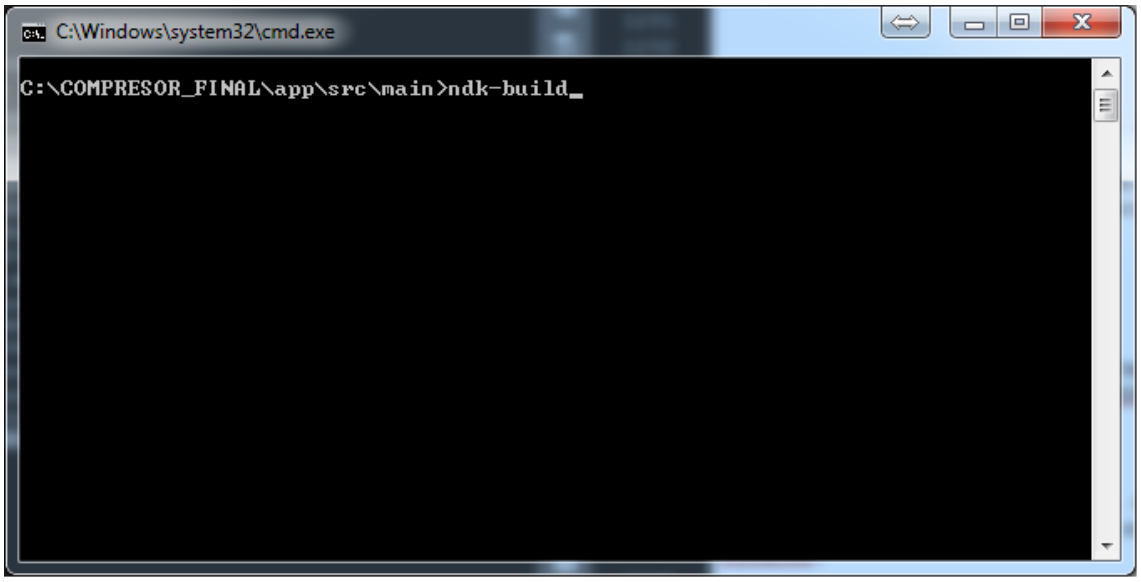


Figura 28: Ejecución de ndk-build en el directorio donde se encuentra la carpeta jni del proyecto Android

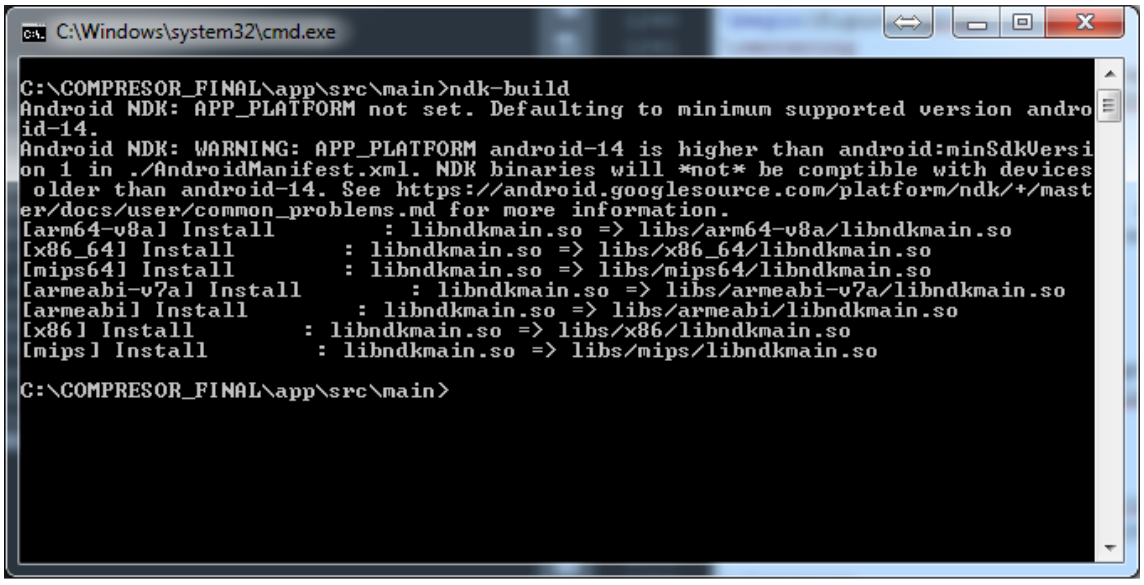


Figura 29: Generación de la librerías del compresor en código nativo tras finalizar la compilación

4.3.2. Compilación del código C++ con Android Studio

Para realizar la compilación con este método, lo primero que debemos hacer es instalar en Android Studio las siguientes herramientas:

- NDK
- LLDB

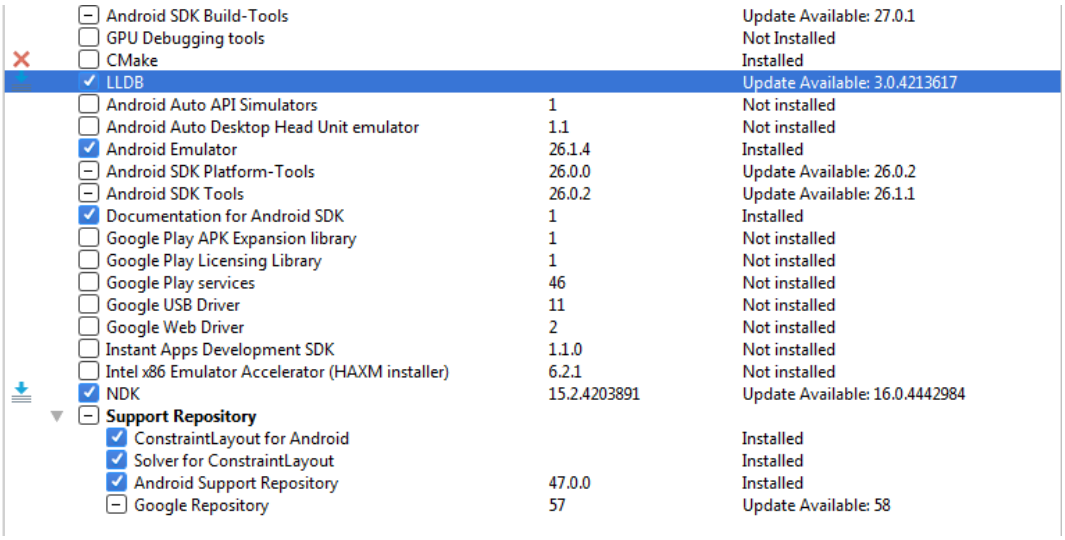


Figura 30: Instalación de NDK y LLDB a través de sdk manager

Estas herramientas la instalamos a través del sdk manager que viene incluido en el entorno de desarrollo tal y como podemos observar en la siguiente captura: Una vez instaladas estas dos herramientas, debemos indicarle al entorno de desarrollo que, en nuestro proyecto, habrá código C/C++ que deberá ser compilado, para ellos hacemos click derecho con el ratón debajo en la carpeta app de nuestro proyecto y seleccionamos la siguiente opción, tal y como podemos observar en la captura:

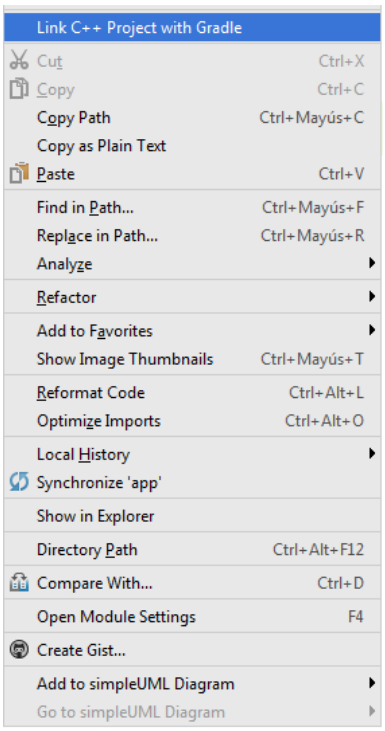


Figura 31: Adición de soporte para C++ en Android Studio

Una vez seleccionada esta opción, nos aparecerá la siguiente ventana emergente:

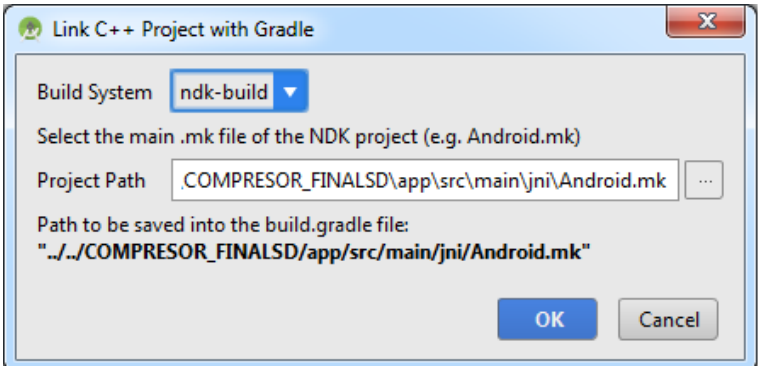


Figura 32: Conexión del proyecto C++ con el gradle de Android

En Build System seleccionamos ndk-build y en la ruta del proyecto seleccionamos el fichero Android.mk dentro de la carpeta jni de nuestro proyecto Android. Una vez hecho esto ya tendremos nuestro proyecto preparado para compilar el código nativo a la vez que el código javo de Android. Hemos de tener en cuenta que con este método el proyecto tardará mucho más en compilar y ejecutarse puesto que antes de compilar el código se compilará el código C++, además cada cambio que hagamos en el código nativo tendrá como resultado una recompilación completa de todo el código C/C++.

Una vez expuestos los dos métodos que tenemos disponibles para compilar código nativo con NDK , hemos de destacar que si bien la integración de NDK en Android Studio es más intuitiva y sencilla a priori, para proyectos complejos esta opción tiene poco sentido , fuera de la comodidad que nos proporciona a la hora de compilar todo el proyecto, puesto que no dispondremos de información de errores en tiempo de ejecución y los errores mostrados en tiempo de compilación no son claros y concisos , muchas veces ni siquiera se indica donde esta exactamente el error, aunque ciertamente existe el depurador LLDB para estos casos , no funciona bien en proyectos C++ complejos, un problema que seguramente se resolverá en futuras versiones de Android Studio. Para detectar errores en tiempo de compilación lo mejor es compilar el código directamente con ndk-build, además, tenemos multitud de comandos para poder detectar errores de todo tipo y además la misma herramienta nos indicara de forma clara donde se encuentran exactamente los errores sintácticos, incluso en algunos casos nos sugerirá el cambio necesario para solucionar el error. Para los errores en tiempo de ejecución se explicará en secciones posteriores el método usado para poder detectar dichos errores y corregirlos.

5. Diseño, análisis e implementación de la aplicación

5.1. Introducción

En esta sección vamos a detallar el desarrollo de la aplicación, desde la construcción de la parte Android hasta la implementación del algoritmo PETW escrito en C++, incluyendo la comunicación entre estos dos lenguajes a través de NDK.

5.2. Objetivos

El principal objetivo de la aplicación es tomar fotografías en formato RAW de forma que estas se compriman automáticamente con los parámetros de compresión introducidos por el usuario.Para ello accederemos al hardware de la cámara del dispositivo tomando la fotografía en formato YUV con enfoque automático y adaptando siempre la resolución a las dimensiones del dispositivo usado, del cual separaremos sus tres componentes Y, U y V de forma que aunque se almacenaran estas dos componentes para futuras implementaciones del algoritmo a color, en esta implementación solo nos quedaremos con la componente monocromática Y, de forma que esta componente en forma de array será pasada directamente a memoria con el fin de ahorrar tiempo de procesamiento para el algoritmo de compresión que posteriormente procesara la imagen, generando el fichero de compresión cuyo nombre estará seguido de la extensión petw.

5.3. Implementación del codificador PETW mediante NDK

En un principio se cuenta con la versión del algoritmo PETW escrita en lenguaje visual C++ y desarrollada en el entorno de desarrollo Visual Studio.Partiendo de esta versión como base deberemos en primer lugar depurar todo el código corrigiendo todos los errores sintácticos provenientes de pasar de Visual C++ a C++ de los compiladores como gcc que son los usados por NDK. Una vez corregidos todos los errores sintácticos, procederemos a solucionar los errores en tiempo de ejecución, los más difíciles de solucionar sin duda, puesto que como hemos descrito en secciones anteriores, aunque Android Studio cuenta con LLDB como depurador, este no funciona bien en proyectos complejos, así que depuraremos el algoritmo mediante una librería basada a su vez en la librería *android/log.h*, tal y como hemos visto en secciones anteriores. Además, también deberemos agregar al proyecto C++ los ficheros Android y Application .mk necesarios para la compilación del proyecto en NDK. Una vez hecho esto, creamos una clave java en Android que será la responsable de llamar al método nativo de compresión , pasándole los argumentos necesarios que son, el array de enteros que contiene la componente monocromática Y de la imagen, los argumentos que contienen los parámetros de compresión definidos por el usuario , así como las dimensiones finales de la imagen.

5.4. Análisis de la aplicación

En esta sección vamos detallar las funciones principales de la aplicación con el objetivo de comprender mejor su funcionamiento:

- Tomar imágenes en formato YUV(RAW) de la cámara del dispositivo:

  La aplicación constará de una interfaz gráfica, que le permitirá al usuario, tomar fotografías en formato YUV(RAW). Dicha foto sera tomada haciendo uso del paquete camera2raw de forma que solo se recogerá la componente monocromática Y.

- Editar los parámetros de compresión por parte del usuario:

  La aplicación dispondrá de un dialogo que le permitirá al usuario introducir los parámetros de compresión que considere oportunos.

- Comprimir una imagen a partir de un fichero:

  La aplicación dispondrá de un apartado que le permitirá al usuario comprimir una imagen a partir de un fichero, en caso de que esta no haya sido tomada del mismo dispositivo o quiera comprimir una imagen ya almacenada en memoria.

- Enfoque automático

  La aplicación enfocara de forma automática y se ajustara a la resolución de pantalla de cada dispositivo.

5.5. Diseño de la aplicación Android

En esta sección, explicaremos el funcionamiento y manejo de la aplicación Android. En primer lugar, lo primero que vemos al analizar la aplicación es lo siguiente:

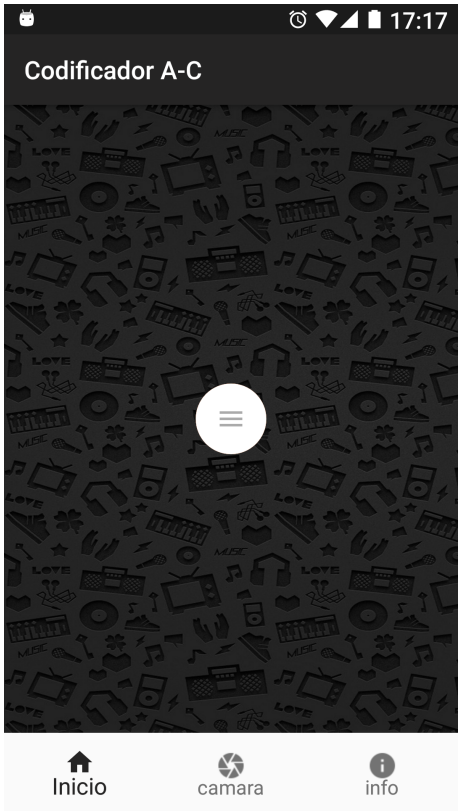


Figura 33: Pantalla principal de la aplicación Android

Como podemos observar en la figura 34, nuestro menú principal, consta de una barra de navegación inferior con tres botones:

- Inicio
- Cámara
- Info

Siguiendo el orden de derecha a izquierda, en primer lugar, nos encontramos con el botón info, que una vez presionado nos mostrara un mensaje informativo con los autores de la aplicación, por un lado, Andrés Ruiz García, autor de esta memoria y por otro, Christian Moreno Navarro que se ha encargado de desarrollar el descompresor de imágenes de este mismo algoritmo, tal y como podemos observar en la siguiente figura:

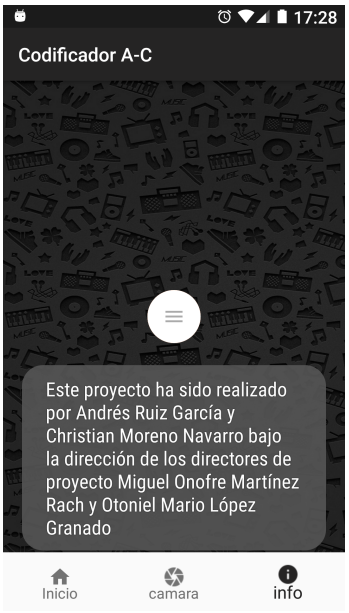


Figura 34: Resultado de presionar el botón info

En segundo lugar, el botón cámara, nos permitirá tomar fotografías en formato YUV(RAW), tomando únicamente la componente monocromática y siendo recortada(crop) de forma automática a la resolución correcta que requiere el algoritmo para poder ser procesada, nos encontraremos con dos botones, tal y como podemos observar en la siguiente figura:

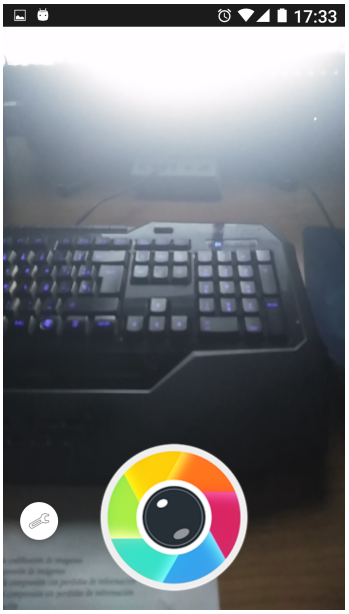


Figura 35: Resultado de presionar el botón cámara

El botón izquierdo nos servirá para desplegar el menú que le permitirá al usuario escoger los parámetros de compresión que desee, tal y como podemos observar en la siguiente figura:

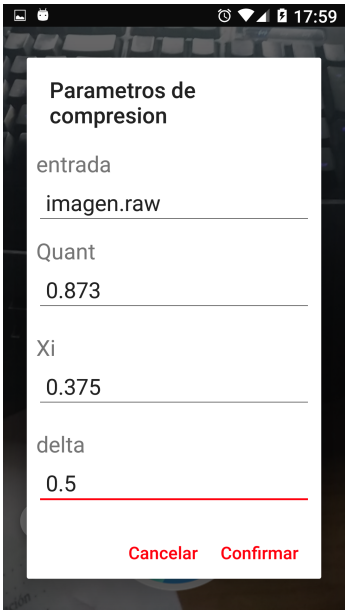


Figura 36: Resultado de presionar el botón del menú de configuración

Una vez que el usuario haya introducido los parámetros, pulsará sobre el botón confirmar y posteriormente, pulsará sobre el botón central para tomar la fotografía que será comprimida perceptualmente de forma automática con dichos parámetros. Una vez tomada la fotografía, nos aparecerá un mensaje en pantalla indicándonos que la fotografía se ha realizado con éxito, como podemos observar en la siguiente figura:

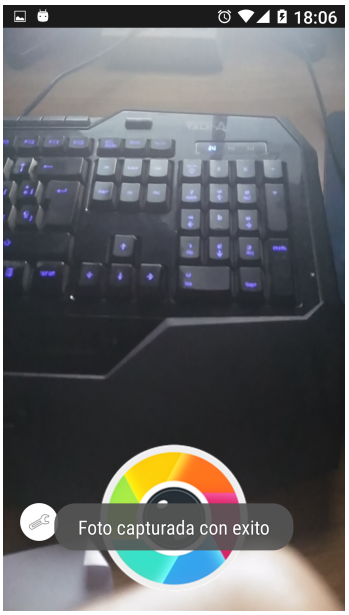


Figura 37: Mensaje de confirmación de fotografía tomada con éxito

Una vez tomada la fotografía se generarán los siguientes ficheros, como podemos observar en la siguiente figura:

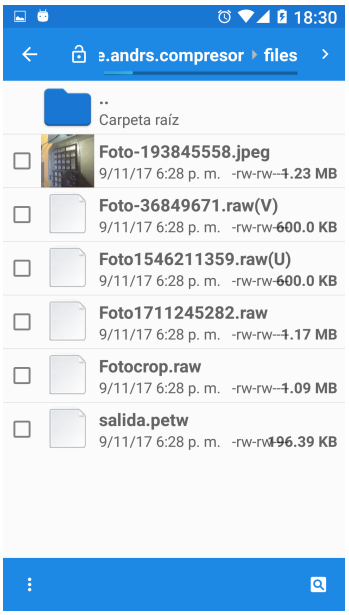


Figura 38: Ficheros generados por la aplicación

Por un lado y con fines exclusivamente de depuración de la aplicación, tenemos la tres componentes de la imagen en formato YUV, separadas (las componentes U y V, en este caso no serán utilizadas puesto que esta versión del compresor solo utilizara la componente monocromática Y), Fotocrop, que es la imagen perteneciente a la componente monocromática recortada a la resolución valida del algoritmo de compresión, la imagen en formato jpg y finalmente el fichero que contiene la imagen ya comprimida perceptualmente(salida.petw).

Volviendo a la pantalla principal de la aplicación, si pulsamos sobre el botón de inicio, podremos pulsar sobre el botón central, desplegándose un menú a su alrededor, tal y como podemos observar en la siguiente figura:
Los tres botones de arriba son los únicos funcionales, el resto se dejan como propuesta para futuras implementaciones de esta aplicación con el objetivo de añadir más funciones. A continuación, se describirá la función cada uno:
Por un lado tenemos un visor para imágenes en formato jpeg y dng, como podemos observar en la siguiente figura:
Una vez que hemos pulsado este botón, podremos elegir la imagen que queramos de nuestra galería, la cual será mostrada:
Por otro lado, tenemos el botón de compresión, como podemos observar en la siguiente figura:

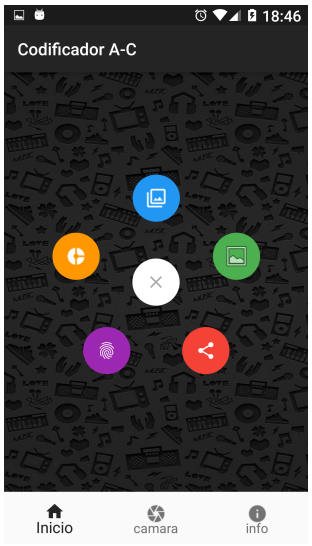


Figura 39: Menu principal de la aplicación

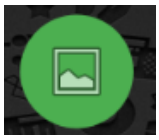


Figura 40: Icono del visor de imágenes

Figura 41: Imagen mostrada a través del visor

Figura 42: Icono del modo compresión

Una vez que pulsamos este botón, se nos abrirá un dialogo para seleccionar los parámetros de compresión y el fichero imagen que queremos comprimir, como podemos observar en la siguiente figura:

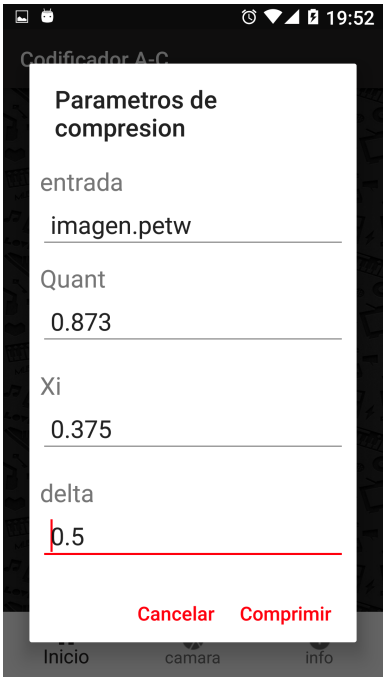


Figura 43: Menu de compresión

Una vez que el usuario ha escogido los parámetros de compresión y el nombre del fichero imagen que quiere comprimir, dicho fichero imagen será comprimido, generando el fichero de extensión. petw correspondiente. Finalmente tenemos el botón de descompresión tal y como podemos observar en la siguiente figura:

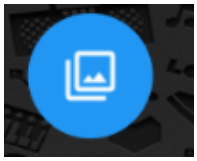


Figura 44: Icono del modo descompresión

Una vez que pulsamos este botón, se nos abrirá un dialogo para seleccionar los parámetros de descompresión y el fichero imagen que queremos descomprimir de extensión. petw, como podemos observar en la siguiente figura:

Figura 45: Menu de descompresión

5.6. Implementación de la librería *"Camera2"* para la realización de las fotografías en formato RAW

El paquete android.hardware.camera2[**13**] nos proporciona una interfaz desde la cual, el usuario puede tomar fotografías en formato RAW.

Dicho paquete, podemos usarlo a partir de la API 21 de Android, sustituyendo este al antiguo paquete *camera*. Camera2 funciona de la siguiente forma:

- **CameraManager:** Lo usamos para tener todas las cámaras del dispositivo disponible, como la cámara frontal y la trasera y sus propiedades se obtienen de la case CameraCharacteristics.
- **CameraDevice:** Podemos acceder un sensor de cámara de la clase CameraManager a partir de su número de identificación(ID) y realizar las operaciones relacionadas.
- **CaptureRequest:** Crea un CaptureRequest desde el dispositivo de cámara para capturar imágenes.
- **CameraCaptureSession:** Crea un CameraCaptureSession para obtener CaptureRquest desde CameraDevice.
- **CameraCaptureSession.CaptureCallback:** Proporcionará todos los resultados de la sesión de captura.

En la siguiente figura, podemos ver de forma gráfica, el funcionamiento de la librería Camera2:

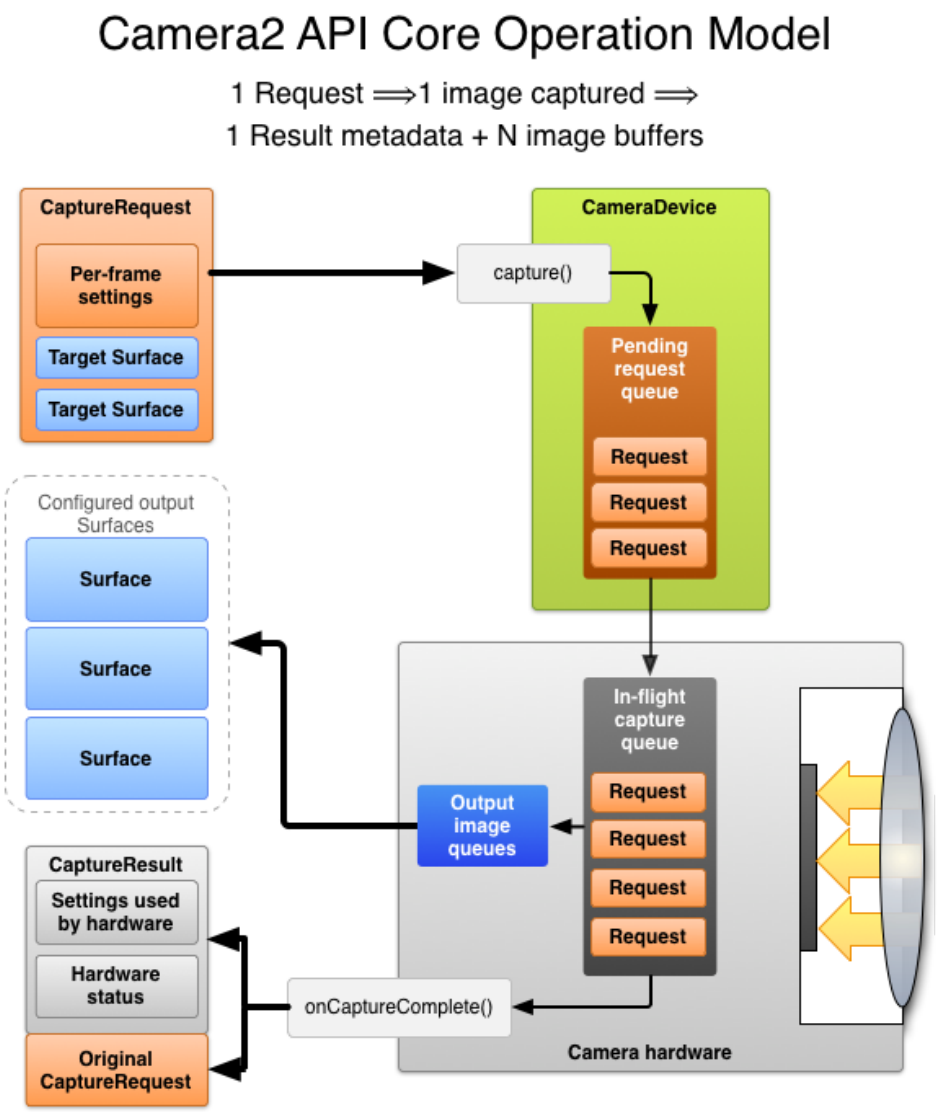


Figura 46: Funcionamiento de la librería Camera2

5.7. Diagramas UML de la aplicación

A continuación, en las siguientes figuras se mostrarán las distintas partes del código más importantes, a través de diagramas UML con el fin de comprender mejor su funcionamiento, estos diagramas se dividirán en dos grupos, por un lado, el diagrama perteneciente al código Java y por otro, el código C++ del algoritmo PETW.

5.7.1. Parte Java

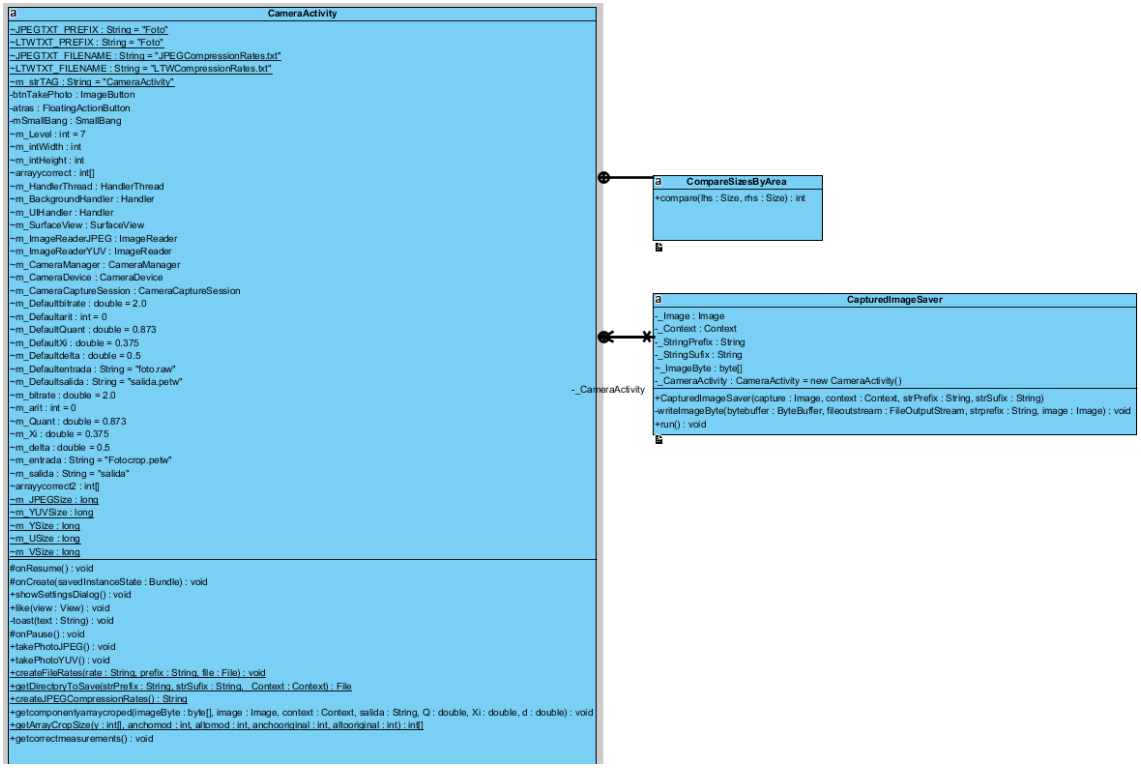


Figura 47: Diagrama UML de la clase CameraActivity

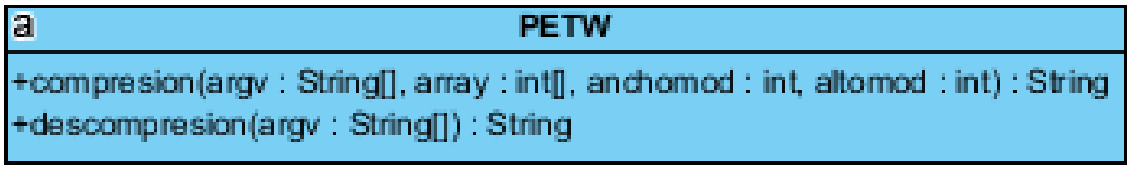


Figura 48: Diagrama UML de la clase PETW

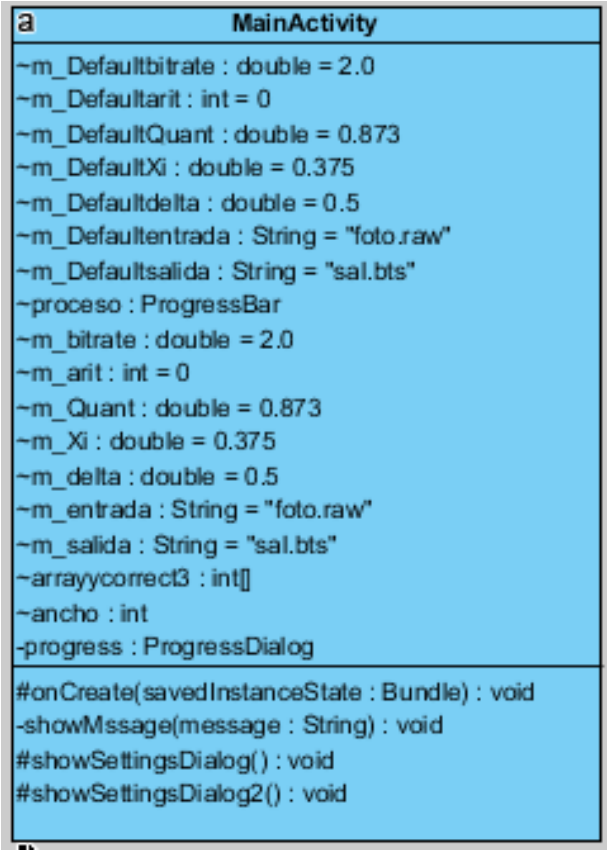


Figura 49: Diagrama UML de la clase MainActivity

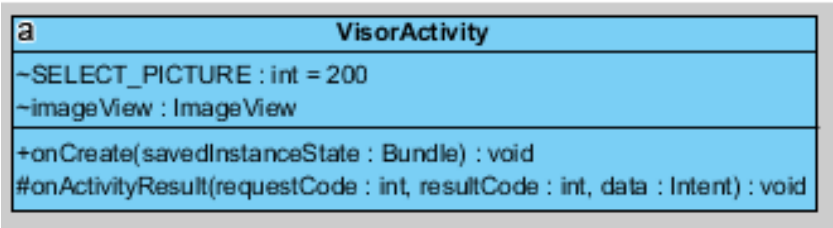


Figura 50: Diagrama UML de la clase VisorActivity

5.7.2. Parte C++

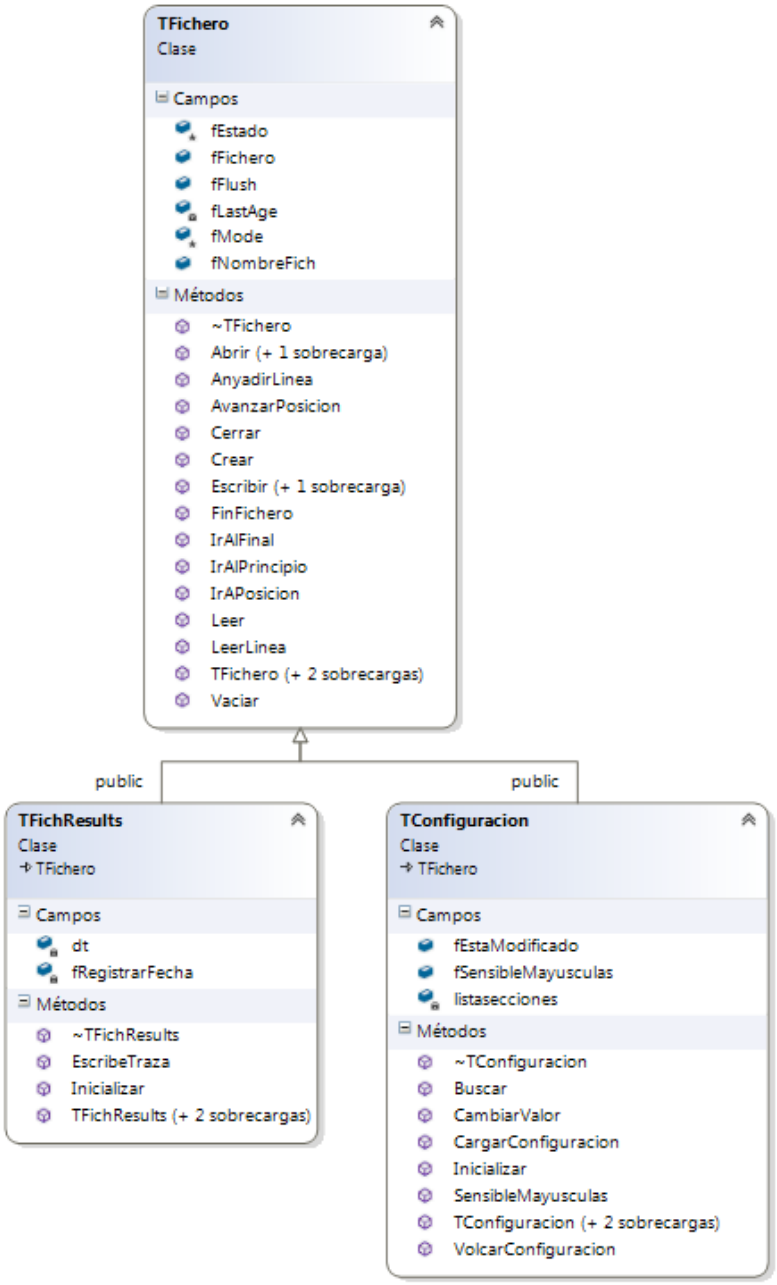


Figura 51: Diagrama UML de las clases TFichero, TFichResults y Tconfiguración

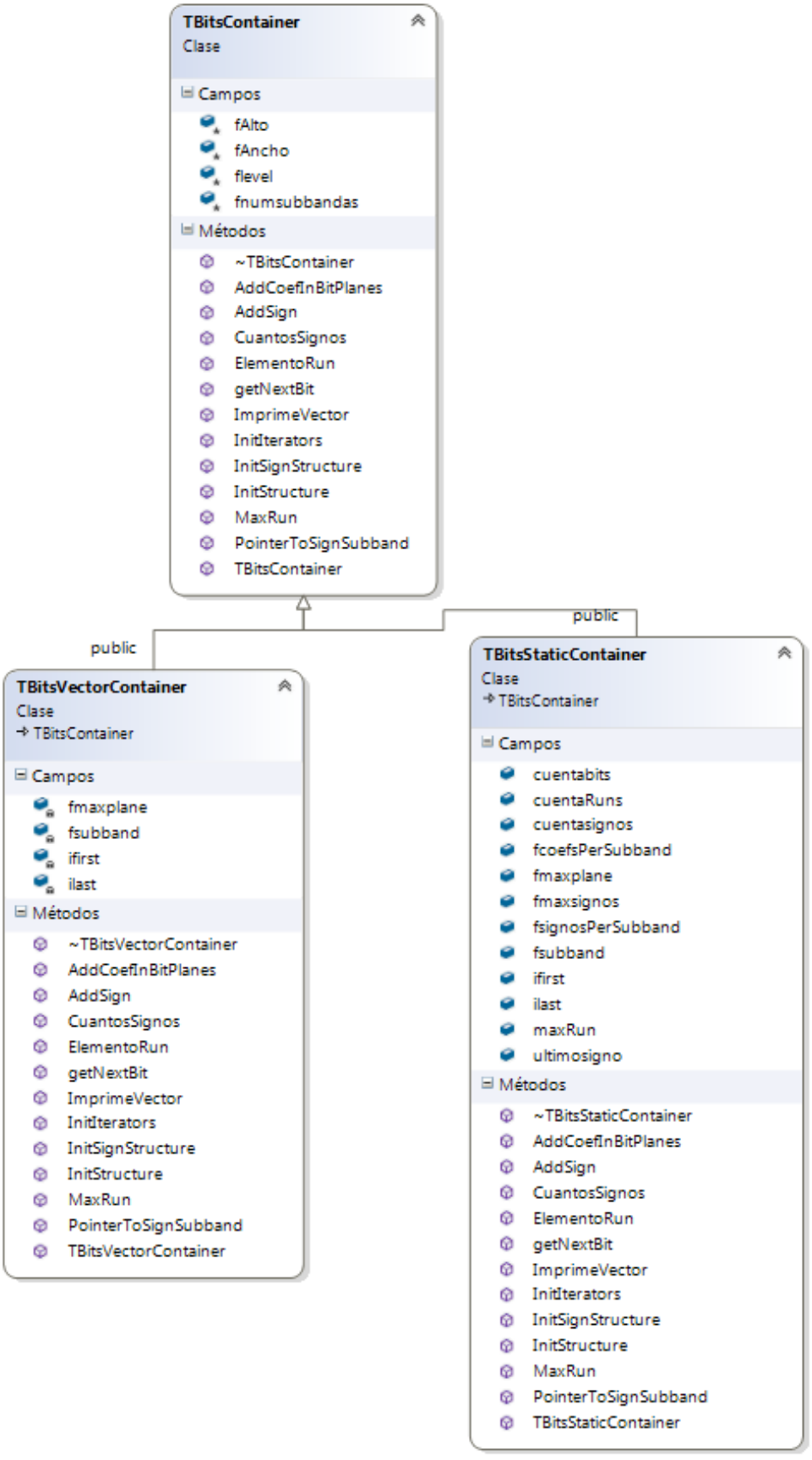


Figura 52: Diagrama UML de las clases TBitsContainer, TBitsVectorContainer y TBitsStaticContainer

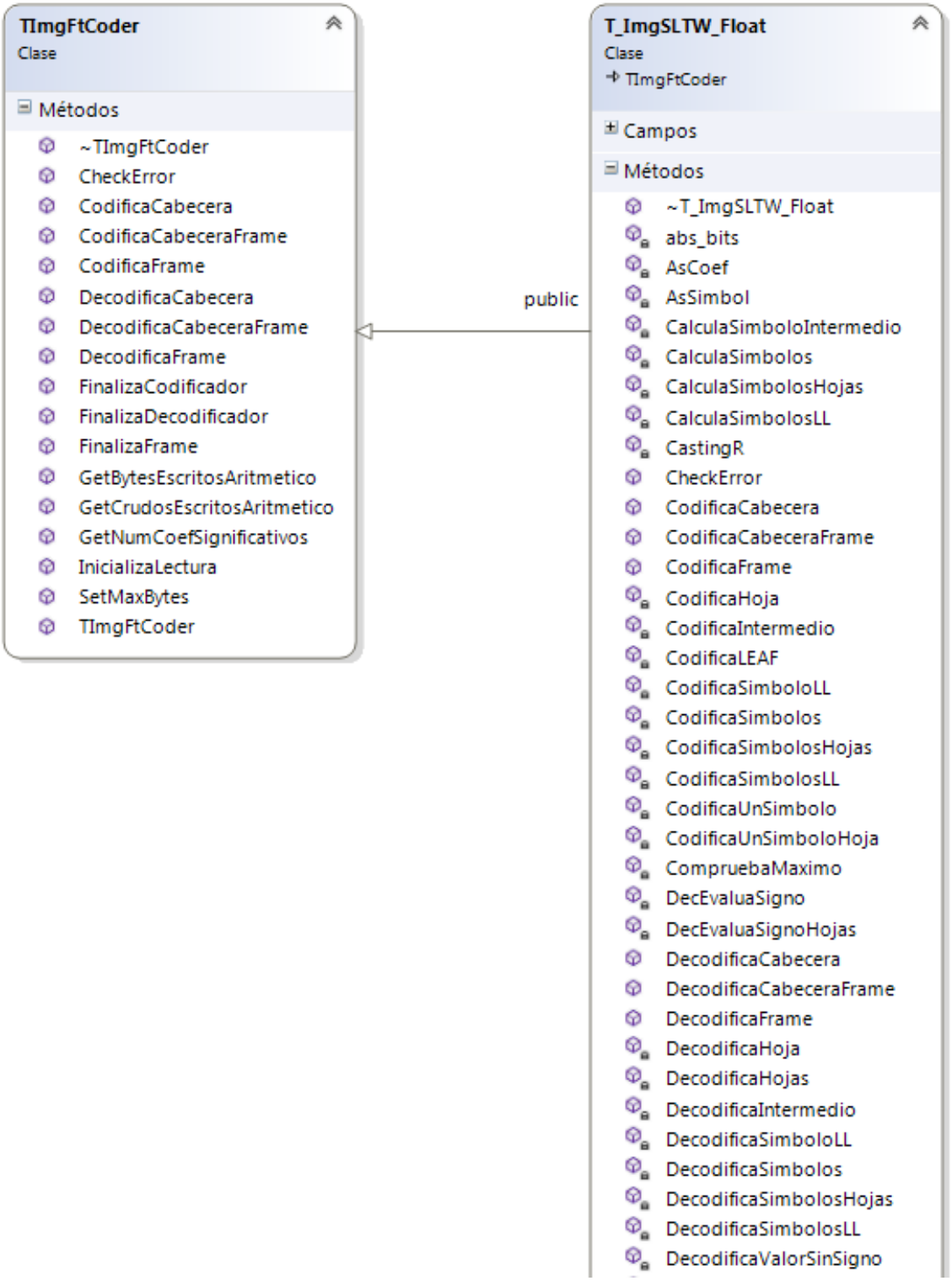


Figura 53: Diagrama UML de las clases TImgFtCoder y TImgSLTWFloat

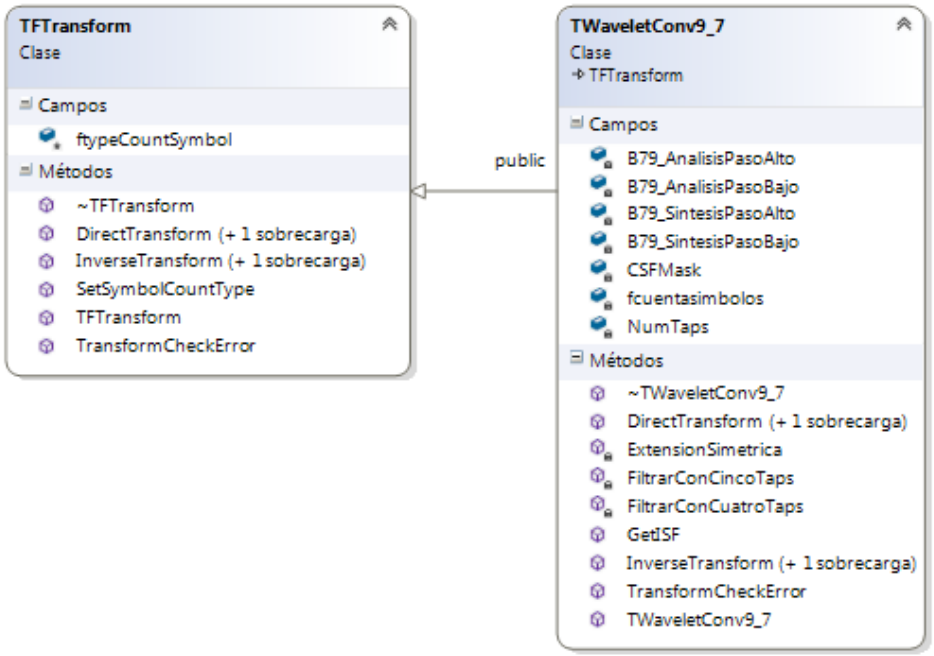


Figura 54: Diagrama UML de las clases TFTransform y TWaveletConv97

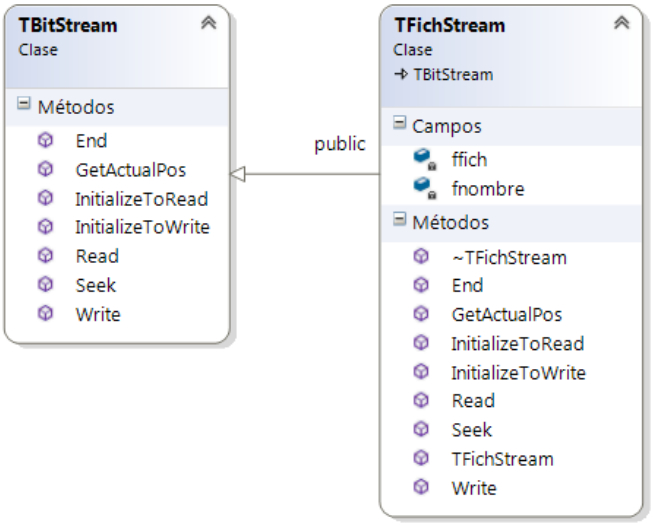


Figura 55: Diagrama UML de las clases TBitStream y TFichStream

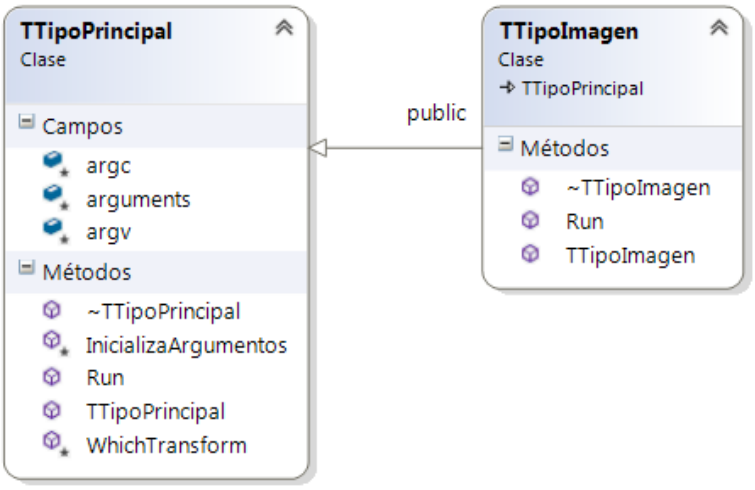


Figura 56: Diagrama UML de las clases TTipoPrincipal y TTipoImagen

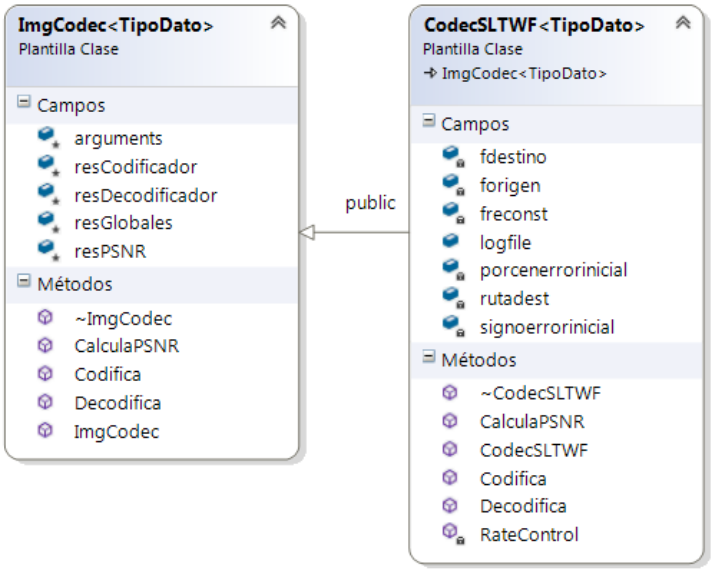


Figura 57: Diagrama UML de las clases ImgCodec y CodecSLTWF

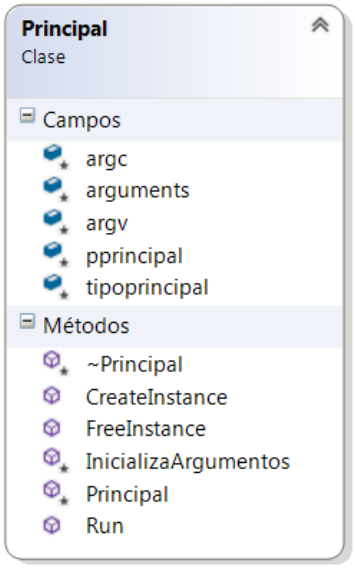


Figura 58: Diagrama UML de la clase Principal

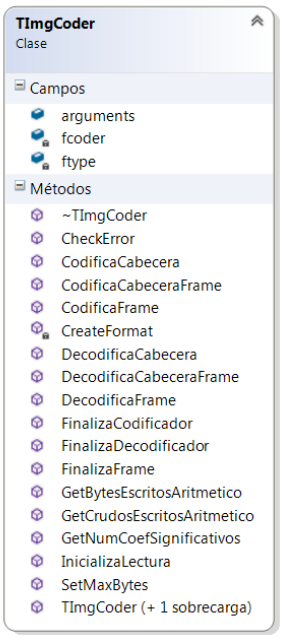


Figura 59: Diagrama UML de la clase Principal

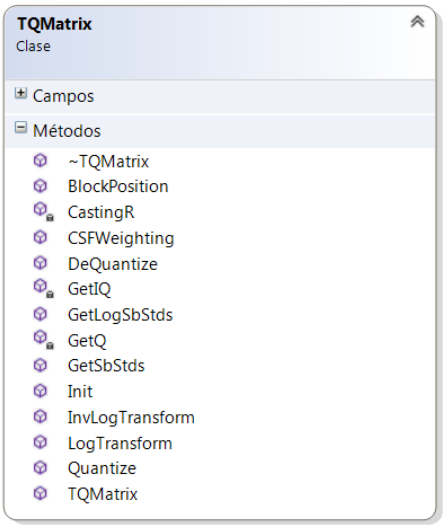


Figura 60: Diagrama UML de la clase TQMatrix

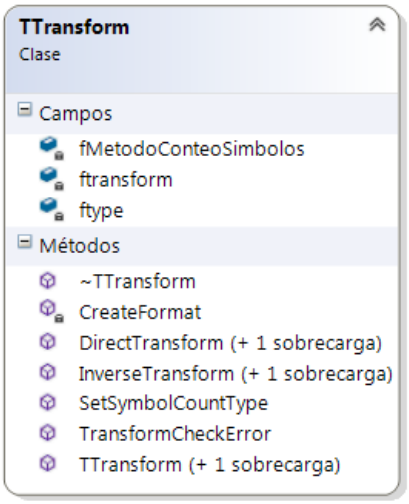


Figura 61: Diagrama UML de la clase TTransform

6. Resultados

6.1. Introducción

En esta sección, vamos comprobar el rendimiento del algoritmo PETW frente JPEG2000, así como un test de tiempos de escritura tanto en memoria de almacenamiento como en memoria RAM, una vez hechas las pruebas, se realizará un análisis de los resultados.

6.2. Test de la aplicación

Para las pruebas se realizan cuatro fotos que se comprimirán a distintos valores de Q(Tasa(bpp)), con y sin elevación perceptual(CSF). De forma que podremos determinar con que método se obtiene un mejor resultado. Las fotografías son las siguientes:

Figura 62: Fotografía silla

Figura 63: Fotografía Climatizador

Figura 64: Fotografía Papelera

Figura 65: Fotografía Baúl

Estas fotografías serán comprimidas a distintos valores de los parámetros Q, con y sin CSF y a continuación, se mostrarán las gráficas correspondientes para poder observar los resultados y obtener las conclusiones:
Posteriormente realizamos las pruebas con VIF para comprobar los resultados:

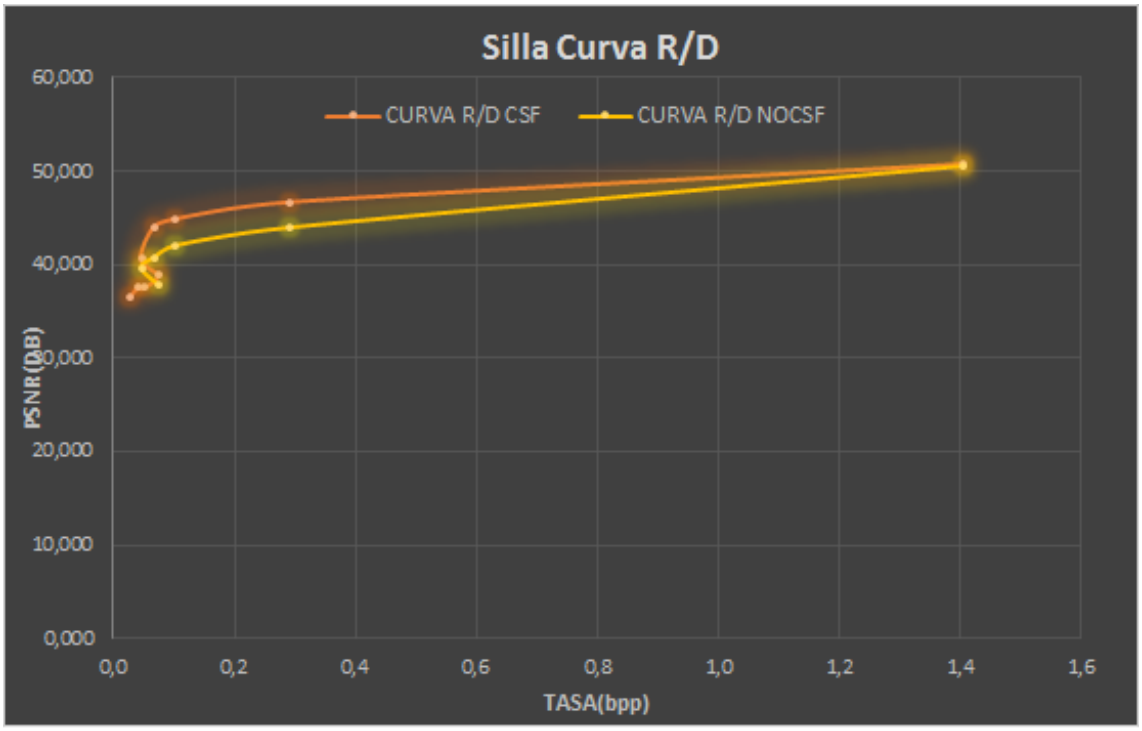


Figura 66: Resultados de la fotografía silla

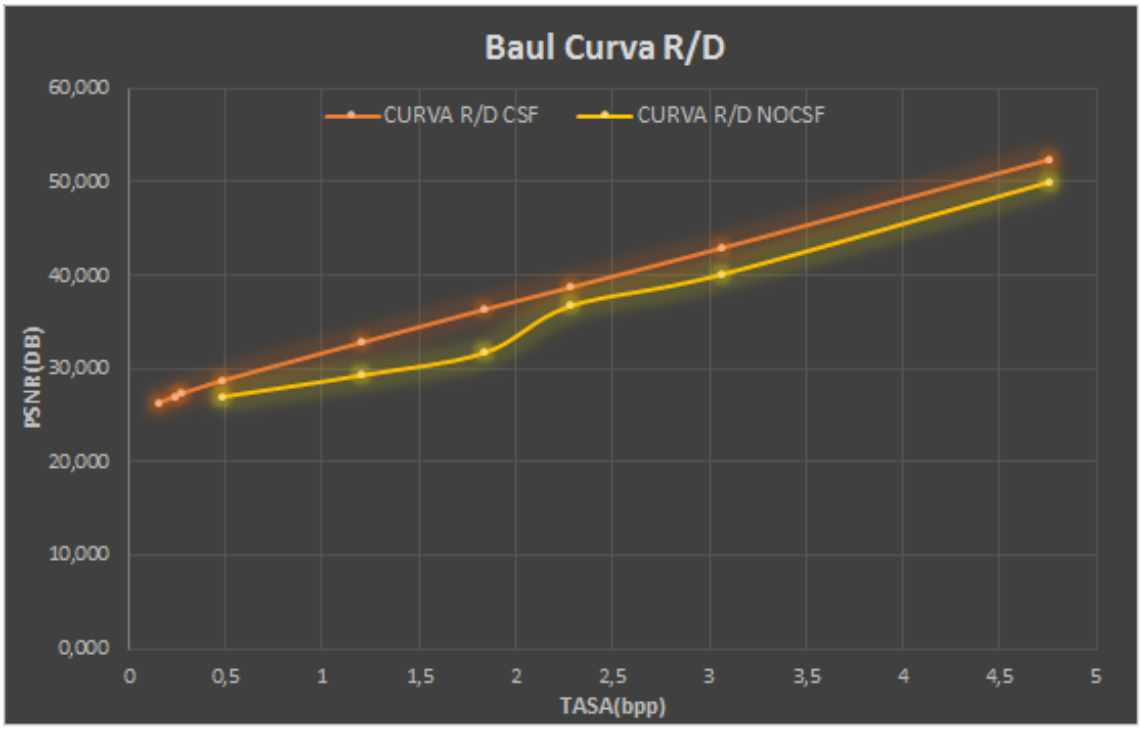


Figura 67: Resultados de la fotografía baúl

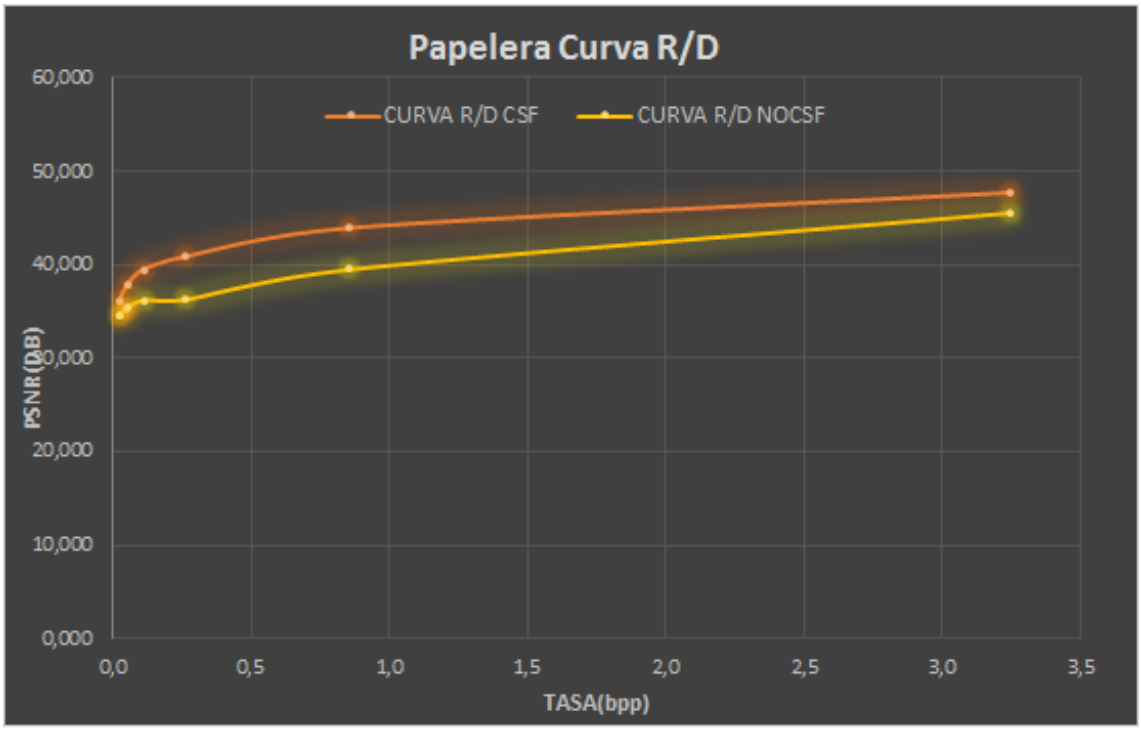


Figura 68: Resultados de la fotografía papelera

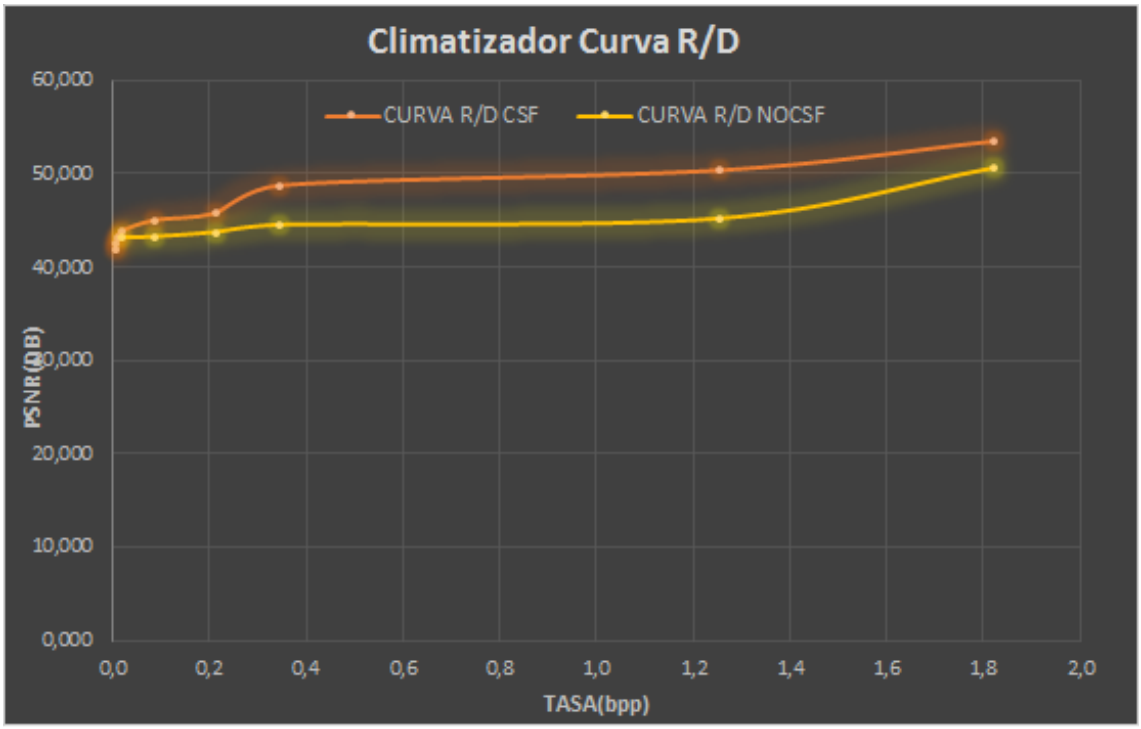


Figura 69: Resultados de la fotografía climatizador

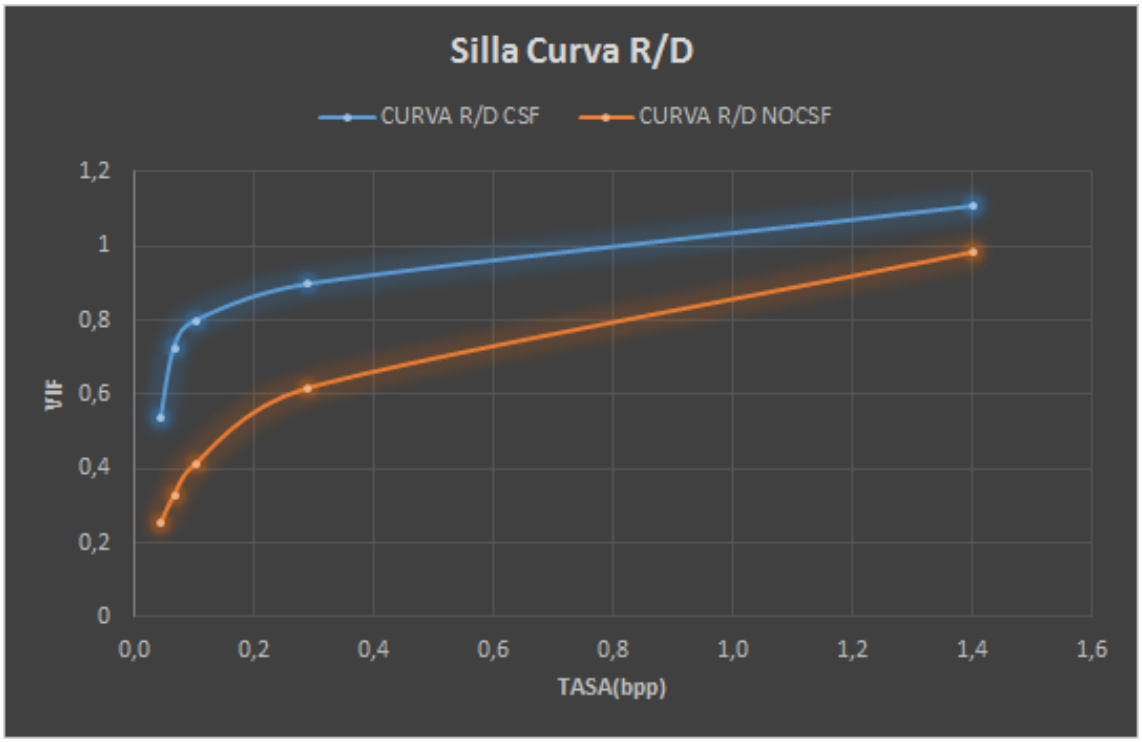


Figura 70: Resultados de la fotografía silla

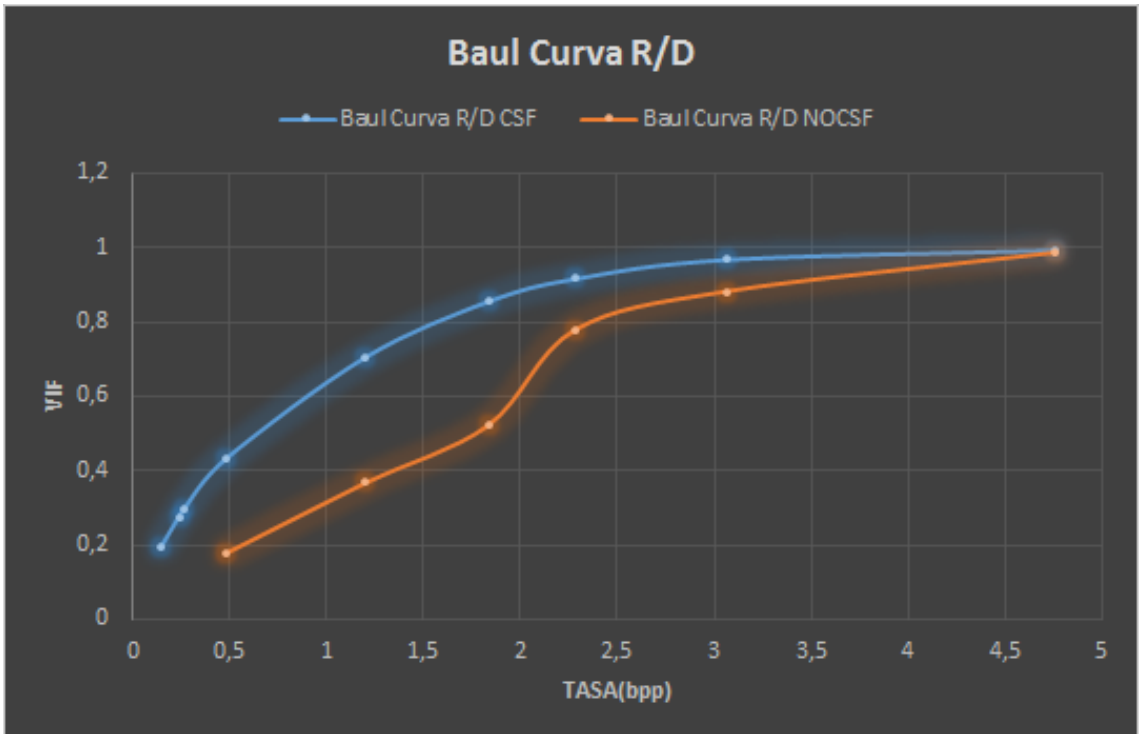


Figura 71: Resultados de la fotografía baúl

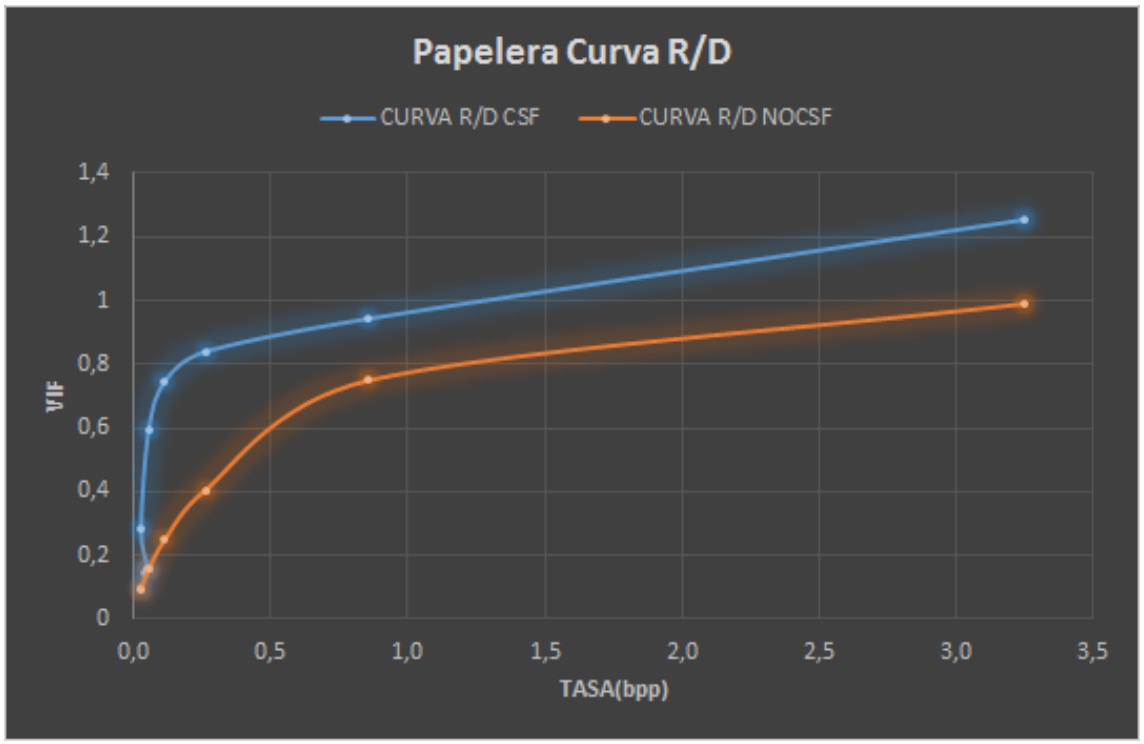


Figura 72: Resultados de la fotografía papelera

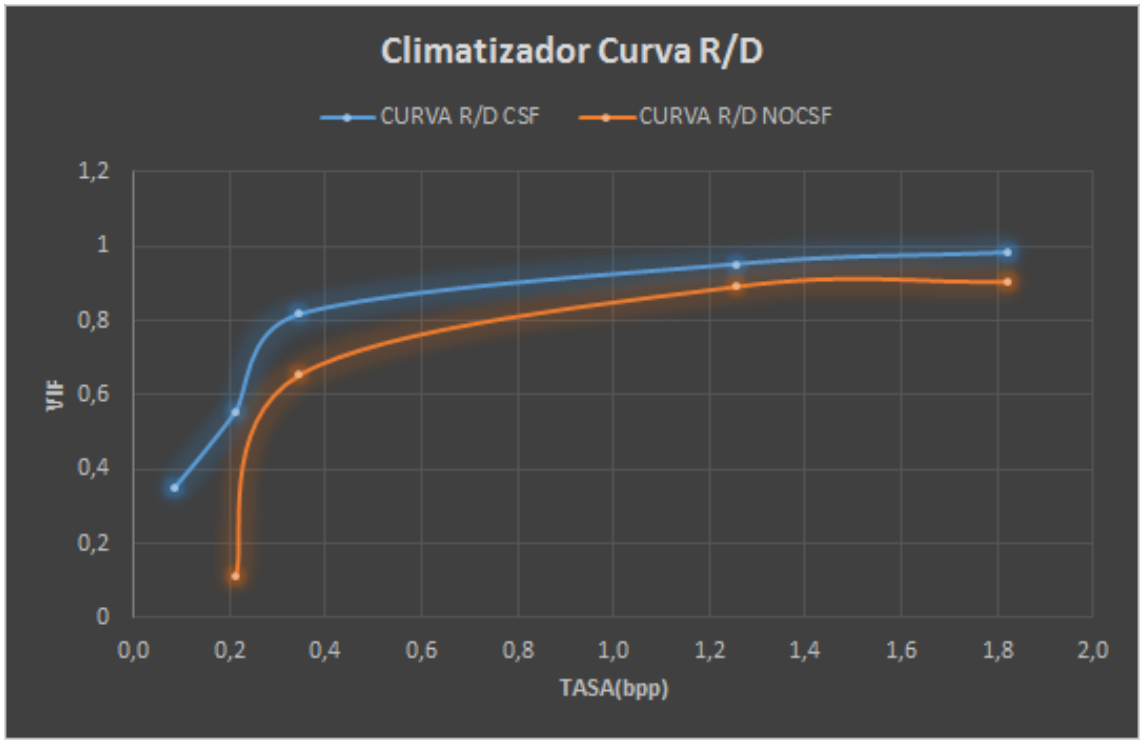


Figura 73: Resultados de la fotografía climatizador

Como podemos observar, en el eje x, tenemos la Tasa(bpp) que nos indica el nivel de compresión la imagen. Por otra parte, en el eje y, tenemos el PSNR en DB(decibelios) o VIF(variance inflation factor) que nos indica la calidad de la imagen para una tasa de compresión dada. Como podemos observar, la mejor calidad se obtiene siempre con CSF. A continuación, en la fotografía Baúl, podemos ver este resultado, analizando un recorte la imagen recuperada tras ser comprimida con y sin CSF:

Figura 74: Fotografía Baúl recuperada tras la compresión con CSF

Figura 75: Fotografía Baúl recuperada tras la compresión sin CSF

Como podemos observar en ambas figuras, la imagen que es recuperada tras ser comprimida con CSF, se ve más definida y por tanto, con mayor calidad.

7. Conclusiones

7.1. Conclusiones finales

Una vez finalizado el proyecto , he llegado a la conclusión de que NDK es una herramienta muy potente que nos permite desde reutilizar librerías ya existentes hasta programas completos escritos en C/C++ pero no debemos dejarnos engañar por este concepto que puede ser simple en principio y muy atractivo puesto que existe mucho código ya escrito en código C/C++ que nos puede resultar útil pero el uso de NDK lleva consigo un aumento de complejidad considerable hasta el punto de que solo debemos usar NDK si es estrictamente necesario o realmente la utilidad o funcionalidades conseguidas con el código nativo reutilizado es superior a la complejidad que este añade. Además, también es importante destacar que con NDK para la implementación de algoritmos matemáticos conseguimos un rendimiento superior al que obtendríamos con el mismo algoritmo escrito en lenguaje java, es por ello que en numerosas aplicaciones Android, sobre todo en videojuegos numerosas partes de estos son implementados en código nativo por el aumento de rendimiento que este produce. Por otra parte, Android es un sistema operativo con el que es muy cómodo programar puesto que es de código libre y hoy en día existe muchísima documentación sobre el además tienen una comunidad de usuarios muy activa y resolver dudas que puedan surgir es muy sencillo.

Además con este proyecto he estado más cerca que nunca del desarrollo de un proyecto real y complejo que sale de las típicas practicas o ejercicios vistos en clase, algo que sin duda creo que me será de mucha utilidad de cara al mundo laboral, además he adquirido muchos conocimientos sobre Android y NDK que me permitirán desarrollar herramientas en un futuro.

7.2. Dificultades encontradas a la hora de realizar el proyecto

A continuación se exponen las dificultades encontradas a lo largo del desarrollo de este proyecto:

- Escasa documentación actualizada sobre NDK.
- Gran dificultad para encontrar errores en tiempo de ejecución con NDK.
- Problemas de compatibilidad con NDK por debajo de la versión 2.3.3 del entorno de desarrollo Android Studio para la versión r15b de NDK.
- Problema a la hora de escribir y leer ficheros con NDK en la memoria interna, debido a la información desactualizada en la red acerca de las rutas(path) que el sistema operativo permite usar.
- Dificultad a la hora de adaptar el código C++ escrito con el entorno de desarrollo Microsoft Visual Studio a NDK, puesto que el compilador de código nativo de NDK es mucho más estricto que visual studio y en numerosas ocasiones ndk-build no nos proporciona con exactitud la información necesaria para encontrar el problema.
- Problema a la hora de escribir ficheros en la tarjeta SD de Android con NDK, debido que a los fabricantes ya no proporcionan de forma transparente la ruta hacia la tarjeta SD, siendo distinta en cada dispositivo y además constando de un número hexadecimal en la ruta(path) que es distinta para cada fabricante.

7.3. Propuestas de mejora

Se proponen las siguientes mejoras para futuros proyectos confeccionados partiendo de este como base:

- Implementar sobre tecnología Android una implementación del algoritmo de compresión que procese las imágenes a color, tomando todas las componentes de la imagen en formato YUV, en vez de únicamente la componente monocromática.
- Implementar una base datos remota que se conecte con la aplicación, actuando como repositorio de imágenes en la nube. Dicho servicio web gestionado por un servlet de java o un servicio PHP que sera el encargado de guardar las imágenes en el servidor.
- Implementar un repositorio de imágenes con google drive o dropbox,haciendo uso de sus APIs públicas, de forma que el usuario siempre tendra acceso a las imágenes comprimidas en su cuenta de dropbox o drive.
- Implementación del algoritmo PETW para vídeo sobre tecnología Android, permitiéndole al usuario grabar un video, comprimiéndolo y descomprimiéndolo en la misma aplicación.
- Realizar el diseño de la aplicación para otros sistemas operativos como IOS, ubuntu phone, windows phone..., de forma que la aplicación esté disponible en la mayoría de los dispositivos móviles.
- Dar soporte en el algoritmo de compresión para más formatos como dng, tif, crw...

8. Bibliografía

Referencias

[1] Sylvain Ratabouil. *ANDROID NDK BEGINNERS GUIDE*. PACKT PUBLISHING,2015.

[2] Nora La Serna, Luzmila Pro Concepción, Carlos Yañez Durán. *Compresión de imágenes: Fundamentos, técnicas y formatos*. 2009.

[3] Android developers NDK,
https://developer.android.com/ndk/index.html?hl=es-419

[4] José Salvador Oliver Gil, Manuel Pérez Malumbres. *On the design of fast and efficient wavelet image coders with reduced memory usage*. Febrero 2006.

[5] Página web jpeg.org ,
https://jpeg.org/jpeg2000/

[6] Adrian Ford, Alan Roberts. *Colour Space Conversions*. Agosto 1998.

[7] D.A Huffman. *A method for the construction of minimum-redundancy codes*. Septiembre 1952.

[8] INTERNATIONAL TELECOMMUNICATION UNION. *JPEG Standard (JPEG ISO/IEC 10918-1 ITU-T Recommendation T.81)*. 1993.

[9] Carlos Rincón, David Bracho, Alfredo Acurero. *ANÁLISIS TEÓRICO DEL ALGORITMO DE COMPRESIÓN LLRUN PARA LA COMPRESIÓN DE CADENAS DE BITS SPARSE)*. Junio 2014.

[10] Marc Antonini, Michel Barlaud, Member, IEEE, Pierre Mathieu, and Ingrid Daubechies, Member, IEEE. *Image Coding Using Wavelet Transform*. Abril 1992.

[11] Amir Said. *Introduction to Arithmetic Coding - Theory and Practice*. Abril 2014.

[12] Miguel Onofre Martínez Rach. *Perceptual image coding for wavelet based encoders*. Diciembre 2014.

[13] Página de Android Developers sobre Camera2api ,
https://developer.android.com/reference/android/hardware/camera2/package-summary.html

[14] Página oficial de Oracle sobre JNI ,
https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/jniTOC.html

[15] Documentación sobre el fichero android.mk ,
http://android.mk/

[16] Documentación sobre el fichero application.mk de Android Developers ,
https://developer.android.com/ndk/guides/application_mk.html