Diseño de algoritmo de visión artificial para la lectura de códigos de barras por medio de dispositivos Android

Miguel Elkin Jiménez Avila

Universidad Santo Tomás

Facultad de Ingeniería Electrónica

Bogotá D.C.

2022

Diseño de algoritmo de visión artificial para la lectura de códigos de barras por medio de dispositivos Android

Miguel Elkin Jiménez Avila

Trabajo de monografía que presenta el diseño de un algoritmo de visión artificial para obtener el título de Ingeniero Electrónico

Director

Ing. Dario Alejandro Segura Torres

Universidad Santo Tomás

Facultad de Ingeniería Electrónica

Bogotá D.C.

2022

Nota de aceptación

_____

_____

_____

_____

_____

_____

Firma del director

_____

Firma del jurado

_____

Firma del jurado

Bogota D.C., 2022

AGRADECIMIENTOS

A mis padres y mi hermana por ser un apoyo incondicional en mi desarrollo personal y profesional.

A mis amigos y compañeros que vivieron la experiencia de estudiar Ingeniería Electrónica junto a mí.

A todas las personas con las que compartí una sonrisa, una anécdota o cualquier momento efímero; a aquellos tartufos que habrían inhibido mi eudaimonía, les agradezco la oportunidad de practicar la resiliencia.

Índice general

Índice de figuras

Índice de tablas

INTRODUCCIÓN

1.1. Planteamiento del problema

Desde las dos últimas décadas los códigos de barras han sido la principal herramienta para la identificación de productos [1]. Generalmente se utiliza un escáner (Lector de código de barras) para ejecutar la tarea de identificación de productos. Los escáneres se integran como un dispositivo externo a algún computador o máquina que sea capaz de comunicarse con el escáner. Existen dos tipos de escáneres, los alámbricos y los inalámbricos; el modo de operación de los escáneres alámbricos como [2] se caracteriza por desplazar el código de barras hasta el escáner para ser leído, por otro lado, los escáneres inalámbricos como [3] se conforman de dos partes, no conectadas físicamente, emisor (Emite el mensaje deificado del código de barras) y receptor (Recepción el mensaje del emisor y lo redirección al computador), y el modo de operación se caracteriza por desplazar el escáner hasta el código de barras para ser leído, no obstante, se presentan restricciones de distancia y obstáculos entre el emisor y el receptor, si no se respetan las restricciones, el mensaje decodificado por el emisor no se recibe o es distorsionado, por lo tanto, el receptor no recibe el mensaje o recibe un mensaje erróneo.

Otra alternativa para la lectura de códigos de barras es la visión artificial, la cual requiere de hardware capaz de procesar imágenes a alta velocidad para tener una experiencia en tiempo real [4]. Los proyectos relacionados con la lectura, reconocimiento e identificación de códigos de barras se desarrollan para su uso en múltiples interfaces y lenguajes que son multiplataforma [5], pero aun así presentan problemas de rendimiento, fiabilidad (Lee

información errónea del código de barras) y efectividad (No lee la información del código de barras o lee información cuando no hay un código de barras) [6], aunque estos contienen gran cantidad de métodos de lectura para los diferentes códigos de barras, estos no son productivos en el campo industrial, el cual requiere de productividad y fiabilidad en sus procesos de reconocimiento de productos.

En la Universidad Santo Tomás (Sede principal Bogotá) existe un sistema de préstamo y recepción de equipos, denominado LabManager, el cual está implementado en un entorno de trabajo que tiene equipos y maquinaria ubicados en diferentes lugares, ellos cuentan con un código de barras para ser identificados. Un dispositivo Android se encarga de la identificación de equipos (Implementación de librería de código libre para la lectura de código de barras) para su préstamo y recepción, pero por falta de fiabilidad y bajo rendimiento entorpece los procesos [7].

1.2. Justificación

Actualmente cualquier entidad necesita cuantificar, registrar y analizar datos para planificar y actuar. Para muchas empresas la gestión de inventario es un reto diario, una de las herramientas que ayuda al control y orden de un inventario es el código de barras. El código de barras es una codificación de uno o más datos datos, comúnmente usado para codificar el identificador de algún objeto o persona [8], en otros casos los códigos de barras pueden almacenar un conjunto de datos como: número serial, precios, método de pago, fechas, direcciones de páginas web, entre otros. Los códigos de barras se dividen en dos grupos, 1D y 2D, los códigos de barras 1D se caracterizan por ser líneas paralelas de diferente grosor y color (Generalmente blancas y negras), los códigos de barras 2D se caracterizan por ser matrices de figuras geométricas de diferente color (Generalmente un conjunto de cuadrados blancos y negros) [9].

Uno de los proyectos de la Facultad de Ingeniería Electrónica de la Universidad Santo Tomás sede principal fue el desarrollo e implementación del sistema LabManager [7], está

compuesto por una aplicación de escritorio, una plataforma web y una aplicación móvil (App basada en Android). El sistema LabManager se encarga de gestionar los datos de préstamo y recepción de equipos de laboratorio, adicionalmente gestiona la información de estudiantes, docentes, auxiliares de laboratorio, inventario de laboratorio, multas y anotaciones. LabManager utiliza la lectura de códigos de barras para ejecutar el préstamo y recepción de equipos de laboratorio, inicialmente se requiere identificar al usuario (La identificación se realiza por medio de la lectura de un código de barras tipo Code-128) y posteriormente se requiere la identificación del equipo a prestar o recepcionar (La identificación se realiza por medio de la lectura de un código de barras tipo Code-39). La aplicación de escritorio y la plataforma web de LabManager realizan la lectura de códigos de barras con ayuda de un escáner (Lector de código de barras) alambico conectado al equipo que este ejecutando el sistema, por otra parte, la aplicación móvil cuenta con la posibilidad de ejecutar la cámara, realizar procesamiento digital de imágenes y aplicar visión artificial, con el fin de leer un código de barras desde una imagen. Se utilizó ZXing, una librería de código libre, con su repositorio disponible en GitHub, para realizar la lectura de códigos de barras desde la aplicación móvil de LabManager [6].

El sistema LabManager está implementado en la Universidad Santo Tomás, en los laboratorios ETM y los laboratorios del edificio Calatayud. La aplicación de escritorio de LabManager está instalada en los computadores del almacén de los laboratorios ETM y la aplicación móvil está instalada un dispositivo móvil que opera en los laboratorios del edificio Calatayud. Se requirió de diferentes implementaciones en cada una de las zonas de laboratorios debido a la modalidad de préstamo y recepción, en los laboratorios ETM los equipos son desplazados por los auxiliares, estudiantes y docentes, y en los laboratorios del edificio Calatayud los auxiliares, estudiantes y docentes tienen que desplazarse al lugar donde están los equipos, debido al tamaño del equipo, no es posible moverlo de su lugar.

Durante la puesta en marcha de la aplicación móvil de LabManager en los laboratorios del edificio Calatayud se detectaron fallas en la lectura de los códigos de barras, se presentaron

errores de fiabilidad, los datos decodificados con la lectura de los códigos de barras no correspondían a los datos esperados, en consecuencia el sistema no permitía seguir el proceso de préstamo y recepción, además se presentaron errores de rendimiento, donde el tiempo mínimo de lectura era hasta 3 segundos y el tiempo máximo era hasta 15 segundos, en el peor de los casos no finalizaba el proceso de lectura.

Los usuarios finales de la aplicación móvil del sistema LabManager son los auxiliares de los laboratorios del edificio Calatayud, los auxiliares al operar con las falencias de la aplicación móvil, poco a poco la dejaron de lado porque no es una herramienta que mejore sus condiciones laborales, al contrario, las empeoro, aumento su tiempo de ejecución de tareas y aumento el número de retrocesos.

La carencia de la aplicación móvil impide el registro sistematizado de la información de préstamo y recepción de los equipos de los laboratorios del edificio Calatayud al sistema LabManager, como consecuencia no se generan los datos que se necesita para hacer el análisis del estado de los equipos, frecuencia de uso y detección de equipos que requieran mantenimiento o sean obsoletos, adicionalmente los auxiliares de laboratorio no cuentan con una herramienta de trabajo que facilite tareas repetitivas, como el préstamo y recepción de equipos.

Este proyecto buscaba desarrollar una librería compatible con dispositivos Android, que implemente técnicas de procesamiento digital de imágenes y visión artificial, y brinde una alternativa para la lectura de códigos de barras Code-128 y Code-39, que son los utilizados en los carnets de la Universidad Santo Tomás y para la identificación de los equipos de laboratorio, respectivamente. El desarrollo de este proyecto permitió que una aplicación Android realice lecturas de códigos de barras, para la identificación de equipos, personal o cualquier objeto que tenga un código de barras Code-39 o Code-128, con el fin de brindar una mejor alternativa que otras librerías de código libre. La culminación de este proyecto brinda un conocimiento útil para los desarrolladores interesados en trabajar con proyectos similares, compitiendo con las librerías de lectura de códigos de barras que existen actualmente.

1.3. Antecedentes

1.3.1. Procesamiento digital de imágenes.

Una imagen digital es un conjunto de píxeles, valores numéricos, que se pueden interpretar como una función $f_(x, y)$, donde x y y son las coordenadas espaciales de un plano. Toda técnica u operación que permita, guardar, transmitir, extraer y comprender la información de una imagen se considera procesamiento digital de imágenes [10].

El procesamiento digital de imágenes es relevante para este proyecto, ya que a partir del procesamiento digital de imágenes se extrae la información necesaria para realizar la decodificación de códigos de barras.

1.3.2. Visión Artificial

Se define la visión artificial como un campo de la inteligencia artificial, que se enfoca en la utilización de técnicas de procesamiento de imágenes para el tratamiento y análisis de información proveniente de imágenes digitales, así como se menciona en [11].

Dentro del campo de la inteligencia artificial existe el subcampo del aprendizaje profundo (Deep Learning), el cual se centra en el desarrollo de redes neuronales, estas pretenden simular el comportamiento de un cerebro por medio de modelos matemáticos [12]. Deep Learning es relevante para este proyecto debido a que a partir de redes neuronales se realiza visión artificial, dando la oportunidad de leer códigos de barras desde imágenes digitales.

Localización y detección de códigos de barras.

Una solución efectiva para la localización y detección de objetos es la implementación de una red neuronal y de la transformada de Hough según el proyecto “Automated barcode recognition for smart identification and inspection automation”, el proyecto ha utilizado

estas técnicas para la localización y detección de códigos de barras, incluso si este código de barras presenta daños. Esto es llamado, sistema inteligente de detección y reconocimiento de códigos de barras (SBDR por sus siglas en inglés). Este proyecto está enfocado a la localización y detección de códigos de barras de tipo Code-39, por lo tanto, no cumple con los requerimientos planteados para solucionar la problemática del proyecto de este documento [13]. En la Figura 1.1 se muestra el flujo que siguió el sistema planteado en dicho proyecto.

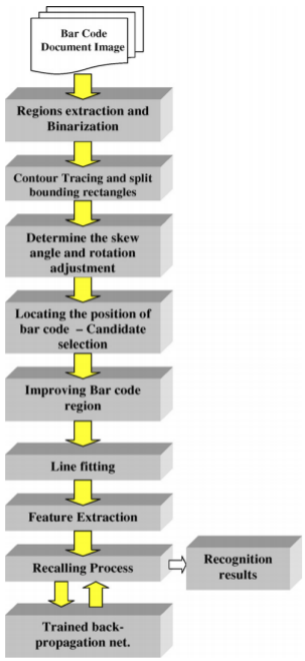

Figura 1.1: Diagrama de bloques para el sistema SBDR [13]

Localización de múltiples códigos de barras 1D y 2D.

El proyecto "GBVS based 1D and 2D barcodes Localization in complex scene" plantea un algoritmo capaz de localizar uno o múltiples códigos de barras, sin importar la cantidad, tamaño, ángulo de percepción y fondos. Utilizando un algoritmo de GBVS (Graph-Based Visual Saliency) y definiendo múltiples valores de umbral para la segmentación pueden localizar en una imagen códigos de barras 1D y 2D [14], sin embargo, no es totalmente fiable, como se observa en la Figura 1.2 sección f, un segmento de la imagen no corresponde a un código de barras. Dicho proyecto es relevante ya que es necesario la localización de los códigos de barras Code-39 y Code-128, pero este no llega al punto de identificar la

información de los códigos de barras, lo cual es necesario.

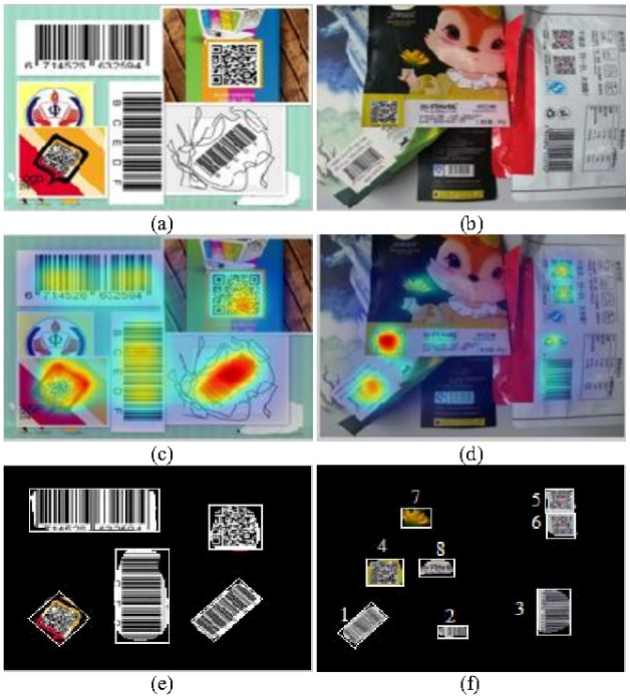

Figura 1.2: Localización de múltiples códigos de barras [14]

Librerías de código abierto.

El desarrollo de proyectos de visión artificial, requieren de un arduo trabajo de programación y cálculos matemáticos, por fortuna existen librerías que facilitan el trabajo.

ZBar es una librería de uso libre para la lectura de códigos de barras, desarrollada en C, esta tiene interfaces en los siguientes lenguajes de programación: Python, Perl, C++, es compatible para los siguientes sistemas operativos: Windows, Linux y IOS [15].

OpenCV es una librería de uso libre para la visión artificial, desarrollada en C/C++, esta tiene interfaces en los siguientes lenguajes de programación: Python, C++ y Java, es compatible para los siguientes sistemas operativos: Windows, Linux, Mac OS, IOS y Android [5]. OpenCV proporciona una alternativa de escáner de códigos QR en el cual se localiza y se identifica el un código QR. Los desarrolladores de OpenCV compararon su escáner de códigos QR con un módulo llamado ZBar y el resultado de esta comparación fue que ZBar es mucho mejor que OpenCV QR CODE [16].

OBJETIVOS

2.1. Objetivo general

Generar una librería de lectura fiable y ágil para los códigos de barras Code-39 y Code-128 basado en un algoritmo propietario, enfocado al desarrollo de aplicaciones Android, para brindar una mejor alternativa que otras librerías de código libre.

2.2. Objetivos específicos

- Consultar técnicas de procesamiento digital de imágenes y visión artificial para analizar y sintetizar las imágenes de códigos de barras.
- Evaluar diferentes tecnologías para el desarrollo de aplicaciones Android para seleccionar uno, según su rendimiento en las operaciones de visión artificial.
- Implementar al menos dos herramientas de uso libre para lectura de códigos de barras, en un dispositivo Android con el fin de determinar su tiempo de lectura y fiabilidad.
- Diseñar e implementar, en un dispositivo Android, un algoritmo para la lectura de códigos de barras desde una imagen, basado en técnicas de visión artificial para determinar su tiempo de lectura y fiabilidad.
- Desarrollar la librería de lectura de códigos de barras, basada en el algoritmo diseñado, generando una herramienta para el desarrollo de aplicaciones Android que

requieran lectura de códigos de barras.

- Comparar los resultados obtenidos por cada una de las librerías empleadas para determinar el mejor funcionamiento respecto a tiempo de lectura y fiabilidad.

MARCO TEÓRICO

Con el fin de tener una referencia para el lector se presenta el siguiente marco teórico donde se exponen conceptos básicos de procesamiento de imágenes y visión artificial que fueron fundamentales para el desarrollo de este proyecto.

3.1. La imagen

Una imagen es la representación, reproducción o recreación, de una persona, animal, objeto o idea [17]. Dentro del contexto del procesamiento de imágenes y visión artificial una imagen se puede definir como una función de dos dimensiones $f(x, y)$, donde x y y son las coordenadas del plano que representan. El rango o amplitud de la imagen está dado por un escalar en el caso de las imágenes en escala de grises o por un vector en el caso de las imágenes a color, a esto se le conoce como píxel. Existen diversos modelos que describen el color de la imagen, entre los más populares está el modelo RGB, su nombre debido a las iniciales en inglés (Red - Green - Blue), este modelo está basado en el sistema cartesiano de tres ejes, para comprender su comportamiento se plantea que cada uno de los ejes representa un color (Rojo - Verde - Azul), se asume que el rango es el conjunto cerrado $[0, 1]$, posteriormente se posiciona en el espacio un cubo de arista de 1, donde un vértice se ubica en el origen y el vértice opuesto se ubica en las coordenadas 1,1. La diagonal del cubo que se forma de las coordenadas 0,0 a 1,1 representa la transición de color de negro a blanco, y cada una de las aristas del cubo representa la transición de color de un vértice a otro [10]. Los colores que se ubican en los vértices son los siguientes:

rojo, verde, azul, negro, blanco, magenta, cian y amarillo. En la Figura 3.1 se aprecia el comportamiento del modelo RGB.

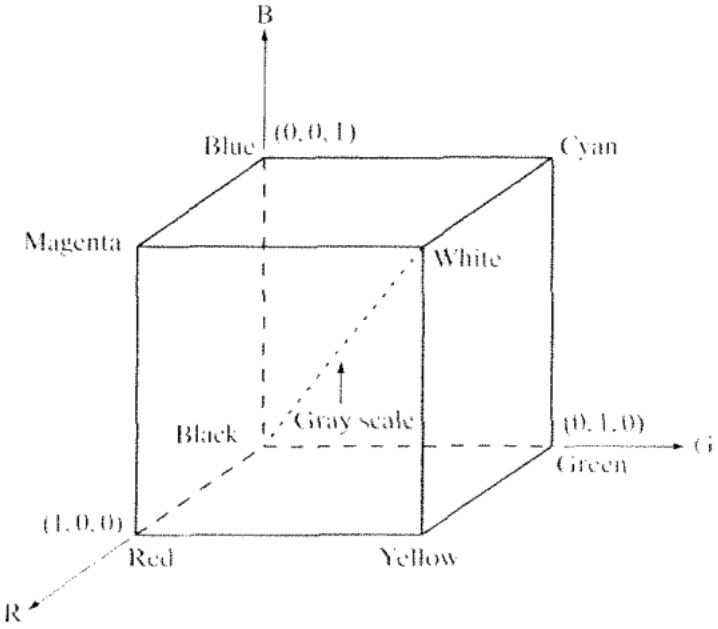

Figura 3.1: Esquema del modelo RGB [10]

Para el dispositivo encargado de realizar la tarea de procesar imágenes (microprocesador, microcontroladores, etc) una imagen no es más que un conjunto de bits organizados de determinada manera, de la cual puede leer píxeles o las componentes de un píxel (Por ejemplo el valor del componente rojo del píxel). El número de bits usados para la representación de un píxel es llamado profundidad del píxel. En general para el modelo RGB el rango de color es el conjunto cerrado $[0, 255]$, por lo tanto en un byte se puede almacenar una componente del color de un píxel (rojo, verde o azul), por ende se requieren de 3 bytes para formar un píxel, por lo tanto, la profundidad del píxel es de 24 bits [10].

3.2. La importancia del color

El procesamiento de imágenes y visión artificial utilizan la composición de color de una imagen principalmente por dos factores: el color es el mejor método para la identificación y extracción de objetos de una imagen, y el ojo humano puede distinguir entre miles de

colores, sombras e intensidades, a comparación de los pocos tonos de grises que puede identificar. Por lo tanto el uso del color en el procesamiento digital de imágenes es fundamental para operaciones de segmentación e identificación de información.

La luz es una onda electromagnética y como toda onda tiene una longitud de onda. La luz visible tiene un rango entre los 400 y 700 nm en su longitud de onda, gracias a los trabajos de investigación realizados por Sir Isaac Newton en 1666, se conoce que un haz de luz blanca está conformado por diferentes ondas que forman el espectro de luz visible [10]. Gracias a la física óptica se implementaron técnicas procesamiento digital de imágenes que contemplen el color, como el modelo RGB.

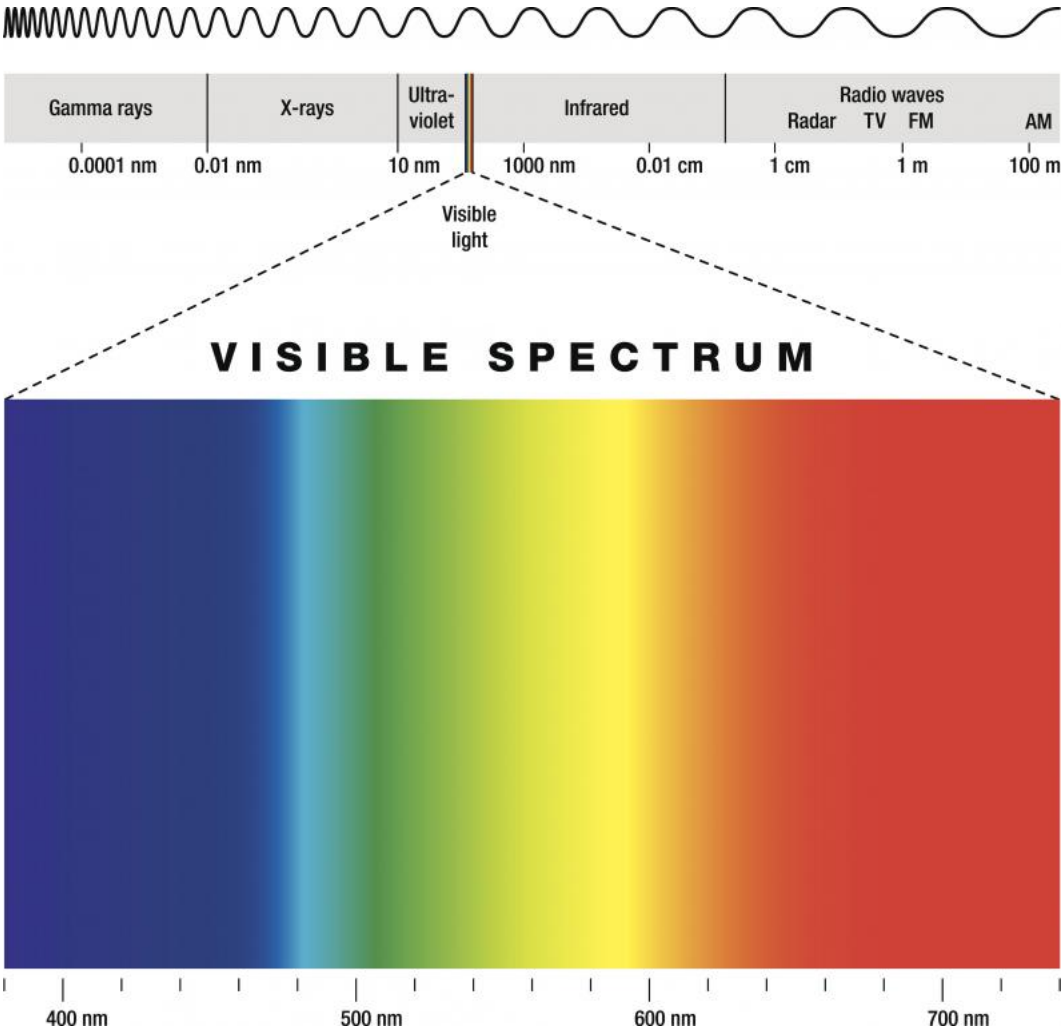

Figura 3.2: Espectro visible [18]

3.3. Transformación de color

En diferentes operaciones morfológicas se requiere una imagen en escala de grises o en blanco y negro, para ello se debe realizar una transformación de color. Tomando en consideración que una imagen se puede representar de la forma $f(x, y)$, se plantea que una transformación de color se representa de la siguiente forma:

$$g(x, y) = T[f(x, y)] \tag{3.1}$$

Donde f es la imagen de entrada, T es la transformada y g es la imagen resultante. La transformada es el proceso al cual es sometida la imagen de entrada, esta puede ser una imagen a color, en escala de grises o a blanco y negro. A continuación, se presentan algunas transformadas, tener presente que: p es el escalar de un punto euclidiano que representa a $g(x, y)$, por otro lado, R, G y B son las componentes del vector de un punto euclidiano que representa $f(x, y)$ [10].

3.3.1. Transformación ponderada

Como se puede apreciar en la Figura 3.2 el espectro visible muestra las bandas de colores rojo, verde y azul, cada banda de color abarca diferentes longitudes de onda, al calcular los rangos para cada color se obtienen los porcentajes de cada uno de ellos dentro del rango visible del espectro. La transformación ponderada toma cada píxel de una imagen a color en el modelo RGB y aplica una sumatoria ponderada de cada una de las componentes del píxel [19], así como se expresa en la siguiente ecuación:

$$p = 0{,}3 * R + 0{,}59 * G + 0{,}11 * B \tag{3.2}$$

3.3.2. Transformada por promedio

Según la Figura 3.1 una imagen a color con el modelo RGB se compone de un plano euclidiano de vectores, los cuales se conforman por los valores que representan cada uno de los colores (rojo, verde y azul). La transformada por promedio toma cada píxel de una

imagen a color en el modelo RGB y aplica el promedio de las componentes de cada vector en cada punto de la imagen [19], así como se expresa en al síguete ecuación:

$$p = (R + G + B)/3 \tag{3.3}$$

3.3.3. Transformada por umbral

Uno de los métodos más simples de segmentar un imágenes es aplicar una transformada por umbral, la cual convierte una imagen en escala de grises a blanco y negro. Para lograr ilustrar correctamente la transformada por umbral se asumen las variable r y s como el nivel de gris de $f(x, y)$ y $g(x, y)$ correspondientemente, evaluadas en cualquier punto (x, y), dando como resultado la siguiente ecuación:

$$s = T[r] \tag{3.4}$$

En esta transformada se debe definir previamente un valor umbral, que se encuentre dentro del rango permitido por la profundidad de los píxeles de la imagen. Se evalúa el nivel de gris de determinado píxel según el umbral. Si el nivel de gris es mayor o igual al umbral se produce un píxel blanco, por otro lado si el nivel de gris es menor al umbral definido se produce un píxel negro. Esta transformada genera una imagen binaria, en la cual sus píxeles pueden tomar el valor mínimo o máximo permitido por la profundidad de píxel [10]. En la Figura 3.3 se aprecia la relación entre r y s, teniendo en cuenta que el umbral en la figura se representa como m.

3.4. Teoría de conjuntos

Los conjuntos son el grupo, lista, colección o clase de elementos con una o más características en común. Para la denotación de un conjunto se utilizan letras mayúsculas, como ejemplo las afirmaciones, x pertenece a el conjunto C, x no pertenece a el conjunto C, el conjunto C es nulo o vació y el conjunto C está conformado por los elementos 1, 2

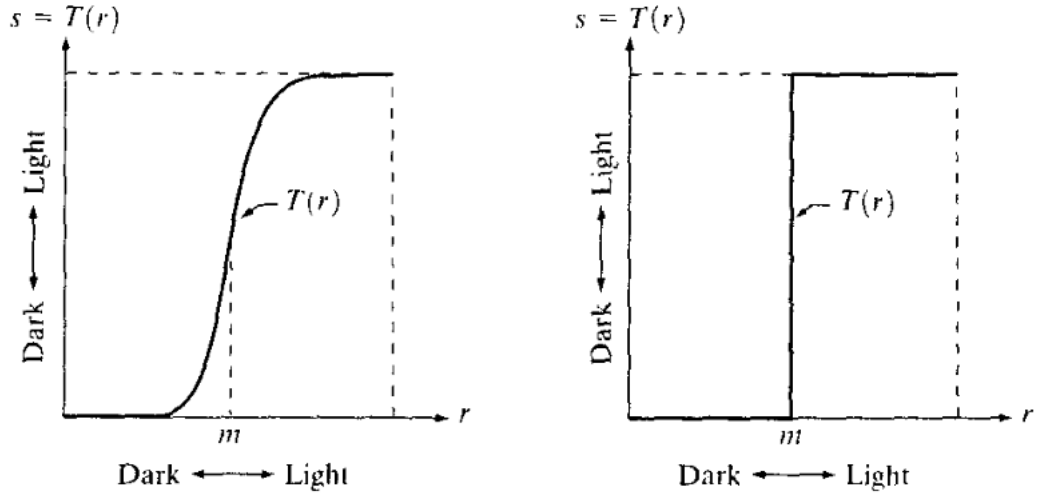

Figura 3.3: Transformada por umbral [10]

y 3, se expresan en las ecuaciones 3.5, 3.6, 3.7 y 3.8, respectivamente [20].

$$x \in C \tag{3.5}$$

$$x \notin C \tag{3.6}$$

$$C = \emptyset \tag{3.7}$$

$$C = \{1, 2, 3\} \tag{3.8}$$

Dentro de la teoría de conjuntos se definen propiedades y operaciones entre ellos, a continuación, se presentan los casos más relevantes. Como primer caso se encuentra el subconjunto, este caso se da cuando todos los elementos de un conjunto A están presentes en el conjunto B, pero no todos los elementos del conjunto B están presentes en el conjunto A. El segundo caso es la unión, este caso se da cuando se crea un conjunto con todos los elementos de un conjunto A y todos los elementos de un conjunto B. El tercer caso es la intersección, en este caso se contemplan los elementos que tiene en común dos conjuntos para formar otro conjunto. Como cuarto y último caso se tiene los conjuntos disyuntos, este caso se da cuando la intersección entre dos conjuntos da como resultado un conjunto nulo o vacío. Los cuatro casos mencionados anteriormente se expresan en las siguientes ecuaciones, 3.9, 3.10, 3.11 y 3.12, respectivamente [20].

$$A \subseteq B \tag{3.9}$$

$$A \cup B \tag{3.10}$$

$$A \cap B \tag{3.11}$$

$$A \cap B = \emptyset \tag{3.12}$$

3.5. Operaciones morfológicas

En el contexto de procesamiento de imágenes y visión artificial se habla de morfología de la imagen para referirse a los procesos a los que son sometidas las imágenes con el fin de extraer información relevante, como ejemplo la huella digital en una imagen. Entre las operaciones morfológicas más relevantes se encuentran, la dilatación, la erosión, la apertura y el cierre.

3.5.1. Dilatación

Esta operación morfológica consiste en la maximización de las regiones brillantes de una imagen. Para aplicar dilatación a una imagen se requiere de un kernel (Imagen más pequeña que la imagen a dilatar, puede tener cualquier figura, por lo general se aplica con una cruz o circulo) y se consideran las imagen como conjuntos, donde la imagen a dilatar se define como el conjunto A y el kernel se define como el conjunto B, la dilatación se denota como [10]:

$$A \oplus B = \left\{ z | \left[\hat{B} \cap A \right] \subseteq A \right\} \tag{3.13}$$

Por lo tanto, la dilatación consiste en la intersección de la negación de B con A que sea subconjunto de A para todo elemento z. Es importante aclarar que esta es una de las formas de interpretar y aplicar dilatación, entre muchas otras. En la Figura 3.4b se aprecia el resultado de aplicar dilatación a la Figura 3.4a.

(a) Imagen base

(b) Dilatación de 3.4a

Figura 3.4: Aplicación de dilatación [21]

3.5.2. Erosión

La erosión comparte muchas similitudes con la dilatación, requieren de un kernel, hay gran variedad de formas de interpretación y aplicación. A diferencia de la dilatación, la erosión consiste en la minimización de las regiones brillantes, la operación inversa a la dilatación. La erosión se puede expresar (Se maneja la misma nomenclatura para la imagen y el kernel, conjuntos A y B, respectivamente) como [10]:

$$A \ominus B = \{z | B \subseteq A\} \tag{3.14}$$

Por lo tanto, la erosión se define como: kernel como subconjunto de la imagen para todo elemento z. En la Figura 3.5b se aprecia el resultado de aplicar erosión a la Figura 3.5a.

(a) Imagen base

(b) Erosión de 3.5a

Figura 3.5: Aplicación de erosión [21]

3.5.3. Apertura

La apertura es una operación morfológica particular, ya que es la aplicación de dos operaciones morfológicas, erosión seguida de dilatación. Esta operación morfológica elimina

el ruido del contorno de una imagen y se puede expresar como [10]:

$$A \circ B = (A \ominus B) \oplus B \tag{3.15}$$

En la Figura 3.6b se aprecia el resultado de aplicar apertura a la Figura 3.6a.

(a) Imagen base

(b) Apertura de 3.6a

Figura 3.6: Aplicación de apertura [21]

3.5.4. Cierre

Al igual que la apertura, el cierre consiste en la aplicación de dos operaciones morfológicas, dilatación seguido de erosión. Esta operación morfológica elimina el ruido de los contornos de objetos en una imagen y se puede expresar como [10]:

$$A \bullet B = (A \oplus B) \ominus B \tag{3.16}$$

En la Figura 3.7b se aprecia el resultado de aplicar cierre a la Figura 3.7a.

(a) Imagen base

(b) Cierre de 3.7a

Figura 3.7: Aplicación de cierre [21]

3.6. Histograma

Un histograma es un diagrama de barras que representa una variable de tipo cuantitativa, el cual permite analizar el comportamiento de un grupo de datos agrupados por clase, por lo tanto, el histograma se entiende como un gráfico de frecuencia de datos [22].

El histograma de una imagen es la representación gráfica de la frecuencia de cada uno de los píxeles que la compone. El histograma de una imagen en escala de grises almacenada digitalmente en bytes tiene 256 clases, debido a que cualquier píxel de la imagen puede tomar un valor dentro del rango de $[0, 255]$, un histograma se puede representar de la siguiente manera [10]:

$$h\left(r_k\right) = n_k \tag{3.17}$$

Donde r_k es la clase de píxel y n_k es el número píxeles que pertenecen a la clase r_k. Generalmente se requiere analizar ocurrencia de clases por medio de valores ponderados, para ellos se deben modificar los resultados del histograma aplicando normalización, la cual se representa como [10]:

$$p\left(r_k\right) = \frac{n_k}{n} \tag{3.18}$$

Donde n es la cantidad de datos (Cantidad de píxeles que conforman una imagen), es importante notar que la sumatoria de las ocurrencias de cada clase es igual a 1. En la Figura 3.8 se aprecia el histograma de dos tipos de imágenes, oscura y brillante.

3.7. Transformada de Hough

Tomando en consideración que una imagen es un conjunto de píxeles al cual se le aplican operaciones (Por ejemplo la operación morfológica cierre) para obtener información, la transformada de Hough es una operación que permite identificar las líneas que se forman con la relación de un píxel a otro. Hough plantea que considerando un píxel (x_i, y_i) como un punto que cruzan determinadas rectas, se puede expresar cualquiera de esas rectas con la ecuación 3.19, donde a y b tomaran múltiples valores, por lo tanto la ecuación previa se puede escribir de la forma en la que se expresa la ecuación 3.20. Para hallar la recta que pasa por los puntos o píxeles (x_i, y_i) y (x_j, y_j) se requiere trabajar en el espacio de parámetros (Plano a-b), variando el valor de a en la ecuación 3.20 por cada uno de los puntos a evaluar, generando así dos rectas, con el fin de hallar el punto de intersección entre las rectas. El punto de intersección de las rectas indica los valores a' y b', si estos son reemplazados en la ecuación 3.19 se obtiene la ecuación de la recta que cruza los puntos

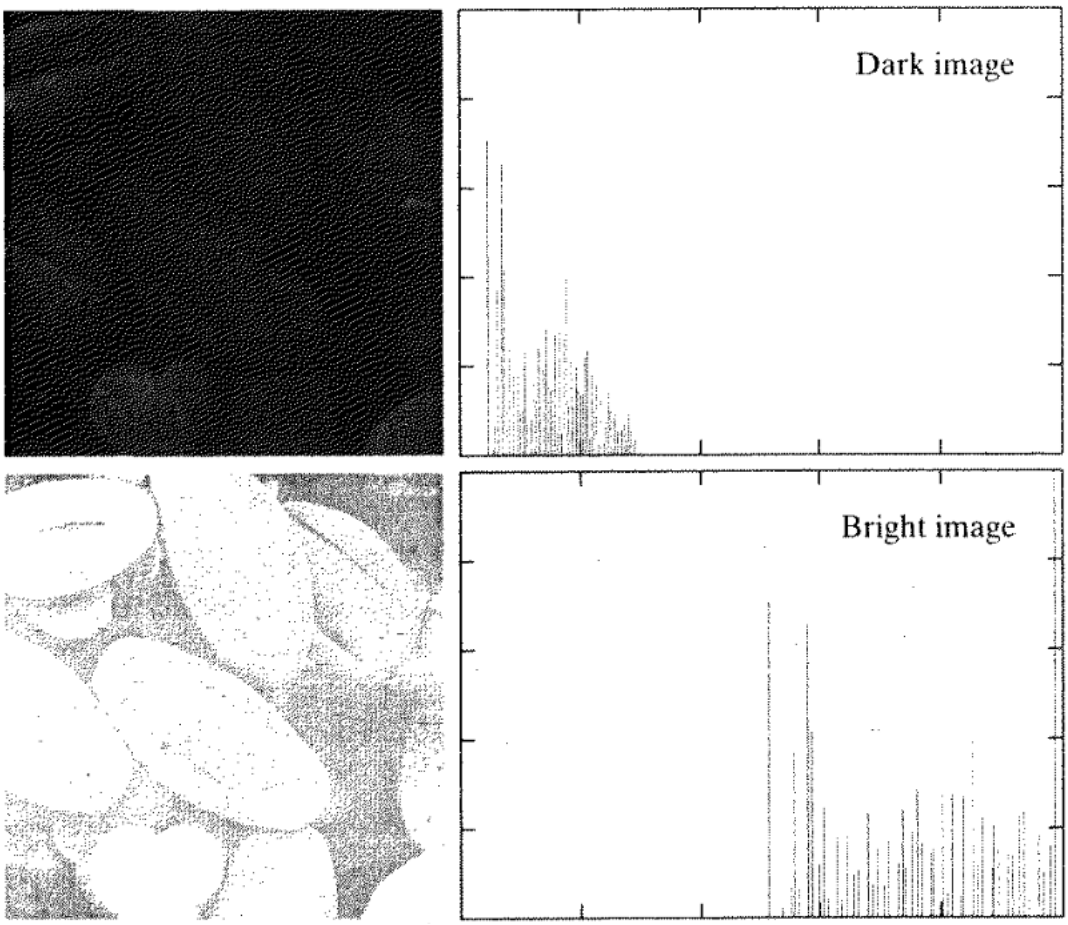

Figura 3.8: Histograma de imágenes [10]

(x_i, y_i) y (x_j, y_j) [10]. En la Figura 3.9 se aprecia el plano x-y y a su derecha el plano a-b.

$$y_i = ax_i + b \tag{3.19}$$

$$b = -ax_i + y_i \tag{3.20}$$

3.7.1. Transformada de Hough mejorada

La transformada de Hough tradicional implica un gran procesamiento y alto espacio de memoria para el dispositivo que la implemente, ya que por cada punto de la imagen tiene que crear rectas que lo relacionen con otros puntos y almacenar esa información para determinar las rectas más predominantes en una imagen, por lo tanto, para reducir las operaciones en el proceso se plantea usar la ecuación normal de la recta, que se representa

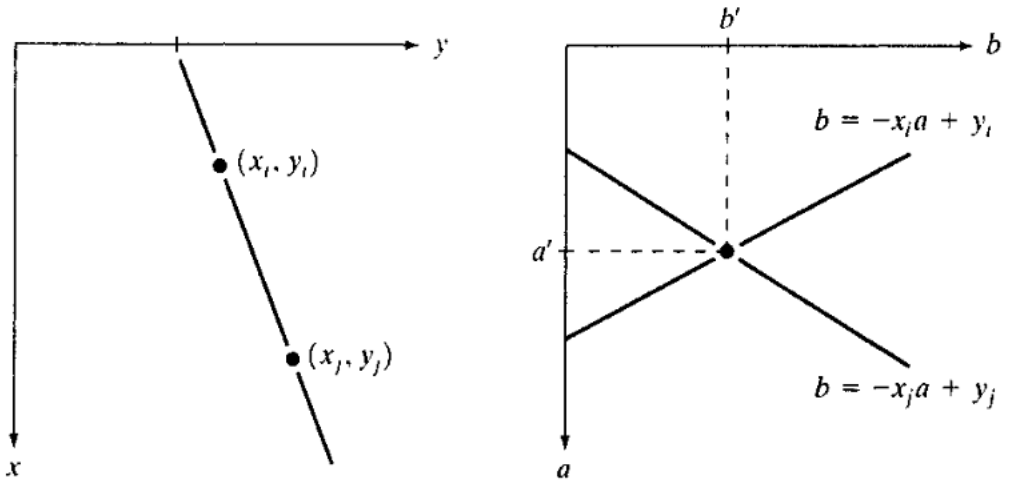

Figura 3.9: Planos x-y y a-b [10]

como:

$$x\,cos\Theta + y\,sen\Theta = \rho \tag{3.21}$$

Donde ρ es la distancia entre el origen y la recta de interés, y Θ es el ángulo que se forma entre el eje x y una recta perpendicular a la recta de interés. El uso de esta representación consiste en la evaluación del rango de Θ, el cual es de $-90°$ a $90°$, y construir una tabla de acumulados. Por cada punto de una imagen se evalúa la ecuación 3.21 dentro del rango de Θ, obteniendo resultados como en la Figura 3.10 donde se aprecia que el autor hay indicado numéricamente 5 puntos en una imagen y a su derecha se aprecia la transformada de Hough por cada uno de esos puntos. La tabla de acumulados indica que los puntos 1, 3 y 5, forman una de las rectas más predominantes en la imagen, al igual que 2, 3 y 4 [10].

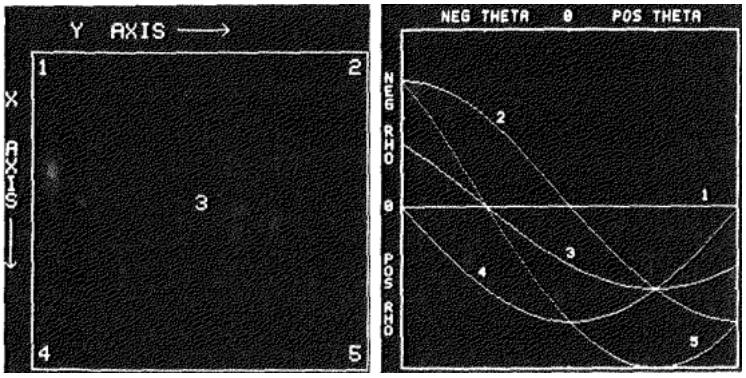

Figura 3.10: Transformada de Hough [10]

METODOLOGÍA

A partir de la necesidad de leer códigos de barras por medio de un dispositivo Android, se busca desarrollar una librería que permita la decodificación de códigos de barras Code-39 y Code-128 desde imágenes, para brindar una mejor alternativa a los proyectos que requieran de este proceso y así dar cumplimiento a cada un de los objetivos planteados. En la Figura 4.1 se aprecia cada una de las etapas del diseño metodológico de este proyecto.

Figura 4.1: Diseño metodológico
Fuente: Autor

4.1. Investigación

Como primer paso del diseño metodológico se elige el tipo de investigación descriptivo que permite la identificación de atributos, propiedades y características del objeto a investigar. Este tipo de investigación resulta ser idóneo para la recopilación de información fundamental para la lectura de códigos de barras, desarrollo de módulos para Android en diferentes tecnologías y el análisis de imágenes por medio del procesamiento digital.

4.2. Selección

En segunda instancia se realiza la selección de la información y las tecnologías relevantes para el desarrollo del proyecto, teniendo en cuenta referencias bibliográficas y análisis del estado del arte, lo cual permite la correcta identificación de las variables que afectan el cumplimiento de los objetivos planteados.

4.2.1. Criterios de selección

- Las técnicas de procesamiento digital de imágenes y visión artificial, se evalúan según su tiempo de ejecución (Rendimiento) y utilidad del resultado.
- Las tecnologías para el desarrollo del proyecto se evalúan según su tiempo de ejecución (Rendimiento).

4.3. Diseño

Basado en los resultados de la etapa de selección, se procede con la etapa de diseño que consiste en el análisis de diferentes variables y procesos, para el desarrollo de un algoritmo de lectura de códigos de barras Code-39 y Code-128.

4.4. Desarrollo

La etapa de desarrollo describe la implementación del algoritmo diseñado en la etapa previa, dando como resultado una librería de códigos de barras para un dispositivo móvil Android, tomando en consideración las diferentes tecnologías para el desarrollo de aplicaciones nativas Android.

4.5. Comparación

Como última etapa, se compara la librería desarrollada en este proyecto con dos librerías de código libre para la lectura de códigos de barras, con el fin de determinar puntos a favor y en contra de la librería desarrollada en este proyecto.

4.5.1. Criterios de comparación

- Se analiza el tiempo de ejecución de la lectura de códigos de barras Code-39 y Code-128 (Rendimiento).
- Se analiza la cantidad de lecturas de códigos de barras Code-39 y Code-128 acertadas (Fiabilidad).

EJECUCIÓN DEL PROYECTO

Se presenta cada una de las etapas necesarias para el cumplimiento de cada uno de los objetivos de este proyecto, dando como resultado una librería de lectura fiable y ágil para los códigos de barras Code-39 y Code-128, basado en un algoritmo propietario, enfocado al desarrollo de aplicaciones Android, para brindar una mejor alternativa que otras librerías de código libre.

5.1. Pruebas de concepto

5.1.1. Técnicas de procesamiento digital de imágenes y visión artificial

Las técnicas de procesamiento digital de imágenes y visión artificial de interés para el desarrollo de este proyecto, son aquellas que permitan la localización y el análisis de información de líneas paralelas sin importar la cantidad de puntos (Píxeles) que conformen cada línea, ya que los códigos de barras Code-39 y Code-128 son un conjunto de líneas paralelas de diferente grosor, organizadas de determinada manera, para la codificación de caracteres alfanuméricos y símbolos.

Pruebas a la transformada de Hough

La primera prueba consistió en la implementación de la transformada de Hough usando imágenes de códigos de barras ideales (imagen a blanco y negro con únicamente la in-

formación del código de barras), donde varía el ángulo de referencia. Ver Figuras 5.1 a 5.8.

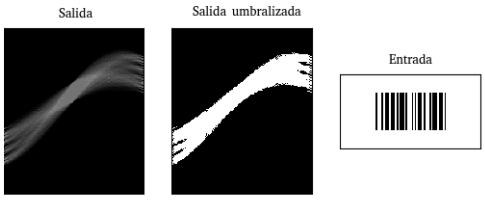

Figura 5.1: Transformada de Hough a código de barras en ángulo 0
Fuente: Autor

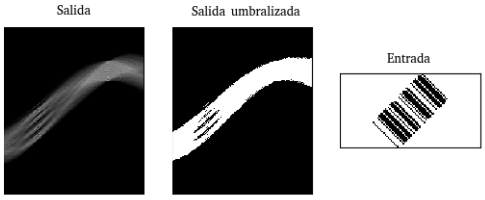

Figura 5.2: Transformada de Hough a código de barras en ángulo 45
Fuente: Autor

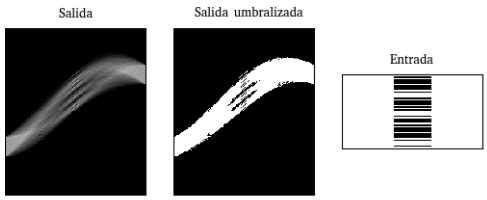

Figura 5.3: Transformada de Hough a código de barras en ángulo 90
Fuente: Autor

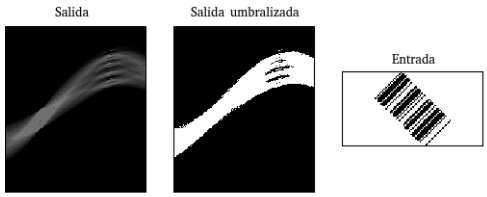

Figura 5.4: Transformada de Hough a código de barras en ángulo 135
Fuente: Autor

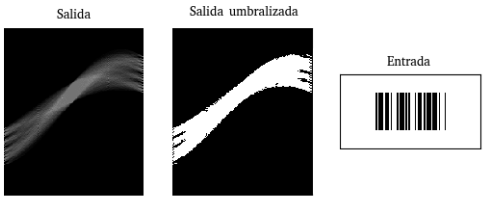

Figura 5.5: Transformada de Hough a código de barras en ángulo 180
Fuente: Autor

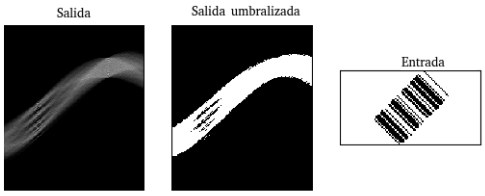

Figura 5.6: Transformada de Hough a código de barras en ángulo 225
Fuente: Autor

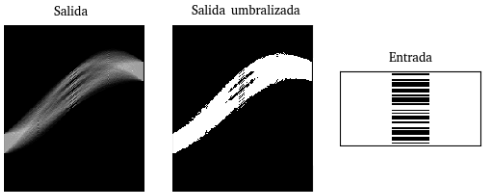

Figura 5.7: Transformada de Hough a código de barras en ángulo 270
Fuente: Autor

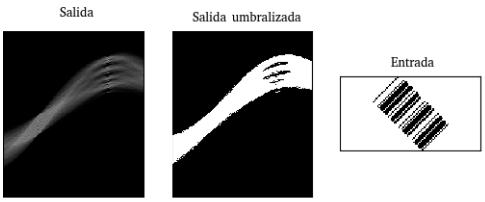

Figura 5.8: Transformada de Hough a código de barras en ángulo 315
Fuente: Autor

La segunda prueba consistió en la implementación de la transformada de Hough usando imágenes de códigos de barras ideales (imagen a blanco y negro con únicamente la información del código de barras), donde varía la posición del código de barras. Ver Figuras 5.9 a 5.13.

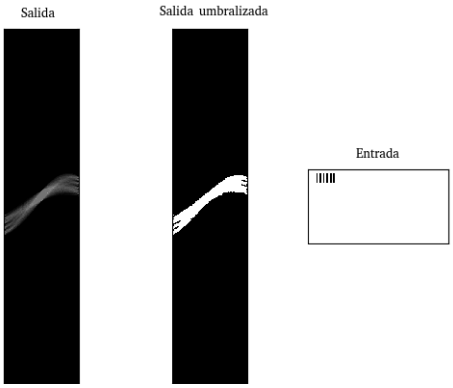

Figura 5.9: Transformada de Hough a código de barras en zona superior izquierda
Fuente: Autor

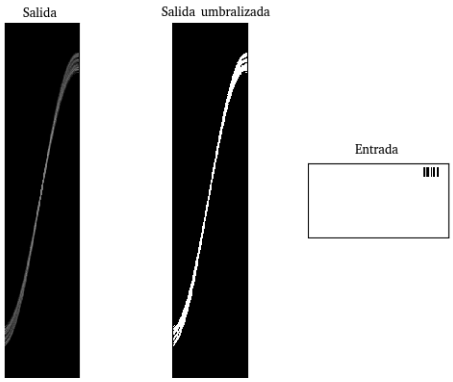

Figura 5.10: Transformada de Hough a código de barras en zona superior derecha
Fuente: Autor

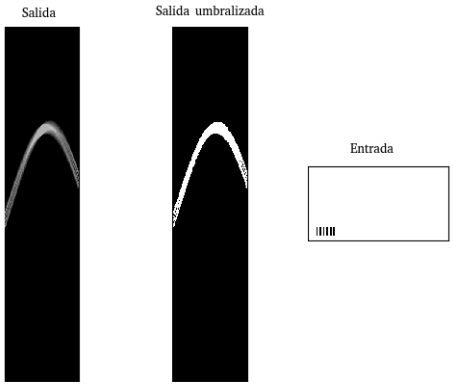

Figura 5.11: Transformada de Hough a código de barras en zona inferior izquierda
Fuente: Autor

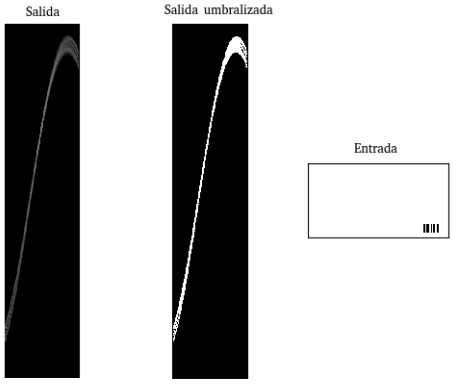

Figura 5.12: Transformada de Hough a código de barras en zona inferior derecha
Fuente: Autor

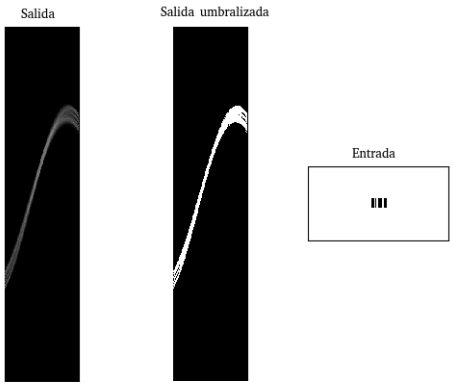

Figura 5.13: Transformada de Hough a código de barras en zona central
Fuente: Autor

Como resultado de estas pruebas se descarta la implementación de la transformada de Hough, debido a:

- El plano a-b requiere de un procesamiento extra para hallar el umbral y obtener la información de los grupos de píxeles que conforman las líneas en una imagen, lo que lo vuelve vulnerable a perder información de código de barras si la imagen tiene más líneas.
- El plano a-b de la transformada de Hough es una imagen con mayor cantidad de píxeles que la imagen procesada, por lo tanto, se requiere de mayor uso de memoria y tiempo de procesamiento.

Pruebas de umbralización por histograma

Esta prueba buscaba hallar un método de poco procesamiento para transformar la imagen a blanco y negro por medio de un umbral. El proceso consiste en obtener el histograma de una imagen, posteriormente calcular la segunda derivada del histograma y finalmente hallar los puntos de umbral. Ver Figuras 5.14 a 5.17.
Los umbrales para aplicar la transformación de color a blanco y negro en esta prueba son: 82, 164 y 202, tal y como se aprecia en la Figura 5.14 (Gráfica de puntos de umbral).

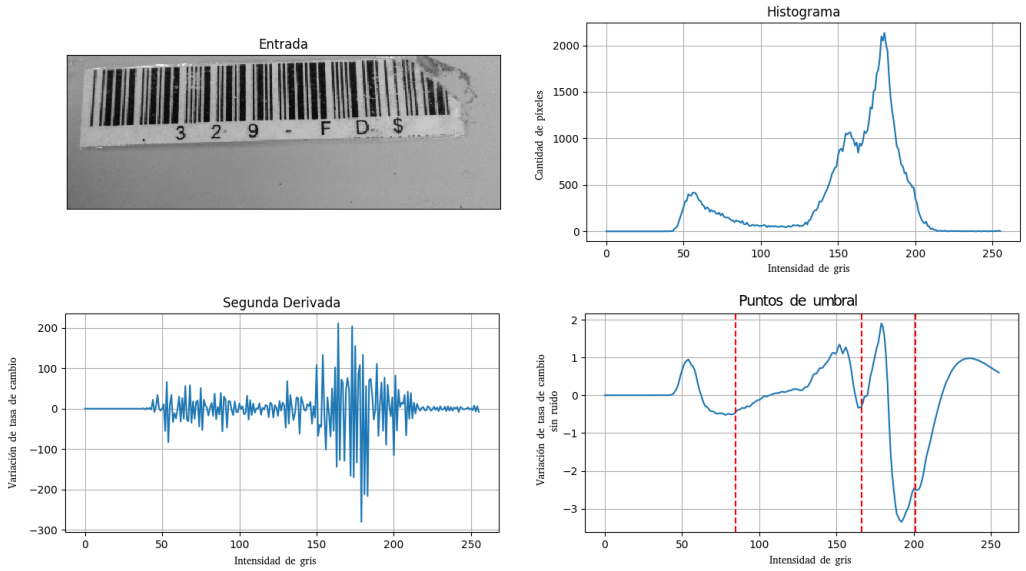

Figura 5.14: Imagen de prueba, histograma, derivada y puntos de umbral
Fuente: Autor

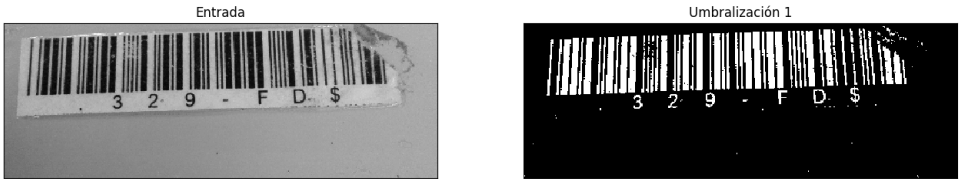

Figura 5.15: Imagen de prueba y umbralización por valor igual a 82
Fuente: Autor

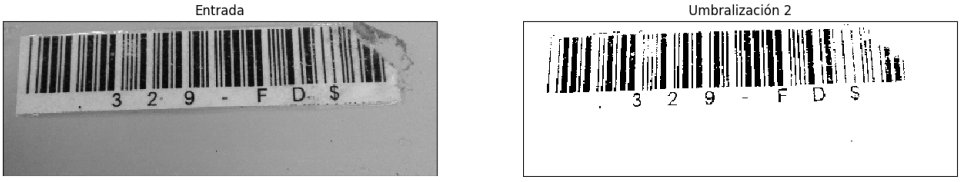

Figura 5.16: Imagen de prueba y umbralización por valor igual a 164
Fuente: Autor

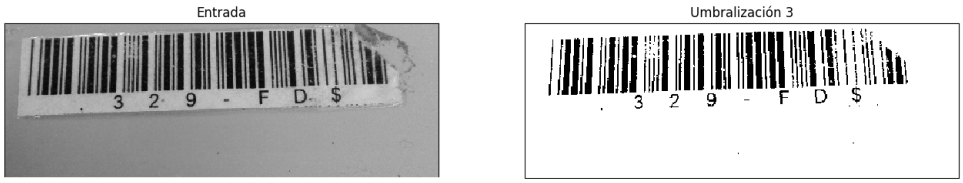

Figura 5.17: Imagen de prueba y umbralización por valor igual a 202
Fuente: Autor

Como resultado de estas pruebas se descarta la implementación del umbral por histograma, debido a:

- La umbralización por histograma produce perdida de información en los bordes de los grupos de píxeles, generando así, distorsión en las líneas que conforman el código de barras. El cambio de grosor de las líneas en un código de barras impide su decodificación.
- Con determinado valor de umbral se intercambia el color blanco con el negro y viceversa, dando como resultado una imagen negativa, por lo tanto este proceso no es consistente en su respuesta.

Pruebas de umbralización por promedio de región

Esta prueba consiste en dividir la imagen en regiones iguales (Subconjuntos de píxeles vecinos), con el fin de obtener por región un 5 % de la información de la imagen y procesar detalladamente la imagen, teniendo en cuenta los pequeños detalles que con un mayor porcentaje se omitirían. Por cada región se calcula el promedio, el promedio calculado sera el umbral para transformar la región de imagen a blanco y negro. Ver Figura 5.18.

Figura 5.18: Umbralización de la Figura 5.25a por promedio de región
Fuente: Autor

Como resultado de estas pruebas se decide implementar el umbral por promedio segmentado, debido a:

- Es un proceso de poco uso de memoria, requiere de 3 variables para su ejecución (sumatoria, cantidad de elementos, índice auxiliar) sin contar con la memoria que ya está ocupada por la imagen.
- Permite mantener la información de la imagen del código de barras (Grosor de cada barra) independientemente del cambio de brillo que tengan los píxeles en las filas y columnas de la imagen.

5.1.2. Tecnologías de desarrollo para Android

El desarrollo de aplicaciones Android se divide en aplicaciones nativas y aplicaciones híbridas. Las aplicaciones nativas se caracteriza principalmente por tener acceso directo a los recursos del sistema. Las aplicaciones híbridas se caracterizan principalmente por tener un código base que puede ser ejecutado por un navegador web. Como opciones de tecnologías para el desarrollo Android se encuentran: Java y Kotlin para aplicaciones nativas [23] y Xamarin con C# para aplicaciones híbridas [24].
La evaluación de cada una de las tecnologías se basa en el rendimiento de procesamiento, lo que conllevó al desarrollo de una aplicación Android que implemente un algoritmo que realice operaciones matemáticas con los elementos de una estructura de datos (análogo

al procesamiento digital de una imagen). El algoritmo de prueba consiste en crear una imagen con alta definición (Resolución de 1920X1080 píxeles) en escala de grises con píxeles de valores aleatorios (Una matriz de números enteros), posteriormente se realiza el procesamiento de la imagen que consiste en calcular el promedio de los píxeles vecinos por cada uno de los píxeles de la imagen y finalmente se calcula el tiempo que tarda el procesamiento. Se tomaron diez datos en cada una de las evaluaciones, debido a que las aplicaciones de prueba no presentaron consistencia en el tiempo de ejecución del algoritmo, principalmente porque la aplicación de prueba se ejecuta sobre un sistema operativo que orquesta más tareas que la afectan sutilmente. En la Figura 5.19 se aprecia el diagrama de flujo de esta evaluación.

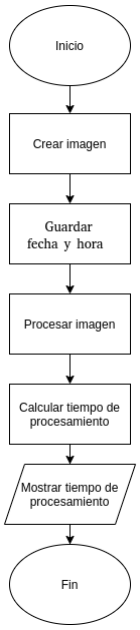

Figura 5.19: Diagrama de flujo para test de tecnologías
Fuente: Autor

Evaluación de rendimiento - Java

Utilizando Android Studio como entorno de desarrollo integrado y con el lenguaje de programación Java, se desarrolló una aplicación Android que ejecuta el algoritmo de la Figura 5.19. La aplicación desarrollada se implementó en dos dispositivos Android para realizar el correspondiente análisis, los resultados obtenidos se presentan en el Tabla 5.1. El resultado promedio de esta evaluación es 28.6ms para el dispositivo Xiaomi Redmi

Note 7 y 247.8ms para el dispositivo Huawei ALE-L23 P8 lite.

	Pruebas (Milisegundos - ms)									
	1	2	3	4	5	6	7	8	9	10
Xiaomi Redmi Note 7	25	31	29	28	26	29	33	25	31	29
Huawei ALE-L23 P8 lite	248	247	248	248	249	246	247	249	249	247

Tabla 5.1: Tiempos de procesamiento - Java
Fuente: Autor

Evaluación de rendimiento - Kotlin

Utilizando Android Studio como entorno de desarrollo integrado y con el lenguaje de programación Kotlin se desarrolló una aplicación Android que ejecuta el algoritmo de la Figura 5.19. La aplicación desarrollada se implementó en dos dispositivos Android para realizar el correspondiente análisis, los resultados obtenidos se presentan en el Tabla 5.2. El resultado promedio de esta evaluación es 39.5ms para el dispositivo Xiaomi Redmi Note 7 y 268.3ms para el dispositivo Huawei ALE-L23 P8 lite.

	Pruebas (Milisegundos - ms)									
	1	2	3	4	5	6	7	8	9	10
Xiaomi Redmi Note 7	40	42	41	43	38	42	38	42	40	39
Huawei ALE-L23 P8 lite	269	266	269	268	270	267	270	269	267	268

Tabla 5.2: Tiempos de procesamiento - Kotlin
Fuente: Autor

Evaluación de rendimiento - C#

Utilizando Visual Studio 2019 como entorno de desarrollo integrado y con el lenguaje de programación C# se desarrolló una aplicación Android que ejecuta el algoritmo de la Figura 5.19. La aplicación desarrollada se implementó en dos dispositivos Android para

realizar el correspondiente análisis, los resultados obtenidos se presentan en el Tabla 5.3. El resultado promedio de esta evaluación es 180.4ms para el dispositivo Xiaomi Redmi Note 7 y 611.4ms para el dispositivo Huawei ALE-L23 P8 lite.

	Pruebas (Milisegundos - ms)									
	1	2	3	4	5	6	7	8	9	10
Xiaomi Redmi Note 7	182	178	185	180	176	179	181	185	174	184
Huawei ALE-L23 P8 lite	611	610	615	611	611	612	611	610	612	611

Tabla 5.3: Tiempos de procesamiento - C#
Fuente: Autor

5.1.3. Resultados de pruebas de concepto

Técnicas de procesamiento digital de imágenes y visión artificial

La umbralización por promedio de región al ser una técnica de procesamiento digital de imágenes con poco procesamiento y poco uso de memoria, presento el mejor rendimiento frente a las otras técnicas evaluadas. La imagen resultante de esta técnica, fue la imagen que respetó más la integridad del grosor de cada barra del código de barras, por lo tanto, esta técnica fue seleccionada para la ejecución de este proyecto.

Tecnologías de desarrollo para Android

En el Tabla 5.4 se aprecia el tiempo promedio de cada una de las pruebas realizadas. Java presentó los mejores resultados dando como conclusión que es el lenguaje de programación más eficiente entre los evaluados para el procesamiento digital de imágenes desde un dispositivo Android, por lo tanto, Java fue seleccionado para la ejecución de este proyecto.

	Tiempo promedio de respuesta (Milisegundos - ms)		
	Java	Kotlin	C#
Xiaomi Redmi Note 7	28.6	39.5	180.4
Huawei ALE-L23 P8 lite	247.8	268.3	611.4

Tabla 5.4: Tiempo promedio de respuesta
Fuente: Autor

5.2. Decodificación de códigos de barras

El código de barras es un conjunto de líneas de diferente color y grosor, la decodificación de un código de barras se basa en la correlación, entre un número binario y un carácter, el número binario puede contener uno o más caracteres, dependiendo de su longitud. Por cada código de barras existe una tabla de caracteres permitidos.

Por convención del autor se crearon 4 clases de grosor de línea (Medida en píxeles), no discrimina entre línea negra o línea blanca, ver Tabla 5.5. Como ejemplo de como se representa un binario en el sistema de clases propuesto por el autor, el número 1001110000 se representa como $SMLX$.

Clase	Grosor (Píxeles)
S	1
M	2
L	3
X	4

Tabla 5.5: Clases de grosor de líneas de códigos de barras
Fuente: Autor

5.2.1. Code-39

Los códigos de barras Code-39 pueden representar un conjunto de letras mayúsculas, números, algunos símbolos y el espacio, el carácter $*$ se usa para indicar el inicio y el fin de un código de barras. A partir del sistema de clases de grosor del Tabla 5.5 se crea el Tabla 5.6.

Líneas	Carácter
LSSSSLSSLS	A
SSLSSLSSLS	B
LSLSSLSSSS	C
SSSSLLSSLS	D
LSSSLLSSSS	E
SSLSLLSSSS	F
SSSSSLLSLS	G
LSSSSLLSSS	H
SSLSSLLSSS	I
SSSSLLLSSS	J
LSSSSSSLLS	K
SSLSSSSLLS	L
LSLSSSSLSS	M
SSSSLSSLLS	N
LSSSLSSLSS	O
SSLSLSSLSS	P
SSSSSSLLLS	Q
LSSSSSLLSS	R
SSLSSSLLSS	S
SSSSLSLLSS	T
LLSSSSSSLS	U
SLLSSSSSLS	V
LLLSSSSSSS	W
SLSSLSSSLS	X
LLSSLSSSSS	Y
SLLSLSSSSS	Z
SSSLLSLSSS	0

LSSLSSSSLS	1
SSLLSSSSLS	2
LSLLSSSSSS	3
SSSLLSSSLS	4
LSSLLSSSSS	5
SSLLLSSSSS	6
SSSLSSLSLS	7
LSSLSSLSSS	8
SSLLSSLSSS	9
SLLSSSLSSS	
SLSSSSLSLS	-
SLSLSLSSSS	$
SSSLSLSLSS	%
LLSSSSLSSS	.
SLSLSSSLSS	/
SLSSSLSLSS	+
SLSSLSLSSS	*

Tabla 5.6: Representaciones para Code-39 [25]

5.2.2. Code-128

Los códigos de barras Code-128 pueden representar un conjunto de letras mayúsculas, minúsculas, números de dos dígitos, algunos símbolos, el espacio y funciones especiales del modo de lectura del código de barras. A partir del sistema de clases de grosor del Tabla 5.5 se crea el Tabla 5.7, notar que los códigos de barras Code-128 permiten tener hasta 3 representaciones (Carácter o función) para un mismo conjunto de líneas, las representaciones se dividen en Code-A, Code-B y Code-C, la interpretación depende de que tipo de representación se defina al inicio del código de barras.

Líneas	Carácter Code-A	Carácter Code-B	Carácter Code-C
MSMMMM			00
MMMSMM	!	!	01
MMMMMS	”	”	02
SMSMML	#	#	03
SMSLMM	$	$	04
SLSMMM	%	%	05
SMMMSL	&	&	06
SMMLSM	’	’	07
SLMMSM	(	(	08
MMSMSL	)	)	09
MMSLSM	*	*	10
MLSMSM	+	+	11
SSMMLM	,	,	12
SMMSLM	-	-	13
SMMMLS	.	.	14
SSLMMM	/	/	15
SMLSMM	0	0	16
SMLMMS	1	1	17
MMLMSS	2	2	18
MMSSLM	3	3	19
MMSMLS	4	4	20
MSLMSM	5	5	21
MMLSSM	6	6	22
LSMSLS	7	7	23
LSSMMM	8	8	24
LMSSMM	9	9	25
LMSMMS	:	:	26

LSMMSM	;	;	27
LMMSSM	<	<	28
LMMMSS	=	=	29
MSMSML	>	>	30
MSMLMS	?	?	31
MLMSMS	@	@	32
SSSLML	A	A	33
SLSSML	B	B	34
SLSLMS	C	C	35
SSMLSL	D	D	36
SLMSSL	E	E	37
SLMLSS	F	F	38
MSSLSL	G	G	39
MLSSSL	H	H	40
MLSLSS	I	I	41
SSMSLL	J	J	42
SSMLLS	K	K	43
SLMSLS	L	L	44
SSLSML	M	M	45
SSLLMS	N	N	46
SLLSMS	O	O	47
LSLSMS	P	P	48
MSSLLS	Q	Q	49
MLSSLS	R	R	50
MSLSSL	S	S	51
MSLLSS	T	T	52
MSLSLS	U	U	53
LSSSML	V	V	54

LSSLMS	W	W	55
LLSSMS	X	X	56
LSMSSL	Y	Y	57
LSMLSS	Z	Z	58
LLMSSS	[	[	59
LSXSSS	\	\	60
MMSXSS	]	]	61
XLSSSS	^	^	62
SSSMMX	_	_	63
SSSXMM	NUL	‘	64
SMSSMX	SOH	a	65
SMSXMS	STX	b	66
SXSSMM	ETX	c	67
SXSMMS	EOT	d	68
SSMMSX	ENQ	e	69
SSMXSM	ACK	f	70
SMMSSX	BEL	g	71
SMMXSS	BS	h	72
SXMSSM	HT	i	73
SXMMSS	LF	j	74
MXSMSS	VT	k	75
MMSSSX	FF	l	76
XSLSSS	CR	m	77
MXSSSM	SO	n	78
SLXSSS	SI	o	79
SSSMXM	DLE	p	80
SMSSXM	DC1	q	81
SMSMXS	DC2	r	82

SSXMSM	DC3	s	83
SMXSSM	DC4	t	84
SMXMSS	NAK	u	85
XSSMSM	SYN	v	86
XMSSSM	ETB	w	87
XMSMSS	CAN	x	88
MSMSXS	EM	y	89
MSXSMS	SUB	z	90
XSMSMS	ESC	{	91
SSSSXL	FS	—	92
SSSLXS	GS	}	93
SLSSXS	RS	∼	94
SSXSSL	US	DEL	95
SSXLSS	FNC_3	FNC_3	96
XSSSSL	FNC_2	FNC_2	97
XSSLSS	SHIFT	SHIFT	98
SSLSXS	CODE_C	CODE_C	99
SSXSLS	CODE_B	FNC_4	CODE_B
LSSSXS	FNC_4	CODE_A	CODE_A
XSSSLS	FNC_1	FNC_1	FNC_1
MSSXSM	START_CODE_A	START_CODE_A	START_CODE_A
MSSMSX	START_CODE_B	START_CODE_B	START_CODE_B
MSSMLM	START_CODE_C	START_CODE_C	START_CODE_C
MLLSSSM	END	END	END

Tabla 5.7: Representaciones para Code-128 [26]

5.3. Definición final del algoritmo

Se busca generar un algoritmo de pocos pasos y poco uso de memoria, por lo tanto, se diseña el siguiente algoritmo:

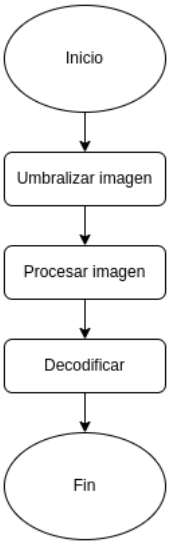

Figura 5.20: Algoritmo del lector de código de barras diseñado
Fuente: Autor

1. La umbralización consiste en la transformación de una imagen, el resultado es una imagen a blanco y negro. Para este proyecto se desarrolló e implementó un método propio, la umbralización de la imagen por promedio de región detallada en la página 43, y se implementó la umbralización adaptativa que provee OpenCV utilizando el método gaussiano.

   La umbralización por promedio de región analiza la imagen como una matriz y se divide en regiones (matrices) que contienen el mismo número de filas pero no el mismo número de columnas, por cada región se calcula el promedio del valor de los elementos que la conforman y finalmente se reemplaza el valor de cada elemento, si el valor del elemento es menor al valor del promedio su valor se reemplaza por un cero, pero si su valor es mayor o igual al promedio su valor se reemplaza por un 255.

   A partir del mismo proceso para obtener regiones de la imagen, mencionado previamente, la umbralización adaptativa de OpenCV se implementó pasando como

parámetros la región a umbralizar, un 255 como máximo valor, un 1 que representa el método gaussiano, un 0 que representa el tipo de umbral, el valor correspondiente al tamaño de la región y un 0 que representa la contante que resta la media de la región.

2. El procesamiento de imágenes consiste en aplicar la operación morfológica apertura, detallada en la página 26, con el fin de eliminar el ruido del contorno de la imagen. La operación morfológica de apertura requiere de la imagen a la cual va a procesar y de un kernel que defina que tipo de apertura realiza. Para poder cumplir con la mayor cantidad de casos de imágenes a decodificar, se requirió un proceso iterativo, por cada iteración cambia el kernel.

   Para el flujo que sigue la umbralización por promedio de región se definieron 4 kernels para el proceso iterativo; primero un kernel igual a null, como resultado no se aplica ningún cambio en la imagen, efectivo para imágenes ideales; segundo un kernel de 5 filas y 5 columnas, los elementos de la columna 3 tienen como valor 255, los demás elementos tiene como valor 0, efectivo para imágenes con códigos de barras que presentan espacios en sus líneas negras; tercero un kernel de 5 filas y 5 columnas, todos los elementos tienen como valor 255; efectivo para imágenes con códigos de barras que presentan espacios en sus líneas negras y blancas, no mayores a 4 píxeles; cuarto un kernel de 7 filas y 7 columnas, todos los elementos tienen como valor 255; efectivo para las imágenes con códigos de barras que presentan espacios en sus líneas negras y blancas, cubre casos que el kernel anterior no logra cubrir, pero con el kernel 4 la operación morfológica apertura requiere de más recursos y tiempo de ejecución.

   Para el flujo que sigue la implementación de OpenCV se definieron 3 kernels para el proceso iterativo; primero un kernel igual a null, como resultado no se aplica ningún cambio en la imagen, efectivo para imágenes ideales; segundo un kernel de 3 filas y 3 columnas, los elementos de la columna 2 tienen como valor 1, los demás elementos tiene como valor 0, efectivo para imágenes con códigos de barras que presentan espacios en sus líneas negras; tercero un kernel de 7 filas y 7 columnas,

todos los elementos tienen como valor 1, efectivo para las imágenes con códigos de barras que presentan espacios en sus líneas negras y blancas.

3. La decodificación consiste en realizar un barrido por filas de la imagen, transformando cada fila en un vector de números enteros que representan el ancho de cada línea del código de barras, por medio de la moda estadística se normaliza el vector y se transforma en un vector con elementos de valor acorde a la Tabla 5.5, para así ser decodificado según las tablas 5.6 y 5.7.

   En cualquiera de los casos de decodificación de código de barras, Code-39 o Code-128, como primer paso se busca en el vector la primera coincidencia de las líneas que representan el carácter de inicio y se busca la última coincidencia de las líneas que representan el carácter de finalización, si la búsqueda no encuentra coincidencias, se invierte el vector y se repite la búsqueda de coincidencias, se realiza de esta forma con el fin de contemplar casos en donde el código de barras este girado 180^o, si después de invertir el vector la búsqueda de coincidencias falla se retorna valor nulo, en caso de ser exitosa la búsqueda, se acota el vector entre el inicio y fin del código de barras, recordar que pueden existir líneas externas al código de barras y son información innecesaria. Una vez acotado el vector se prosigue a realizar un barrido, tomando como razón de cambio la cantidad de dígitos que pueden representar un carácter o función, por cada iteración del barrido se busca en la tabla una llave que sea igual al valor tomado del vector, si la llave no existe se retorna con un valor nulo, si la llave existe se toma el valor respectivo a la llave, si el valor es un carácter se concatena a la variable que se retornara si la decodificación es exitosa y si el valor es una función (Caso presente en la decodificación de códigos de barras Code-128) se ejecuta la función y se continua la decodificación.

   El proceso de decodificación consta de dos pasos secuenciales, primero la decodificación para códigos de barras Code-39 y por ultimo la decodificación para códigos de barras Code-128.

5.4. Implementación del algoritmo

El algoritmo diseñado se implementó en dos paquetes diferentes, meja_scanner y opencv_scanner. El paquete meja_scanner implementa el algoritmo con operaciones de procesamiento digital de imágenes desarrolladas por el autor y el paquete opencv_scanner implementa el algoritmo con operaciones de procesamiento digital de imágenes disponibles en la librería OpenCV. Las dos implementaciones comparten el uso del paquete types que gestiona la lógica de decodificación de los códigos de barras (Expuesto en la página 48), en la Figura 5.21 se aprecia el diagrama de clases del paquete types, la clase abstracta BasicCodeType define el flujo general de la decodificación, y las clases Code39 y Code128 tienen la responsabilidad de decodificar el respectivo código de barras.

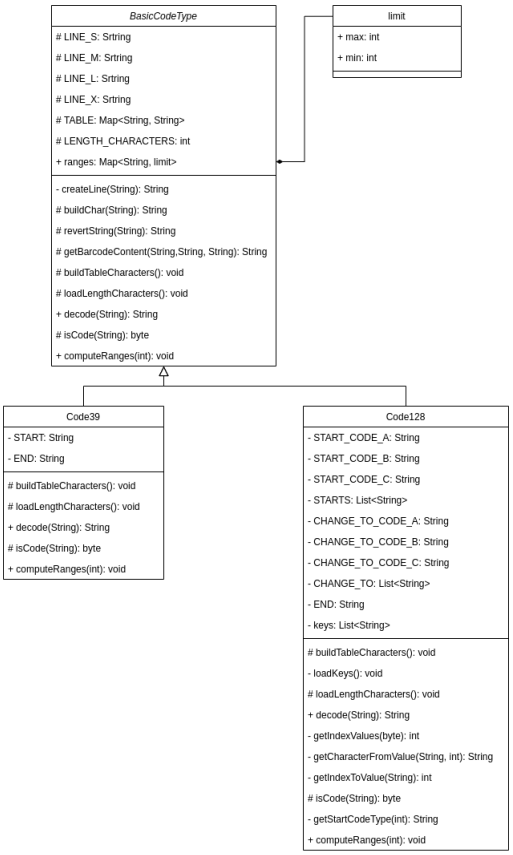

Figura 5.21: Diagrama de clases del paquete types
Fuente: Autor

5.4.1. Paquete meja_scanner

Con el objetivo de no depender de librerías externas para las operaciones de procesamiento digital de imágenes, se desarrolló el paquete meja_scanner.

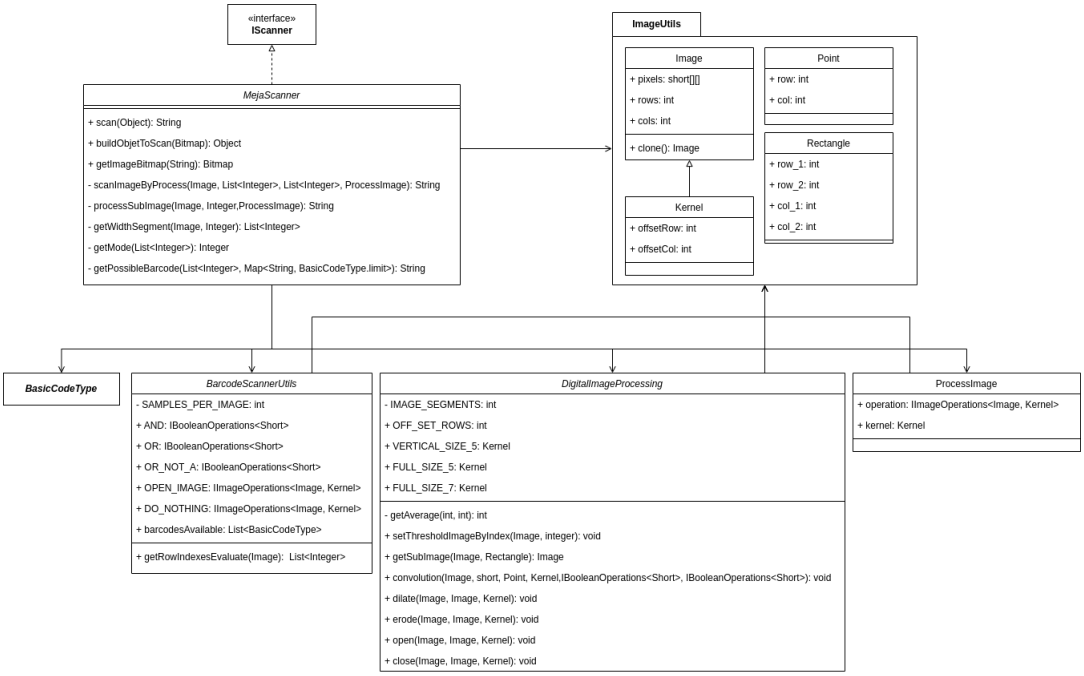

Figura 5.22: Diagrama de clases del paquete meja_scanner
Fuente: Autor

En la Figura 5.22 se aprecia el diagrama de clases del paquete meja_scanner, la clase MejaScanner implementa el algoritmo diseñado en sus tres partes:

1. Se aplica la umbralización de la imagen por promedio de región utilizando la clase DigitalImageProcessing.
2. Se procesa la imagen utilizando la clase ProcessImage, lo que requiere de una operación y un kernel, definidos en la clase BarcodeScannerUtils y en la clase DigitalImageProcessing, respectivamente.
3. Utilizando el principio de sustitución de Liskov se aplica la decodificación de la clase Code39 y Code128, por medio de la clase padre abstrata BasicCodeType.

5.4.2. Paquete opencv_scanner

Con el objetivo de delegar tareas de procesamiento digital de imágenes y gestión de memoria, se implementó OpenCV en el paquete opencv_scanner.

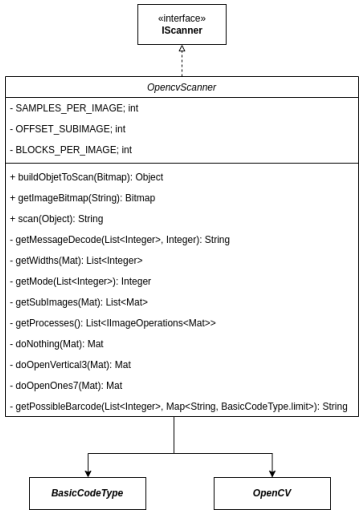

Figura 5.23: Diagrama de clases del paquete opencv_scanner
Fuente: Autor

En la Figura 5.23 se aprecia el diagrama de clases del paquete opencv_scanner, la clase OpencvScanner implementa el algoritmo diseñado en sus tres partes:

1. Se aplica la umbralización adaptativa con el método gaussiano, disponible en OpenCV.
2. Se procesa la imagen utilizando operaciones disponibles en OpenCV y con kernel definidos en la clase OpencvScanner.
3. Utilizando el principio de sustitución de Liskov se aplica la decodificación de la clase Code39 y Code128, por medio de la clase padre abstrata BasicCodeType.

5.4.3. Anotación

En las Figuras 5.22 y 5.23 se aprecia que las clases MejaScanner y OpencvScanner implementan la interfaz IScanner, esto se realizó con el fin de permitirle al desarrollador elegir entre las implementaciones de los paquetes meja_scanner y opencv_scanner.

5.5. Comparación entre librerías

Con el fin de comparar la librería desarrollada en este proyecto se seleccionaron entre las herramientas de desarrollo Android más relevantes para la lectura de códigos de barras las librerías ZXing [6] y ML Kit Barcode Scanning [27]. Se implementaron las librerías ZXing, ML Kit Barcode Scanning y la librería desarrollada por el autor, en diferentes aplicaciones Android. Las pruebas de comparación consisten en leer códigos de barras Code-39 y Code-128 y evaluar el tiempo de ejecución (Rendimiento) y la cantidad de lecturas acertadas (Fiabilidad).

El flujo consiste en cargar a memoria una imagen desde el almacenamiento del dispositivo Android, hacer uso del proceso de lectura de código de barras de la librería implementada con la imagen previamente cargada, almacenar la respuesta dada por la librería y calcular el tiempo que tarda el proceso de lectura, y finalmente se presenta al usuario, en un bloque de texto, los resultados, rendimiento y mensaje decodificado. Gráficamente en la Figura 5.24 se aprecia el diagrama de flujo planteado para las pruebas de comparación de las librerías.

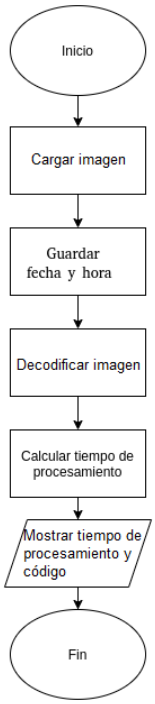

Figura 5.24: Diagrama de flujo para test de librerías de uso libre
Fuente: Autor

Las imágenes que se usan para cada una de las pruebas son: Figura 5.25a (Con 3000x4000 píxeles), Figura 5.25b (Con 2336x4160 píxeles), Figura 5.25c (Con 2336x4160 píxeles), Figura 5.25d (Con 2336x4160 píxeles) y Figura 5.25e (Con 352x140 píxeles).

(a) Carnet

(b) Equipo 1

(c) Equipo 2

(d) Equipo 3

(e) Código de barras

Figura 5.25: Imágenes de prueba
Fuente: Autor

5.5.1. Evaluación de ZXing

Utilizando Visual Studio 2019 como entorno de desarrollo integrado y con el lenguaje de programación C# se desarrolló una aplicación Android que implementa la librería ZXing.

Se realizó la instalación por medio del gestor de paquetes de Visual Studio 2019, NuGet [28] y se implementó el algoritmo de la Figura 5.24. La aplicación desarrollada se implementó en un dispositivo Android (Xiaomi Redmi Note 7) para realizar el correspondiente análisis.

Figura	Tiempo de respuesta promedio (Milisegundos - ms)
5.25a	800,90
5.25b	657,40
5.25c	693,60
5.25d	652,30
5.25e	4,30

Tabla 5.8: Tiempos de respuesta promedio de ZXing
Fuente: Autor

En la Tabla 5.8 se aprecian los tiempos de respuesta promedio de ZXing, a continuación se presentan los resultados detallados.

Al realizar la prueba para la Figura 5.25a se obtienen los siguientes datos de tiempo de respuesta y 0 % de fiabilidad, ver Tabla 5.9.

Prueba	Tiempo de respuesta (Milisegundos - ms)
1	800
2	801
3	799
4	799
5	801
6	811
7	800
8	798
9	801
10	799

Tabla 5.9: Resultados ZXing para Figura 5.25a
Fuente: Autor

Al realizar la prueba para la Figura 5.25b se obtienen los siguientes datos de tiempo de respuesta y 0 % de fiabilidad, ver Tabla 5.10.

Prueba	Tiempo de respuesta (Milisegundos - ms)
1	657
2	657
3	658
4	659
5	658
6	658
7	656
8	655
9	658
10	658

Tabla 5.10: Resultados ZXing para Figura 5.25b
Fuente: Autor

Al realizar la prueba para la Figura 5.25c se obtienen los siguientes datos de tiempo de respuesta y 0 % de fiabilidad, ver Tabla 5.11.

Prueba	Tiempo de respuesta (Milisegundos - ms)
1	693
2	694
3	694
4	695
5	693
6	693
7	695
8	692
9	694
10	693

Tabla 5.11: Resultados ZXing para Figura 5.25c
Fuente: Autor

Al realizar la prueba para la Figura 5.25d se obtienen los siguientes datos de tiempo de respuesta y 0 % de fiabilidad, ver Tabla 5.12.

Prueba	Tiempo de respuesta (Milisegundos - ms)
1	652
2	652
3	653
4	654
5	653
6	653
7	651
8	649
9	653
10	653

Tabla 5.12: Resultados ZXing para Figura 5.25d
Fuente: Autor

Al realizar la prueba para la Figura 5.25e se obtienen los siguientes datos de tiempo de respuesta y 100 % de fiabilidad, ver Tabla 5.13.

Prueba	Tiempo de respuesta (Milisegundos - ms)
1	4
2	5
3	4
4	5
5	4
6	6
7	4
8	3
9	4
10	4

Tabla 5.13: Resultados ZXing para Figura 5.25e
Fuente: Autor

5.5.2. ML Kit Barcode Scanning

Utilizando Android Studio como entorno de desarrollo integrado y con el lenguaje de programación Java se desarrolló una aplicación Android que implementa la librería de Google ML Kit Barcode Scanning. Se realizó la instalación por medio del sistema de automatización de construcción de código, Gradle [29] y se implementó el algoritmo de la

Figura 5.24. La aplicación desarrollada se implementó en un dispositivo Android (Xiaomi Redmi Note 7) para realizar el correspondiente análisis.

Figura	Tiempo de respuesta promedio (Milisegundos - ms)
5.25a	270,10
5.25b	160,40
5.25c	160,60
5.25d	295,30
5.25e	56,90

Tabla 5.14: Tiempos de respuesta promedio de ML Kit Barcode Scanning
Fuente: Autor

En la Tabla 5.14 se aprecian los tiempos de respuesta promedio de ML Kit Barcode Scanning, a continuación se presentan los resultados detallados.

Al realizar la prueba para la Figura 5.25a se obtienen los siguientes datos de tiempo de respuesta y 0 % de fiabilidad, ver Tabla 5.15

Prueba	Tiempo de respuesta (Milisegundos - ms)
1	270
2	256
3	294
4	259
5	251
6	255
7	265
8	255
9	258
10	338

Tabla 5.15: Resultados ML Kit Barcode Scanning para Figura 5.25a
Fuente: Autor

Al realizar la prueba para la Figura 5.25b se obtienen los siguientes datos de tiempo de respuesta y 100 % de fiabilidad, ver Tabla 5.16

Prueba	Tiempo de respuesta (Milisegundos - ms)
1	149
2	162
3	184
4	154
5	154
6	169
7	157
8	155
9	156
10	164

Tabla 5.16: Resultados ML Kit Barcode Scanning para Figura 5.25b
Fuente: Autor

Al realizar la prueba para la Figura 5.25c se obtienen los siguientes datos de tiempo de respuesta y 100 % de fiabilidad, ver Tabla 5.17

Prueba	Tiempo de respuesta (Milisegundos - ms)
1	142
2	210
3	137
4	123
5	161
6	292
7	139
8	132
9	136
10	134

Tabla 5.17: Resultados ML Kit Barcode Scanning para Figura 5.25c
Fuente: Autor

Al realizar la prueba para la Figura 5.25d se obtienen los siguientes datos de tiempo de respuesta y 100 % de fiabilidad, ver Tabla 5.18

Prueba	Tiempo de respuesta (Milisegundos - ms)
1	272
2	285
3	275
4	269
5	377
6	266
7	399
8	278
9	268
10	264

Tabla 5.18: Resultados ML Kit Barcode Scanning para Figura 5.25d
Fuente: Autor

Al realizar la prueba para la Figura 5.25e se obtienen los siguientes datos de tiempo de respuesta y 100 % de fiabilidad, ver Tabla 5.19

Prueba	Tiempo de respuesta (Milisegundos - ms)
1	55
2	67
3	53
4	54
5	59
6	57
7	51
8	60
9	53
10	60

Tabla 5.19: Resultados ML Kit Barcode Scanning para Figura 5.25e
Fuente: Autor

5.5.3. Librería desarrollada por el autor - meja_scanner

Utilizando Android Studio como entorno de desarrollo integrado y con el lenguaje de programación Java se desarrolló una aplicación Android que implementa la librería diseñada en este proyecto, usando el paquete meja_scanner. La aplicación desarrollada se implementó en un dispositivo Android (Xiaomi Redmi Note 7) para realizar el correspondiente análisis.

Figura	Tiempo de respuesta promedio (Milisegundos - ms)
5.25a	8893,70
5.25b	3160,00
5.25c	2891,10
5.25d	3082,90
5.25e	3,00

Tabla 5.20: Tiempos de respuesta promedio de la librería desarrollada por el autor - meja_scanner

Fuente: Autor

En la Tabla 5.20 se aprecian los tiempos de respuesta promedio de la librería desarrollada por el autor - meja_scanner, a continuación se presentan los resultados detallados.

Al realizar la prueba para la Figura 5.25a se obtienen los siguientes datos de tiempo de respuesta y 100 % de fiabilidad, ver Tabla 5.21

Prueba	Tiempo de respuesta (Milisegundos - ms)
1	8828
2	8911
3	8907
4	8935
5	8901
6	8901
7	8876
8	8895
9	8886
10	8897

Tabla 5.21: Resultados de meja_scanner para la Figura 5.25a
Fuente: Autor

Al realizar la prueba para la Figura 5.25b se obtienen los siguientes datos de tiempo de respuesta y 100 % de fiabilidad, ver Tabla 5.22

Prueba	Tiempo de respuesta (Milisegundos - ms)
1	3164
2	3170
3	3159
4	3177
5	3167
6	3148
7	3170
8	3148
9	3150
10	3147

Tabla 5.22: Resultados de meja_scanner para la Figura 5.25b
Fuente: Autor

Al realizar la prueba para la Figura 5.25c se obtienen los siguientes datos de tiempo de respuesta y 100 % de fiabilidad, ver Tabla 5.23

Prueba	Tiempo de respuesta (Milisegundos - ms)
1	2896
2	2894
3	2893
4	2875
5	2881
6	2893
7	2904
8	2889
9	2901
10	2885

Tabla 5.23: Resultados de meja_scanner para la Figura 5.25c
Fuente: Autor

Al realizar la prueba para la Figura 5.25d se obtienen los siguientes datos de tiempo de respuesta y 100 % de fiabilidad, ver Tabla 5.24

Prueba	Tiempo de respuesta (Milisegundos - ms)
1	3080
2	3090
3	3076
4	3084
5	3083
6	3087
7	3080
8	3078
9	3083
10	3088

Tabla 5.24: Resultados de meja_scanner para la Figura 5.25d
Fuente: Autor

Al realizar la prueba para la Figura 5.25e se obtienen los siguientes datos de tiempo de respuesta y 100 % de fiabilidad, ver Tabla 5.25

Prueba	Tiempo de respuesta (Milisegundos - ms)
1	3
2	4
3	2
4	3
5	3
6	3
7	4
8	3
9	3
10	2

Tabla 5.25: Resultados de meja_scanner para la Figura 5.25e
Fuente: Autor

5.5.4. Librería desarrollada por el autor - opencv_scanner

Utilizando Android Studio como entorno de desarrollo integrado y con el lenguaje de programación Java se desarrolló una aplicación Android que implementa la librería diseñada en este proyecto, usando el paquete opencv_scanner. La aplicación desarrollada se implementó en un dispositivo Android (Xiaomi Redmi Note 7) para realizar el correspondiente análisis.

Figura	Tiempo de respuesta promedio (Milisegundos - ms)
5.25a	553,10
5.25b	399,40
5.25c	360,60
5.25d	403,20
5.25e	59,40

Tabla 5.26: Tiempos de respuesta promedio de la librería desarrollada por el autor - opencv_scanner

Fuente: Autor

En la Tabla 5.26 se aprecian los tiempos de respuesta promedio de la librería desarrollada por el autor - opencv_scanner, a continuación se presentan los resultados detallados.

Al realizar la prueba para la Figura 5.25a se obtienen los siguientes datos de tiempo de respuesta y 100 % de fiabilidad, ver Tabla 5.27

Prueba	Tiempo de respuesta (Milisegundos - ms)
1	557
2	550
3	537
4	581
5	583
6	536
7	542
8	536
9	565
10	544

Tabla 5.27: Resultados de opencv_scanner para la Figura 5.25a

Fuente: Autor

Al realizar la prueba para la Figura 5.25b se obtienen los siguientes datos de tiempo de respuesta y 100 % de fiabilidad, ver Tabla 5.28

Prueba	Tiempo de respuesta (Milisegundos - ms)
1	422
2	387
3	390
4	380
5	459
6	401
7	399
8	436
9	401
10	419

Tabla 5.28: Resultados de opencv_scanner para la Figura 5.25b
Fuente: Autor

Al realizar la prueba para la Figura 5.25c se obtienen los siguientes datos de tiempo de respuesta y 100 % de fiabilidad, ver Tabla 5.29

Prueba	Tiempo de respuesta (Milisegundos - ms)
1	353
2	336
3	379
4	383
5	381
6	326
7	356
8	370
9	365
10	357

Tabla 5.29: Resultados de opencv_scanner para la Figura 5.25c
Fuente: Autor

Al realizar la prueba para la Figura 5.25d se obtienen los siguientes datos de tiempo de respuesta y 100 % de fiabilidad, ver Tabla 5.30

Prueba	Tiempo de respuesta (Milisegundos - ms)
1	404
2	396
3	414
4	410
5	398
6	395
7	382
8	419
9	412
10	402

Tabla 5.30: Resultados de opencv_scanner para la Figura 5.25d
Fuente: Autor

Al realizar la prueba para la Figura 5.25e se obtienen los siguientes datos de tiempo de respuesta y 100 % de fiabilidad, ver Tabla 5.31

Prueba	Tiempo de respuesta (Milisegundos - ms)
1	64
2	63
3	61
4	60
5	55
6	68
7	59
8	55
9	57
10	52

Tabla 5.31: Resultados de opencv_scanner para la Figura 5.25e
Fuente: Autor

5.6. Análisis de resultados

5.6.1. Análisis de ZXing

A continuación, se presentan los resultados promedio de las pruebas realizadas a ZXing, ver Tabla 5.32. Se determina que ZXing tiene una fiabilidad general del 20 %.

Figura	Tiempo de respuesta promedio (Milisegundos - ms)	Fiabilidad (%)
5.25a	800,9	0
5.25b	657,4	0
5.25c	693,6	0
5.25d	652,3	0
5.25e	4,3	100

Tabla 5.32: Resultados generales para ZXing
Fuente: Autor

Se determina que la lectura de código de barras de ZXing presenta las siguientes dificultades:

- Los píxeles que rodean la región donde se encuentra el código de barras afectan el proceso de decodificación
- Las líneas que conforman el código de barras no deben presentar irregularidades como variación de grosor (Efecto de ángulo visual en una imagen).
- Las líneas que conforman el código de barras no deben presentar irregularidades como el cambio de tono de color (Causa del desgaste físico del código de barras o iluminación).

5.6.2. Análisis de ML Kit Barcode Scanning

A continuación, se presentan los resultados promedio de las pruebas realizadas a ML Kit Barcode Scanning, ver Tabla 5.33. Se determina que ML Kit Barcode Scanning tiene una fiabilidad general del 80 %.

Figura	Tiempo de respuesta promedio (Milisegundos - ms)	Fiabilidad (%)
5.25a	270,1	0
5.25b	160,4	100
5.25c	160,6	100
5.25d	295,3	100
5.25e	56,9	100

Tabla 5.33: Resultados generales para ML Kit Barcode Scanning
Fuente: Autor

Se determina que la lectura de código de barras de ML Kit Barcode Scanning presenta las siguientes dificultades:

- Las imágenes que contengan un código de barras con solo un carácter codificado no será decodificado por la librería.
- Los píxeles que rodean la región del código de barras afectan el proceso de decodificación.
- No es adaptable con cambios progresivos de brillo en el código de barras afectando la decodificación.

5.6.3. Análisis de meja_scanner

A continuación, se presentan los resultados promedio de las pruebas realizadas a meja_scanner, ver Tabla 5.34. Se determina que meja_scanner tiene una fiabilidad general del 100 %.

Figura	Tiempo de respuesta promedio (Milisegundos - ms)	Fiabilidad (%)
5.25a	8893,7	100
5.25b	3160	100
5.25c	2891,1	100
5.25d	3082,9	100
5.25e	3	100

Tabla 5.34: Resultados generales de meja_scanner
Fuente: Autor

Se determina que la lectura de código de barras de meja_scanner presenta las siguientes dificultades:

- El tiempo que toma en procesar las imágenes no ideales (Fotos) es por lo menos 10 veces mayor que ZXing y ML Kit Barcode Scanning.
- No contempla casos donde la imagen contenga un código de barras en orientación vertical.

5.6.4. Análisis de opencv_scanner

A continuación, se presentan los resultados promedio de las pruebas realizadas a opencv_scanner, ver Tabla 5.35. Se determina que opencv_scanner tiene una fiabilidad general del 100 %.

Figura	Tiempo de respuesta promedio (Milisegundos - ms)	Fiabilidad (%)
5.25a	553,1	100
5.25b	399,4	100
5.25c	360,6	100
5.25d	403,2	100
5.25e	59,4	100

Tabla 5.35: Resultados generales de opencv_scanner
Fuente: Autor

Se determina que la lectura de código de barras de opencv_scanner presenta las siguientes dificultades:

- El tiempo que toma en procesar las imágenes no ideales (Fotos) es aproximadamente el doble del tiempo que tarda ML Kit Barcode Scanning.
- No contempla casos donde la imagen contenga un código de barras en orientación vertical.

CONCLUSIONES

- Dentro de los paquetes de la librería diseñada en este proyecto, se determina que el paquete opencv_scanner es una mejor solución que el paquete meja_scanner, para imágenes no ideales (Fotos), ya que presenta un tiempo de respuesta menor.
- Los resultados indican que la librería diseñada en este proyecto presentó una mayor fiabilidad que las librerías ZXing y ML Kit Barcode Scanning, por lo tanto, en criterios de fiabilidad es el mejor producto entre los evaluados, teniendo en cuenta que se diseñó específicamente para los códigos de barras Code-39 y Code-128.
- En criterios de rendimiento, el paquete meja_scanner de la librería desarrollada en este proyecto es una mejor solución que ZXing, ML Kit Barcode Scanning y opencv_scanner para lectura de códigos de barras en imágenes ideales.
- A partir del análisis de los datos obtenidos en cada una de las pruebas infiere que ML Kit Barcode Scanning es un producto más fiable que ZXing.
- El mayor reto para decodificar la información en una imagen que contiene un código de barras es definir las operaciones de procesamiento de imágenes necesarias para transformar aproximadamente la imagen a una imagen ideal, teniendo en cuenta el tiempo de procesamiento y la memoria requerida. En este caso la operación morfológica de apertura permitió normalizar las líneas que conforman el código de barras, tanto líneas negras como líneas blancas, además de filtrar el ruido generado por píxeles que generan regiones tipo isla que impiden la correcta decodificación de información en cada una de las imágenes.

- Las imágenes presentan diferentes características, como por ejemplo, el cambio de brillo, el ángulo de enfoque, cantidad de píxeles, entre otras. Con el fin normalizar la imagen y decodificar el código de barras, se debe aplicar un procesamiento de imágenes variando el número de iteraciones a ejecutar y el kernel con el que se opera.

RECOMENDACIONES

- En los últimos años se ha avanzado en el área de inteligencia artificial, se debe explorar más ese campo para generar una inteligencia que no requiera de mucha memoria y procesamiento para decodificar códigos de barras en imágenes, teniendo en cuenta los casos críticos, cambio de ancho y tono de color de las líneas que conforman el código de barras.
- Iterar el algoritmo de lectura de código de barras con diferentes estrategias de procesamiento para mejorar el tiempo de respuesta y refactorizar el código, con el propósito de tener una herramienta que con el paso del tiempo pueda ser escalable.
- Se recomienda el uso de imágenes con códigos de barras que ocupen por lo menos una décima parte de la imagen, ya que códigos de barras que sean muy pequeños no se leerán, adicionalmente se menciona que el código de barras debe estar horizontalmente para poder ser decodificado por el algoritmo.

ANEXOS

8.1. Repositorio del código fuente

La librería desarrollada con sus dos modos de operación y la aplicación Android que implementa dicha librería se puede consultar en:

- https://github.com/MiguelJ1/app_barcodescanner

8.2. Pruebas de la librería desarrollada

A continuación, se presentan los anexos de las pruebas realizadas a la aplicación que implementa la librería diseñada.

Figura 8.1: Prueba de la librería diseñada con la Figura 5.25a

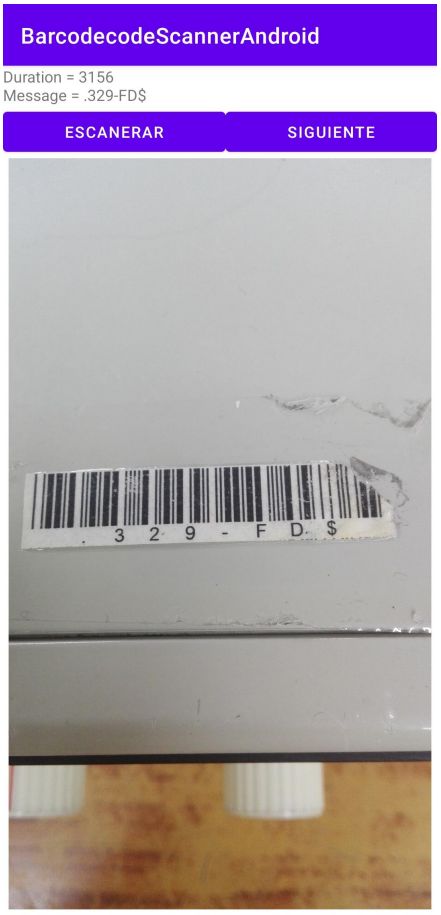

Figura 8.2: Prueba de la librería diseñada con la Figura 5.25b

Figura 8.3: Prueba de la librería diseñada con la Figura 5.25c

Figura 8.4: Prueba de la librería diseñada con la Figura 5.25d

Figura 8.5: Prueba de la librería diseñada con la Figura 5.25e

A continuación, se presentan los anexos de las pruebas realizadas a la aplicación que implementa la librería diseñada con OpenCv.

Figura 8.6: Prueba de la librería diseñada (Con Opencv) con la Figura 5.25a

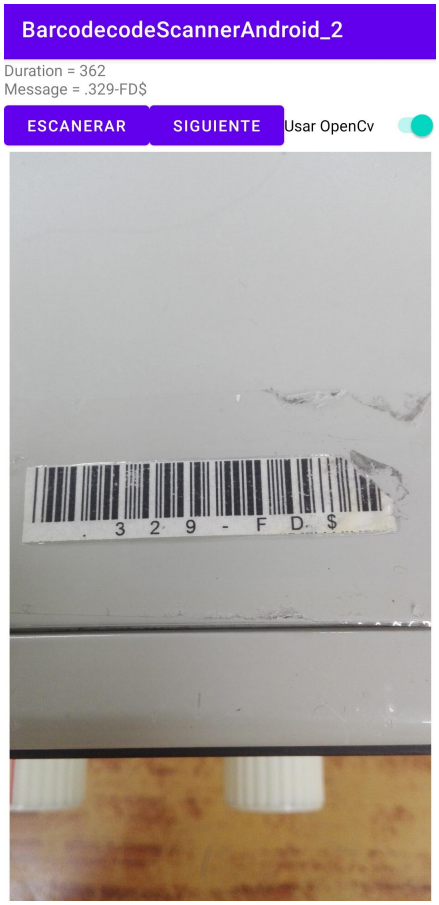

Figura 8.7: Prueba de la librería diseñada (Con Opencv) con la Figura 5.25b

Figura 8.8: Prueba de la librería diseñada (Con Opencv) con la Figura 5.25c

Figura 8.9: Prueba de la librería diseñada (Con Opencv) con la Figura 5.25d

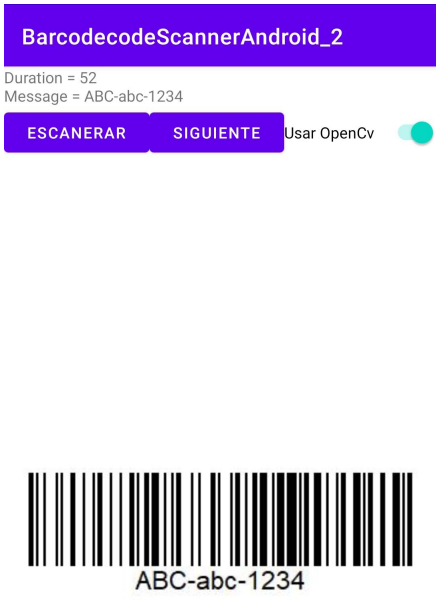

Figura 8.10: Prueba de la librería diseñada (Con Opencv) con la Figura 5.25e

8.3. Construcción de la librería usando Android Studio

El proceso de construcción de la librería consiste en exportar el package que contiene las clases desarrolladas a un librería de Android, para posteriormente compilar la librería de Android. Al tener la compilación de la librería de Android se debe agregar como dependencia al proyecto que la implemente. Android Studio permite realizar este proceso [30]. La librería Android se crea desde la opción, crear nuevo modulo, ver Figura 8.11. La compilación se realiza desde la opción, construir, ver Figura 8.12. La adición de las dependencias se gestiona desde la opción agregar dependencia JAR/AAR y se configura el directorio donde se almacena la librería, ver Figura 8.13.

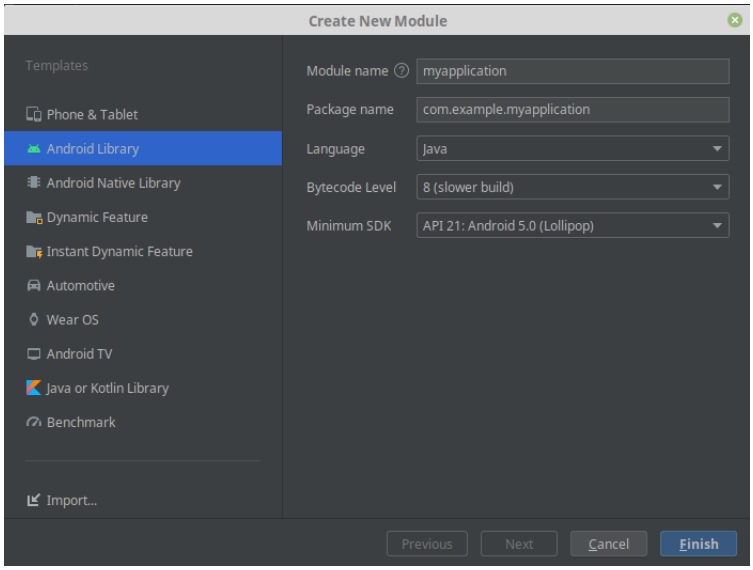

Figura 8.11: Android Studio, crear una librería de Android

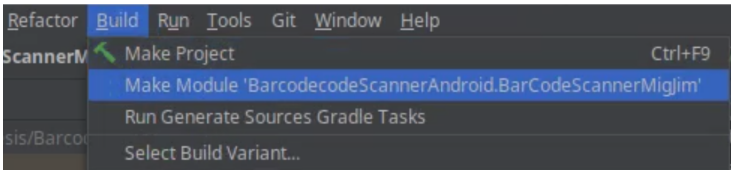

Figura 8.12: Android Studio, construir modulo

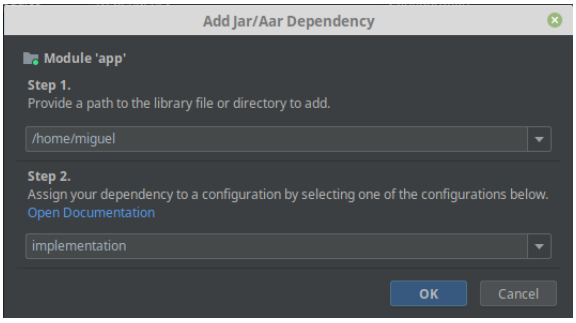

Figura 8.13: Android Studio, agregar dependencia

Como resultado de este proceso se obtiene un archivo *.aar*, este archivo permite importar clases, diseños, archivos de manifiesto y recursos de Android, en la Figura 8.14 se aprecia el contenido del archivo creado para este proyecto. Es importante notar que el archivo *.aar* contiene un archivo *.jar*.

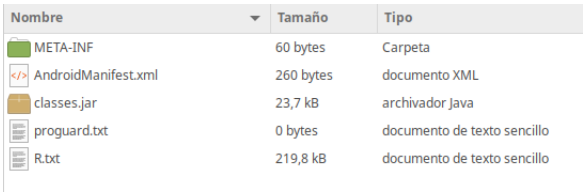

Figura 8.14: Contenido del archivo *.aar* de este proyecto

Bibliografía

[1] RFIDPOINT. *RFID vs. Código de Barras*. URL: http://www.rfidpoint.com/que-es-rfid/rfid-vs-codigo-de-barras/. (Último acceso: 15.08.2020).

[2] 3NSTAR. *SC050 Programming Manual*. URL: http://www.3nstar.com/wp-content/uploads/2016/12/SC050-Programming-Manual.pdf. (Último acceso: 15.08.2020).

[3] 3NSTAR. *SC305 SP*. URL: http://www.3nstar.com/wp-content/uploads/2016/11/SC305-SP.pdf. (Último acceso: 15.08.2020).

[4] NVIDIA. *cuda zone*. URL: https://developer.nvidia.com/cuda-zone. (Último acceso: 15.08.2020).

[5] OpenCV. *Descripción*. URL: https://opencv.org/about/. (Último acceso: 15.08.2020).

[6] S.Owen. *Repositorio*. URL: https://github.com/zxing/zxing/blob/master/README.md. (Último acceso: 15.08.2020).

[7] Vicerrectoría Académica General. *Catálogo de proyectos de investigación*. USTA, Bogotá, Colombia, p. 80, 2017.

[8] A. Correa Espinal, C. E. Álvarez López y R. A. Gómez Montoya. *Sistemas de identificación por radiofrecuencia, código de barras y su relación con la gestión de la cadena de suministro*. Estudios Gerenciales, vol. 26, no 116, 2010.

[9] GS1. *GS1 General Specifications*. vol. 19, Enero 2019.

[10] Rafael Gonzales y Richard Woods. *Digital Image Processing*. New Jersey: Prentice Hall, Second Edition, 1987.

[11] IES-SEP; Centre Integral de formació professional de la Garrotxa; MAI; INTERPLAST; INGenING; OMRON; LAZPIUR; IKUSPE; Generalitat de Catalunya Departament d'Educació; UEIS. *Aplicación práctica de la visión artificial en el control de procesos industriales*. Febrero 2012.

[12] Google. *UNIDAD 4 REDES NEURONALES*. URL: https://sites.google.com/site/mayinteligenciartificial/unidad-4-redes-neuronales. (Último acceso: 15.08.2020).

[13] S. Youssef y R. Salem. *Automated barcode recognition for smart identification and inspection automation*. Expert Systems with Applications, vol. 33, no 4, 2007.

[14] N. Guo y col. *GBVS Based 1D and 2D Barcodes Localization in Complex Scene*. Jabalpur, pp. 352-356, 2015.

[15] J. Brown. *ZBar bar code reader*. URL: http://zbar.sourceforge.net/index.html. (Último acceso: 15.08.2020).

[16] Learnopencv. *OpenCV QR Code Scanner (C++ and Python)*. URL: https://www.learnopencv.com/opencv-qr-code-scanner-c-and-python/. (Último acceso: 15.08.2020).

[17] Real Academia Española. *Imagen*. URL: https://dle.rae.es/imagen. (Último acceso: 15.08.2020).

[18] P. Fernando. *El espectro visible de luz*. URL: https://www.vix.com/es/btg/curiosidades/2011/10/02/el-espectro-visible-de-luz. (Último acceso: 20.02.2021).

[19] S. Johnson. *Stephen Johnson on Digital Photography*. O'Reilly Series. O'Reilly Media, Incorporated, 2006. ISBN: 9780596523701. URL: https://books.google.com.co/books?id=0UVRXzF91gcC.

[20] C. Dávila y A. Pardo. *Teoría de conjuntos. Conceptos, operaciones y propiedades*. FLACSO Mexico, 2016.

[21] OpenCV. *Morphological Transformations*. URL: https://docs.opencv.org/master/d9/d61/tutorial_py_morphological_ops.html. (Último acceso: 05.03.2021).

[22] C. Salazar y S. Castillo. *Fundamentos básicos de estadística*. Quito: sin editorial, 2017.

[23] Google. *Application Fundamentals*. URL: https://developer.android.com/guide/components/fundamentals. (Último acceso: 28.05.2021).

[24] Microsoft. *What is Xamarin?* URL: https://docs.microsoft.com/en-us/xamarin/get-started/what-is-xamarin. (Último acceso: 28.05.2021).

[25] keyence. *CODE 39 and Codabar*. URL: https://www.keyence.com/ss/products/auto_id/codereader/basic/code39.jsp. (Último acceso: 02.08.2021).

[26] keyence. *CODE 128 y GS1-128*. URL: https://www.keyence.com.mx/ss/products/auto_id/barcode_lecture/basic/code128/. (Último acceso: 02.08.2021).

[27] Google. *Barcode Scanning*. URL: https://developers.google.com/ml-kit/vision/barcode-scanning. (Último acceso: 28.05.2021).

[28] Microsoft. *An introduction to NuGet*. URL: https://docs.microsoft.com/en-us/nuget/what-is-nuget. (Último acceso: 28.05.2021).

[29] Gradle. *What is Gradle?* URL: https://docs.gradle.org/current/userguide/what_is_gradle.html. (Último acceso: 28.05.2021).

[30] Google. *Create an Android library*. URL: https://developer.android.com/studio/projects/android-library#groovy. (Último acceso: 02.08.2021).