Hochschule für Technik
und Wirtschaft Berlin

University of Applied Sciences

Faculty 1 - Energy and Information

Computer Engineering

Bachelor's Thesis

Magnetic Data Matrix Decoder

The Thesis is a part of an activity at MIP Technology GmbH.

Bachelor of Computer Engineering

Supervisors: Dr.-Ing. Piriya Taptimthong,

Prof. Dr. rer. nat. Sebastian Bauer

By: Mohamed Youssef Mhiri

Matriculation number: s0561969

August 27, 2023

Eigenständigkeitserklärung

Ich erkläre hiermit, dass

- Ich die vorliegende wissenschaftliche Arbeit selbständig und ohne unerlaubte Hilfe angefertigt habe,
- ich andere als die angegebenen Quellen und Hilfsmittel nicht benutzt habe,
- ich die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe,
- die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfbehörde vorgelegen hat.

Hannover, 27/06/2023 Mhiri, Mohamed Youssef

Contents

Chapter 1

Introduction

The need to identify objects and individuals is as old as human society itself. Since our ancestors first started trading goods, marking territory, and building communities, we have needed ways to distinguish one thing from another. Examining the history and evolution of identification technology reveals a fascinating story about human ingenuity and the continual drive for efficiency.

1.1 Identification

Perhaps the oldest product labels found by archeologists come from ancient Egypt, dating as far back as 1479 BC. Wine Jars found in the palace city of Amarna, the workmen's village of Deir el-Medina, and Tutankhamun's tomb in the Valley of the Kings were all labeled. With many of the labels "Reflecting the presence of wine as a beverage for festivals" [Wah12], this indicates that the presence of such jars and perhaps, such labels, far precedes these antiquities.[Wah12]

The art of identification evolved over the millennia. In the Near East, during the rule of Umar ibn alkhattab, the use of the Caliphate seal was no longer limited to official messages and treaties, but also made to include the authentication of bonds and goods.[Has11]

As the wheels of the Industrial Revolution began to turn, marking systems evolved as mass production demanded robust identification solutions. In 1798, Bavarian inventor

Alois Senefelder invented the printing process called lithography, which allowed for the mass production of labels.[Meg98]

Then, the tedious and lengthy process of attaching labels to goods with glue could finally be avoided as R. Stanton Avery invented pressure-sensitive self adhesive stickers in 1935.[Car05]

With production ramping up, and the ability to label goods no longer being a problem, a new bottleneck appeared, and there was demand for a way to automate the reading for labels.

As a response to this demand, Joseph Woodland and Bernard Silver filed a patent application for a "Classifying Apparatus and Method" in 1949. Heavily inspired by morse code, the two described a code that could be printed in both a linear and bull's eye printing patterns, as well as the mechanical and electronic systems needed to read it.[WS52] This invention eventually turned into the Universal Product Code in 1973[Nel97], which remains the most widely used optical identification technology even today.

But this was not the end of the line for the development of optical identification technology. As detailed identification was critical for many industries, a two-dimensional data matrix code was developed in order to pack even more data.[Den90]

However, optical technology was not the only path that engineers pursued. In 1973, the patent for a passive radio transponder with memory was granted to Mario Cardullo.[Car03] This was the first true ancestor of what would become known as Radio Frequency Identification (RFID) in 1983.[Wal83]

The RFID used electromagnetic fields to automatically identify and track tags attached to objects. Unlike Barcodes, it was not constrained by the need for line-of-sight reading. Later in 1996, the first patent for a batteryless RFID passive tag with limited interference was granted to David Everett, John Frech, Theodore Wright, and Kelly Rodriguez. [Eve+96]

1.2 Magnetic Storage

The concept of magnetic storage, an integral part of modern information technology, originated in the mind of Oberlin Smith, an American engineer. In 1878, Smith conceived the idea of recording telegraph signals on a wire with a magnetic needle, marking the inception of magnetic recording. However, his idea remained merely theoretical, as Smith did not pursue it further.[DMC99]

This idea was then taken forward by Valdemar Poulsen, a Danish engineer, who is credited for inventing a magnetic wire recorder, known as the 'Telegraphone'. Poulsen's Telegraphone was the first practical device for magnetic sound recording and reproduction.[DMC99]

Throughout the 20th century, the field of magnetic storage saw substantial progress. From the development of steel tape and wire recorders to the onset of digital video recording, the evolution of magnetic storage has been remarkable.[DMC99]

1.3 Magnetic Information Platform

In the ongoing exploration of identification technology, a new type of identification emerged from the CEOs of Magnetic Information Platform (MIP) Technology. It was researched in SFB 653 at Leibniz Universität Hannover,[DHL17] and then further developed to be suitable for use in industry with the founding of MIP Technology GmbH. In total, 14 years of research went into the development of this solution. [Tec]

Benefiting from the possibilities presented by magnetic storage, this technology aims at allowing for the automated identification of objects, particularly in environments where the use of RFID and optical codes would not be feasible or accurate.[Tec] With MIP, "the magnetically stored ID is temperature resistant up to 400 °C, and could be encapsulated in all non ferro-magnetic materials, such as stainless steel, brass, aluminum or polymers." [Tec]

The flexible data carrier can adapt to curved surfaces and, with a minimum thickness of just 0.3 mm, including its protective layer. Therefore, it could be seamlessly integrated

Figure 1.1: hidden MDMC beneath hologram security label

into thin-walled components. Despite this thinness, it is highly resistant to mechanical damage, more so than conventional identification technologies.[Tec]

The first iteration of this technology was the Magnetic ID tag, a one-dimensional barcode variant. As this iteration succeeded, the company decided to go further and develop the Magnetic Data Matrix code.

1.4 Magnetic Data Matrix Code

The next iteration is the Magnetic Data Matrix Code (MDMC). While both use magnetic encoding, the MDMC distinguishes itself by offering a two-dimensional magneto-optical code. It represents encoded information in a from of local magnetisation in a storage medium embedded in a polymer binder. The arrangement of the magnetization mimics a standard Data Matrix Code (DMC), this storage medium can be laminated and hidden under other thin cover layer. Such a feature makes the MDMC particularly usable as an anti-counterfeit measure, ensuring protection against plagiarism. [Tec] The Figure 1.1 shows a hologram security label placed on a the storage medium that contains the magnetic information.

Decoding of the magnetic ID tag is done using magnetic sensors. The sensors detect the magnetic fields from the tag, convert them to electrical signals which are sampled and converted to data by a microcontroller. However, unlike the magnetic ID tag, decoding the Magnetic Data Matrix code requires a different approach that uses the so-called magnetic

Figure 1.2: The magnetic Viewer used during this project

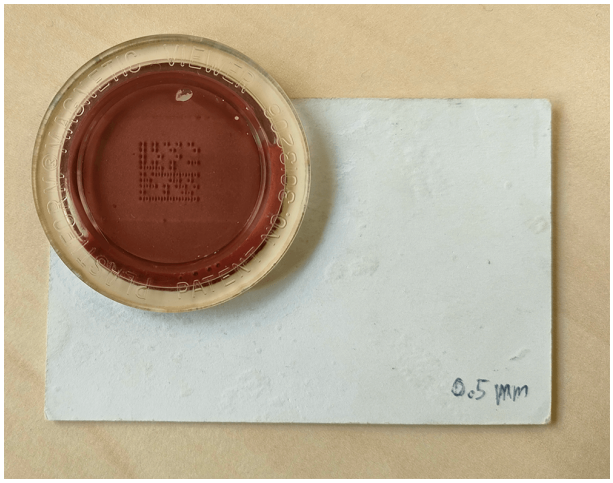


Figure 1.3: MDMC made visible through the magnetic viewer

viewer which consists of a ferromagnetic nano particles in a suspension.[Arn] Figure 1.2 shows the magnetic Viewer used in this project.

The MDMC is visualized utilizing the specified magnetic viewer as illustrated in Figure 1.3. The subsequent decoding of the revealed information captured from the magnetic viewer relies on the power of image processing techniques, which will be elaborated in Chapter 6. The decoding encompasses the transformation of the MDMC into a conventional DMC. Thereafter, the transformed image is decoded by applying established data matrix decoding methods.

Chapter 2

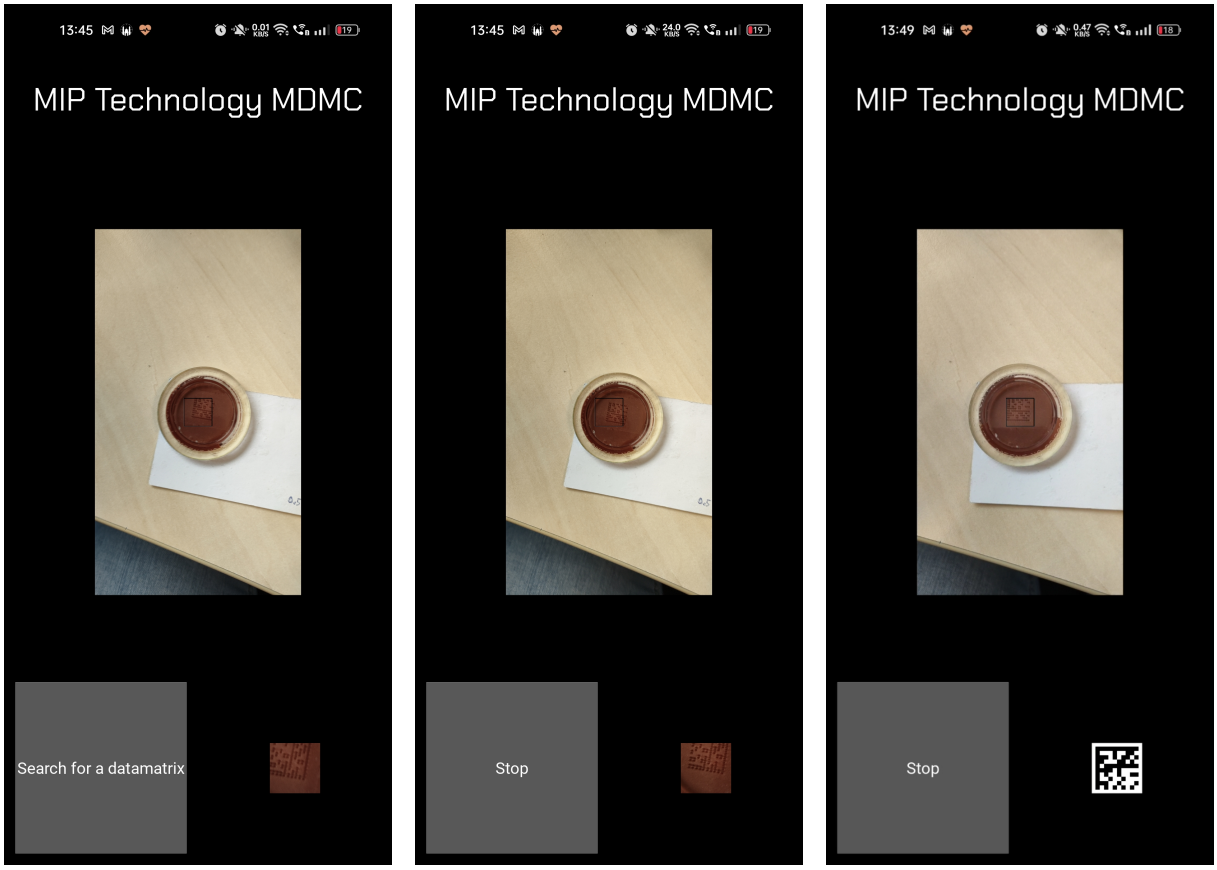
Analysis

In an effort to showcase the potential of the MDMC technology, this project focuses on the development of an Android application that is capable of promptly analyzing and decoding MDMCs from images captured by a mobile android device's camera. This application aims to provide evidence of the MDMC's utility and versatility as an identification solution.

2.1 Existing Software

The development of the Magnetic Data Matrix Decoder leveraged an existing software, which was the product of my internship. This software was fully written in python with a user interface created using the Kivy library and packaged by the Buildozer tool.

The existing software presents two camera views, both showing the captured frame. As illustrated in the Figure 2.1, the small view on the right bottom corner applies a real-time image processing algorithm capable of transforming an MDMC into a standard DMC. When this conversion is successful, the resultant data matrix would substitute the initially captured image in that camera view. The image processing algorithm used in this software will serve as material for the development of the new application.



(a) the idle state of the application

(b) Processing images until a magnetic data matrix is successfully processed

(c) After processing a frame the data matrix is displayed in the small camera view

Figure 2.1: Different states of the existing software

2.1.1 Analysis of Limitations

The existing application, built with Python and Kivy, faced several significant limitations:

- The application's performance was suboptimal, suffering from noticeable lag during the scanning leading to an inferior user experience.
- The image processing element of the application often generated incorrect data matrices, especially under different lighting conditions. This inconsistency in performance hindered the reliability of the application.
- The existing application lacked a data matrix decoder. While capable of converting the MDMC into a standard DMC, the application was not equipped to decode the information embedded within this image. This missing functionality is something that would naturally synergize with the existing application, and its omission significantly limits its usefulness.
- The existing application lacked a history view, a feature that would record the scanned MDMCs. This omission limited the application's ability to keep track of previous scans, which is an important function for users needing to recall previously scanned data.

2.2 Requirements and Use Case

Upon analysis, the existing software exhibited several significant limitations that need to be addressed in the development of the new MDMC decoder. The upcoming stages of this project will be dedicated to overcoming these challenges to deliver an application that is both responsive and accurate, providing an enhanced user experience while properly carrying out its functional requirements.

2.2.1 Functional Requirements

The following list outlines the essential functions required from the application:

1. On its first launch, the application must prompt the user for permission to use the camera.
2. The application must present the user with a camera view, highlighting the Region Of Interest (ROI) for capturing images.
3. The application must capture images in real-time and analyses them.
4. The application must process these images into a decodable DMC.
5. The application must display the successfully decoded DMC and as well as the decoding result.
6. The application must store the decoded DMCs in a local database (optional).
7. The application must provide navigation buttons enabling users to move between different screens within it (optional).

2.2.2 Technical Specifications

The following technical specifications are incorporated in the application to fulfill the functional requirements mentioned above.

1. Android Studio is the chosen Integrated Development Environment (IDE) for building the application.
2. The application is developed using Kotlin, specifically leveraging the Android Jetpack libraries.
3. The application uses the Android CameraX API for real-time capturing of images (part of the Android Jetpack suite).
4. Image processing is carried out using a dedicated algorithm implemented in C++ and utilizing the OpenCV library.

5. The ZXing (Zebra Crossing) library is used for decoding the DMCs.
6. The decoded data is displayed on the application's User Interface (UI).
7. The application's local database is built with the room library to store the decoded DMCs with a time snapshot(optional).
8. The DMCs is created using Android Studio's built-in user interface design tools.
9. The application follows a comprehensive design pattern (Model View View-Model (MVVM)) for separation of concerns, organization, testability, and maintainable code structure.
10. The application targets Android API levels from 27 to 33, ensuring compatibility with Android versions from Android 8.1 (Oreo) to Android 13 (Tiramisu).
11. The application must ensure a uniform behavior and experience across various Android devices and platforms, taking into consideration differences in screen sizes, resolutions, hardware specifications, and Android versions.

Each technology used is discussed in Chapter 3, while their implementation is detailed in Chapter 5.

2.2.3 Use Case: Magnetic Data Matrix Code Decoder

The use case for the MDMC Decoder outlines the Happy Path scenario, detailing the sequential interaction between the user and the application. The following list elaborates on these steps and associated conditions.

1. The use case begins when the user launches the application for the first time.
2. The application prompts the user to grant permission to use the device's camera.
    2.1. If the user denies the permission, the use case ends.
    2.2. If the user grants the permission, the use case proceeds to the next step.
3. The application presents the user with a camera view, highlighting the ROI for capturing images.

4. The user positions the camera to capture an image of an MDMC.
5. The application captures images in real-time and processes them to create a decodable DMC.
6. The application decodes the DMC.
   6.1. If the decoding fails, the application goes back to step 3.
   6.2. If the decoding is successful, the use case proceeds to the next step.
7. The application displays the decoded data matrix code.
8. The application saves the decoded DMC in a local database (optional).
9. The application provides navigation buttons, allowing users to move between different screens within the application, including the database screen (optional).
10. The use case ends successfully when the user navigates away from the application or closes the application.

Chapter 3

Android Application Fundamentals

Android is a mobile operating system based on a modified version of Linux. It started with a company called Android, Inc. In 2005, Google bought this company and its team, continuing the development of Android. Google released most of the Android code under the open-source Apache License, allowing anyone to download and use it [DiM16].

This open-source approach lets vendors, like hardware manufacturers, customize Android with their own additions. After the launch of Apple's iPhone, many phone makers saw Android as a way to compete. Companies like Motorola and Sony Ericsson, which previously had their own mobile systems, started using Android [DiM16].

Today, Android has a hold of over 70% of the mobile operating system market share worldwide [Sta23]. This means that developing apps for Android means that it's likely that they would be compatible with most mobile systems on the market.

3.1 Software Development Kit

The Android SDK (Software Development Kit) is a comprehensive set of software tools and resources released concurrently with Android Studio. They are designed for building applications on the Android platform. It provides developers with libraries, APIs, and debugging tools necessary for creating, testing, and optimizing Android apps [tea23].

3.2 Android Jetpack

In 2018, Google introduced Android Jetpack to the developer community. Designed to facilitate the faster and simpler development of modern, reliable Android apps, Jetpack comprises a suite of tools, libraries, and architectural guidelines. The primary elements of Android Jetpack include the Android Studio IDE, the Android Architecture components, and the modern App architecture guidelines [Smy18].

Given the comprehensive nature of Android Jetpack, which Google endorses as a foundational toolset for Android development, this project incorporates Jetpack's tools to align with modern Android application best practices. While the MDMC application targets Android API levels from 27 (Android 8.1, Oreo) to 34 (Android 13), it's important to note that testing has predominantly been conducted on two Android 13 smartphones and an Android 12 tablet.

3.3 Android Activity Lifecycle

Android activities are tailored for the unique demands of mobile devices, taking into account limited resources such as memory and battery. The Android Activity Lifecycle is a mechanism that delineates the sequence of events an activity undergoes from its creation to its termination. As depicted in Figure 3.1, the Android Activity Lifecycle comprises several events, which are explained as follows:

- **onCreate()**: Triggered when an activity is instantiated. This is typically where views are created, data files are accessed, and the activity is initialized.
- **onStart()**: Executed before the activity becomes visible. Depending on whether the activity can be foregrounded, control may transfer to onResume() or onStop().
- **onResume()**: Activated post-onStart if the activity is foregrounded. Here, the activity interacts with the user and resumes tasks like UI updates.
- **onPause()**: Initiated when another activity is about to take the foreground. Activities should conserve resources during this phase.

- **onStop()**: Initiated when the activity becomes invisible or is about to be terminated.
- **onRestart()**: Called when an activity is about to be restarted after being stopped. Typically followed by **onStart()**.
- **onDestroy()**: The last stage where any final processing occurs before the activity is terminated [Rog+09].

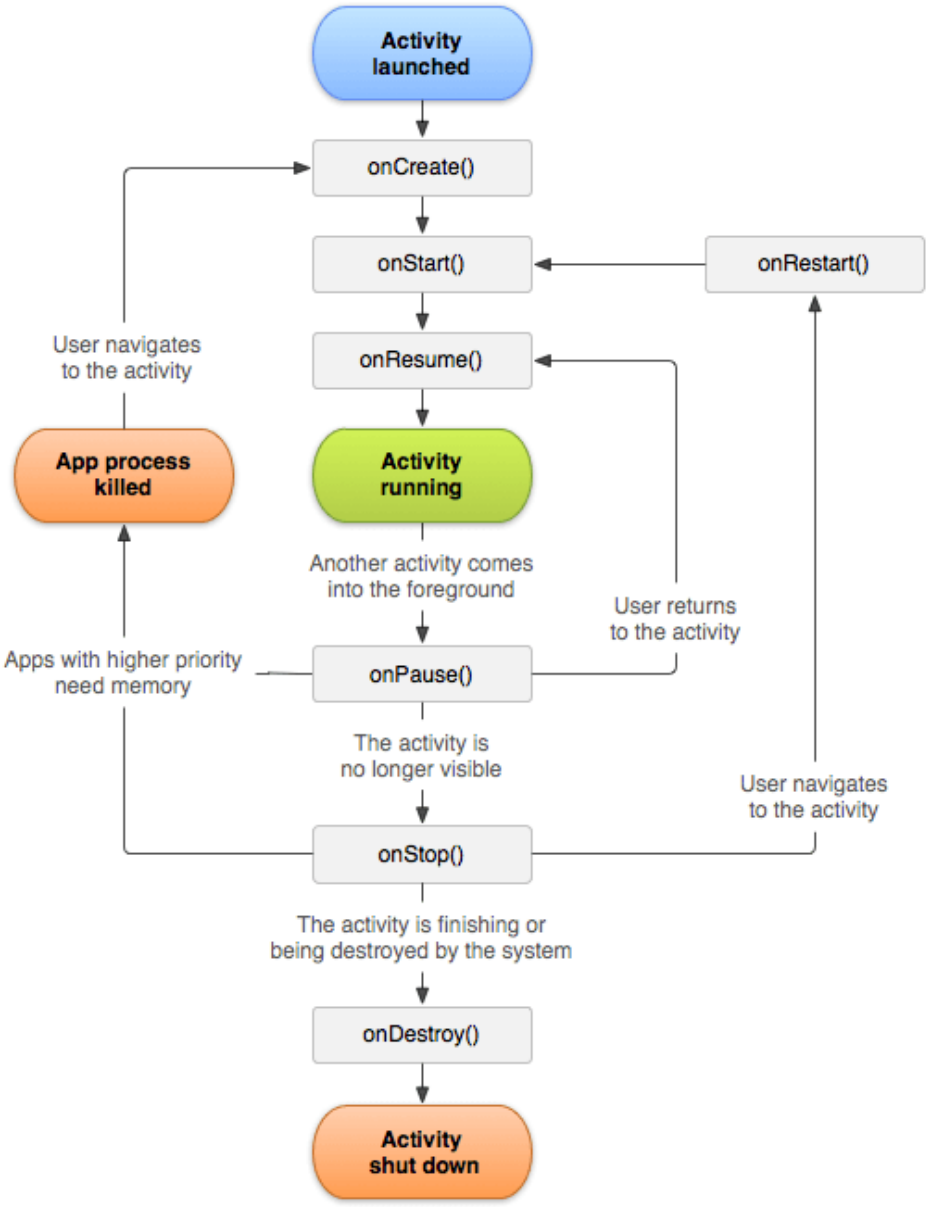


Figure 3.1: A simplified illustration of the activity lifecycle from the main Android studio documentation website https://developer.android.com/

User interactions and their corresponding effects on the Android activity lifecycle:

1. **Launching an Activity:**
    a) `onCreate()`
    b) `onStart()`
    c) `onResume()`
2. **User presses the ”Back” button:**
    a) `onPause()`
    b) `onStop()`
    c) `onDestroy()`
3. **User receives a phone call (assuming your app doesn’t handle phone calls):**
    a) `onPause()`
    b) `onStop()`
4. **User goes back to your app after the call:**
    a) `onRestart()`
    b) `onStart()`
    c) `onResume()`
5. **User presses the ”Home” button or recent apps button:**
    a) `onPause()`
    b) `onStop()`
6. **User navigates back to your app via app switcher:**
    a) `onRestart()`

b) `onStart()`

c) `onResume()`

7. **Another app comes in front of yours (like an incoming call UI or a dialog):**

   a) `onPause()`. If the user interacts with the other app and your app becomes fully hidden:

   b) `onStop()`

8. **User interacts with a dialog or pop-up inside your app:**

   a) No lifecycle methods called as activity is still in the foreground.

Taking activity lifecycles into account ensures optimized mobile application performance and an enhanced user experience. Developing with these lifecycle stages in mind differentiates mobile programming from traditional desktop programming [Rog+09].

In the implementation phase (Chapter 5) of the Android component of this project, adhering to these activity lifecycles is paramount. Properly managing each lifecycle stage will ensure the application's robustness, user-friendliness and also ensures resource-efficient operation.

One of the tools aiding in this approach is *LiveData*, which is designed to notify views when underlying data is created or updated, ensuring UI components are always up-to-date [Tri20]. As a lifecycle-aware data holder class, LiveData guarantees that UI components observe changes only when they're in an active lifecycle state, minimizing potential memory leaks and promoting efficient app operation.

3.4 Asynchronous Programming

Typically, a program operates in a sequential manner, completing one task before initiating the next. This approach, called Synchronous Programming, works well for straightforward tasks [Tig22].

Yet, there are operations that are time-intensive, such as:

- Retrieving or storing data in a database
- Interacting with or updating data on a network
- Handling text, images, videos, or other file types
- Complex calculations [Tig22]

During such operations, the application might appear stagnant and non-responsive to users. Users remain unable to interact with the application until these tasks conclude [Tig22].

Asynchronous programming offers a solution to this challenge. This method enables tasks to operate separately from the main application thread and operate continuously on a background thread, preventing the application from becoming unresponsive. This ensures users can continue interacting with the application or its interface even as the original task is running. Once the task concludes or encounters an error, users can then be notified on the main thread [Tig22].

In the context of this project, a three-threaded structure is used. One thread is responsible for capturing photos, a second thread for analyzing them and a thread for saving the successfully decoded MDMCs.

Kotlin Coroutines: Kotlin coroutines are a library designed for handling background operations such as network calls or file access. They are Google's recommended approach for asynchronous programming in Android. These coroutines are integrated with Android Jetpack libraries and are supported by other Android libraries such as Retrofit. With coroutines, code can be written sequentially. They use "suspend" functions, allowing tasks to pause without halting the entire thread, and resume once ready, improving code readability and maintenance [Tig22].

3.5 Design Pattern

The MVVM pattern, introduced by John Gossman, a Software Architect at Microsoft in 2005, is a specialization of the Presentation Model pattern, which was proposed by Martin

Fowler in 2004 [Gar11].

The structure of an MVVM application primarily comprises three major components: the Model, the View, and the ViewModel. The Model is the entity representing the business concept, ranging from a simple customer entity to a complex stock trade entity. The View is the graphical control or set of controls responsible for rendering the Model data on screen, be it a WPF window, a Silverlight page, or just an XAML data template control. The ViewModel, on the other hand, plays the most crucial role. It contains the UI logic, commands, events, and a reference to the Model [Gar11].

The application adheres to this pattern, with the model encapsulating the image analysis and processing logic and the database component, the View encapsulating the UI and user-interaction related functions, the ViewModel serving as an intermediary managing the interaction between the View and the Model. This separation of concerns allows for easier maintainability and testability and provides cleaner code.

3.6 Database

In this application, the utilization of a database is essential for the persistent storage of records. Such a structure ensures that information is maintained across sessions, providing a stable and efficient method for managing data.

3.6.1 SQLite

SQLite is an open-source relational database, similar to systems like MySQL or PostgreSQL. What sets SQLite apart is that it stores data in straightforward files, rather than a complex server setup. These files can be read and written using the SQLite library. Android has included this SQLite library in its standard offerings, supplementing it with Java helper classes to make it more accessible [Phi+15].

3.6.2 Room Architecture Component Library

Dealing with SQLite for database interaction within Android applications used to be an unpleasant task. Constructing a database required significant effort, there was no mechanism for query validation, and the conversion of data from the Cursor class into Java or Kotlin objects had to be done manually. Fortunately, the advent of Jetpack introduced Room, a persistence library built upon SQLite. This addition has simplified the database handling while retaining all the previous functionalities but with more ease [Faz21].

With Room, developers can define the database structure and the methods to access it using special labels called annotations within the Kotlin programming language [Dev23].

Room consists of three main parts:

1. **Application Programming Interface(API)**: The building blocks used to construct the database.
2. **Annotations**: Special notes in the code that instruct Room on data organization, storage, and access.
3. **Compiler**: The component that reads the annotations and API, assembling the database [Dev23].

Room provides a user-friendly way to manage data in Android apps, abstracting the complexities of working directly with SQLite. Three major components are present in a Room database(Figure 3.2):

1. **Database class**: Holds the database and serves as the main access point to the app's persisted data.
2. **Data entities**: Represent tables in the app's database.
3. **DAO**: Provide methods for querying, updating, inserting, and deleting data [Dev23].

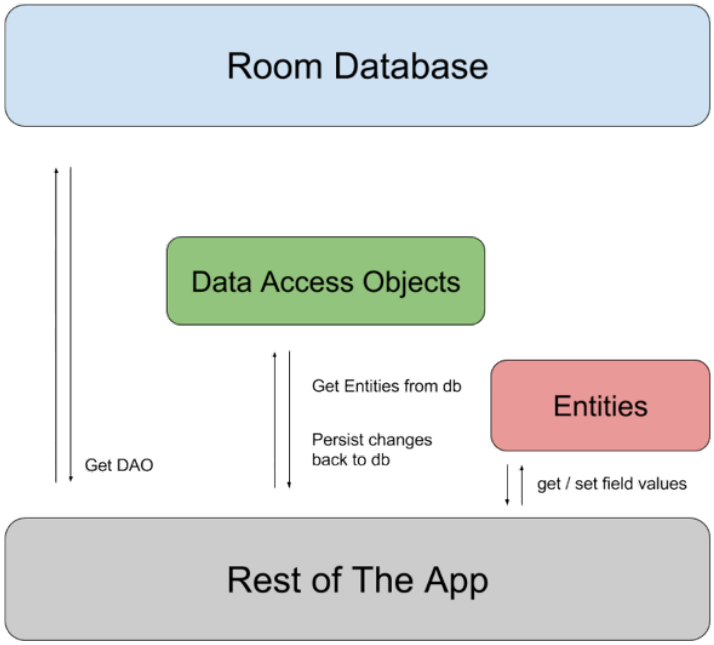


Figure 3.2: Diagram of Room library architecture from the main Android studio documentation website https://developer.android.com/

3.6.3 Repository

A repository is not a particular class related to Room but rather a convention recommended by Jetpack. Utilizing a repository provides a unified location for data access for ViewModel classes, irrespective of the data's source. Whether the data comes from web service calls or database interactions, the front-end components can access it from the same location without concern for its origin. Having a repository separates out how you get data with how you use the data[Faz21].

3.7 Recycler Views

The need to display a list with a variable number of items is frequent, especially in corporate settings. Focus here is on the high-performing recycler views. With Kotlin's conciseness, implementing a recycler view occurs in an elegant manner [Spä18].

Basic concept involves having a data collection, possibly from a database. This collection is sent to a single UI element responsible for presentation, including rendering all visible items and providing a scroll facility if needed. Presentation of each item can depend on an XML layout file or be generated dynamically within the code[Spä18]. This particular view will be used to display the items of the database.

3.8 Java Native Interface and Android NDK

As the image processing algorithm is implemented in C++, the utilization of the NDK becomes essential for seamless integration and execution within the Android environment.

The NDK allows developers to write code parts in native languages like C, C++, and assembly, alongside the usual Java code. While Java code runs on the Dalvik Virtual Machine (VM), the native code gets compiled to binaries that run directly on the Android system, which is fundamentally Linux-based. Bridging these two environments is the Java Native Interface (JNI), acting as an interface for both realms to interact seamlessly [Liu13].

JNI offers both primitive and reference data types, each having corresponding mapping data types in Java. Direct manipulations are often feasible with primitive types, given their equivalence to native C/C++ data types. Conversely, reference data manipulations usually necessitate pre-defined JNI functions. Through JNI, one can invoke native methods from Java, access Java fields, and call Java methods from native code. Effective JNI programming also covers topics like data caching for enhanced performance, error and exception handling, and the integration of assembly in native method implementations [Liu13].

Loading native code typically involves compiling it into a shared library. This library must be loaded before one can call any native methods [Liu13]. The interaction between different application component and the Android system is illustrated in Figure 3.3.

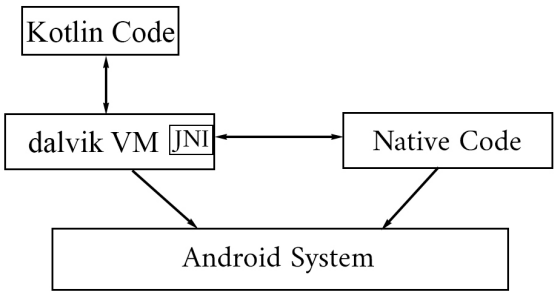


Figure 3.3: Interaction flow between the application components and the Android system

3.9 Gradle

In Android projects, the Gradle build file functions as a script for managing various build-related tasks. The build file exists at two levels: project-level and module-level. The former manages configurations applicable to the entire project, while the latter is responsible for individual modules within the app.

Key components in a Gradle build file include:

- **Dependencies**: Specifies required libraries and SDKs.
- **Plugins**: Adds extra functionalities, such as code assessments or unit tests.
- **Android Version**: Defines the minimum and target SDK versions the app should use.
- **Build Variants**: Sets configurations for different app versions, e.g., debug and release.
- **Compile Settings**: Lays down the rules for resource compilation and source code.

The Gradle file aids in the automated process of resource bundling, code compilation, and APK file generation for deployment on Android devices. [tea23].

Chapter 4

Image Processing Fundamentals

Image processing is the field that focuses on the transformation or alteration of images using various techniques. It is commonly used to enhance image quality or extract information and details. This chapter aims to provide a detailed discussion on techniques such as pixel manipulation, color spaces, and image transformations.

4.1 Digital Images

In the context of computer vision, a digital image is essentially a matrix, where rows and columns correspond to its dimensions. This matrix is composed of individual cells known as pixels, and each pixel holds a value or a vector of values that represent its intensity or color [SS00].

In the case of a colored image, each pixel is represented not by a single value, but by a vector of three elements. This vector represents the intensities of Red, Green, and Blue components in the case of RGB color space, or Blue, Green, and Red in the case of BGR color space [SS00] Figure 4.1 illustrates this concept, showing a small 3x3 pixel section. Within this section, two distinct RGB vectors can be observed: one for a shade of green (R:139, G:199, B:171) and another for white (R:255, G:255, B:255).

In this application, color information is processed in 8-bit RGB or BGR color spaces. RGB is commonly the format captured by Android cameras, while BGR serves as the standard format in OpenCV.

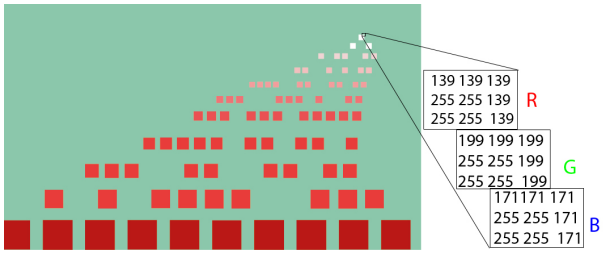


Figure 4.1: Pixel value representation in a segment of the MIP logo.

Each element in the color vector can assume a value ranging from 0 (indicating the absence of the color) to 255 (indicating the maximum intensity of the color). This allows each pixel to have a wide palette of possible colors, with the three components combining to represent the final color.color.[SS00]

Given the matrix and vector representation of an image, mathematical techniques can be employed for image manipulation and analysis. For instance, the average intensity of pixels in a monochrome image can be calculated to determine the brightness level of the image. A bright image corresponds to a high average intensity, whereas a dark image corresponds to a low average intensity. Such insights facilitate the adjustment of the image brightness.

Furthermore, the matrix representation enables the identification and manipulation of specific image sections. Utilizing the image matrix coordinates, precise regions can be cropped, which is especially beneficial when particular features, such as edges, are detected in an image.

Moreover, another essential manipulation technique in image processing is rotation. Rotation of an image involves changing the position of its pixels around a specified point, typically the center, by a defined angle. Mathematically, this is achieved using rotation matrices. In a 2D space, for an image represented as a matrix, the rotation transformation can be expressed as:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \tag{4.1}$$

Where (x', y') are the new coordinates of a pixel after rotation by an angle θ, and (x, y)

are its original coordinates. Such transformations allow for the alignment of images, correction of orientations, and can be pivotal when preparing images for further processing or analysis.

In the context of this work, rotations will be instrumental in ensuring the images are oriented correctly before applying more advanced processing techniques [Str05].

This project will employ various matrix manipulation techniques, harnessing the functionalities provided by the OpenCV library, as elaborated in the subsequent subsection.

4.2 OpenCV Library

OpenCV, which stands for Open Source Computer Vision, is an open-source library dedicated to computer vision and artificial intelligence. It was launched in 1999 by Gary Bradski at Intel Corporation with the aim of providing a robust, comprehensive infrastructure for everyone working in these fields. The library is written primarily in C and C++, and it provides interfaces for various programming languages, including Python, Java, and MATLAB.[KB16]

OpenCV can be built for various platforms including Android, which makes it particularly relevant for our application. It's designed to be user-friendly, facilitating the construction of complex vision applications.[KB16]

One of the key attributes of OpenCV is that it houses over 500 functions spanning various areas in vision. For our purposes, it provides the necessary functionality to convert a MDMC to a standard DMC.[KB16]

Notably, OpenCV was the brainchild of Gary Bradski, who is also the co-author of the book "Learning OpenCV 3: Computer Vision in C++ with the OpenCV Library" which will be a principal reference in the development of the application.

4.3 Image Processing Methods

This section outlines key algorithms and techniques used for manipulating and analyzing images. Familiarity with these methods is required to develop the algorithm for the MDMC to DMC conversion.

- `Color Space Conversion`: Image color spaces, such as RGB (Red, Green, Blue) and BGR (Blue, Green, Red), represent the range of colors contained in an image. Depending on the application, images may need to be converted from one color space to another. For instance, while RGB is commonly used in image displays, BGR is typically utilized with OpenCV functions. The OpenCV function `cv::cvtColor` is used for these conversions. Additionally, images may be converted into grayscale, which simplifies image data by focusing solely on intensity values and removing chromatic information. This is especially important for thresholding and other techniques where color information is not essential.[KB16]

- **Canny Edge Detection** Edge detection is used most frequently for segmenting images based on abrupt changes in intensity. [GW07] Canny edge detection is a technique used to extract useful structural information from an input image by identifying the edges of an object, thus converting it into an "edge image" which is a Boolean format(black and white). An example of an edge image is illustrated in Chapter 6 under Section 6.4

- **Hough Line Transform**

  The Hough transform is a technique used to detect simple shapes in an image. Initially developed for line detection, the method has evolved to identify other shapes too. In essence, the Hough line transform theorizes that any given point in a binary image can belong to various possible lines. When we convert every point in the input image using this theory, lines in the original image will manifest as prominent points in the output, known as the accumulator plane.[KB16]

  However, instead of the familiar slope-intercept representation of lines, the transform uses a more efficient polar coordinate system, (ρ, θ), where a line can be defined by:

$$\rho = x \cos\theta + y \sin\theta$$

While OpenCV offers several Hough line transform variants, for the purposes of this project, the following is of particular interest:

Progressive Probabilistic Hough Transform (PPHT): This algorithm not only determines line orientation but also its extent. It's termed 'probabilistic' because it doesn't analyze every possible point, but a subset. This approach can lead to significant computational savings and is the primary method adopted in our project.[KB16]

- **Contours Detection** Contours in images play a vital role in distinguishing and characterizing different segments. These are essentially lists of points that depict curves in an image and are typically represented in OpenCV using STL-style vector¡¿ template objects. The function cv::findContours() is employed to compute contours from binary images.[KB16] In this application, a more direct geometric representation of these contours is needed, and that's where bounding rectangles become valuable. The cv::boundingRect() function provides an upright bounding rectangle, encapsulating a contour in a straightforward manner. This rectangle, represented as a cv::Rect type, has its sides oriented horizontally and vertically.[KB16]

- **Morphological operations:** Input images are often plagued by defects that cause inaccuracy in image processes. These defects could include noise, protrusions and concavities on the edges of regions of contrasting values, and even holes within them. Therefore, a multitude of Morphological Transformations are used in order to overcome these problems.[KB16]

  These transformations differ in their sophistication, with the more complex among them all being based on two basic transformations that we are concerned with: Dilation and Erosion.[KB16]

  Dilation is a convolution of some image with a kernel in which any given pixel is replaced with the local maximum of all of the pixel values covered by the kernel. Erosion however is the converse operation. The action of the erosion operator is equivalent to computing a local minimum over the area of the kernel.[KB16]

  Thus, both operations eliminate irregularities in different ways. Dilation generally tends to fill concavities, often increasing the size of the region being dilated, while

erosion generally tends to remove the protrusion themselves as it reduces the size of the region being eroded. [KB16]

- **Structuring Elements** When it comes to morphology, a structuring element (Kernel) is a shape that would interact with the image in order to draw conclusions on how it fits or misses the shapes within it. Thus, it could be used in functions such as cv::dilate(), cv::erode().[KB16]

  In OpenCV, the default kernel is characterized as square. Any array can be created and utilized as a structuring element if a nonsquare shape is required. However, the nonsquare shape that is needed is more likely to be common, in which case cv::getStructuringElement() could be employed for the creation of a custom kernel.[KB16]

  In this project, two distinct kernels are utilized:

  1. **Circular Kernel**: This kernel, when applied through the erosion process, modifies shapes within the image to be more rounded, emphasizing circular characteristics.
  2. **Square Kernel**: When used, this kernel accentuates the rectangular aspects of shapes, making them appear more square-like in structure after the process of erosion.

  By selectively applying these kernels, the project aims to influence and control the geometric properties of shapes within the image in order to select the information.

- **Blob detection:** A **blob** is any grouping of pixel values that form a colony or an object distinguishable from its surroundings or background[KB16].

  The `cv::SimpleBlobDetector` class implements just one of the many algorithms for blob detection. It works by first converting the input image to grayscale, and then computing a series of thresholded images from that grayscale image. By defining a minimum and a maximum threshold, as well as a threshold step, we could determine the number of threshold images computed. From these images,connected components are extracted.

By using `cv::findContours()`, and determining the centers of these contours, candidate blob centers are determined. Through the `minDistBetweenBlobs` parameter, control over which candidate blob centers are to be grouped and treated as part of one blob, and which ones are to be treated as the center of their own independent blob, is achieved. The resulting groupings are assigned a radius and a center, calculated from all the contours that form each group. The resulting objects are the keypoints. Once the blobs have been located, they may be filtered by:

  - Size (area)
  - Color (which is intensity, as the input was converted to grayscale earlier in the process)
  - Circularity (ratio of the area of the actual blob to a circle of the blob's computed effective radius)
  - Inertia ratio (the ratio of the eigenvalues of the second moment matrix)
  - Convexity (the ratio of the blob's area to the area of its convex hull).[KB16]

  The keypoints size, representing the number of blobs in an image, will play a crucial role in Chapter 6.

- Denoising: `cv::medianBlur` replaces each pixel's value with the median of the pixel values in its neighborhood. This is highly effective at removing salt-and-pepper noise.[KB16]

- Binary Thresholding: The cv::adaptiveThreshold function is utilized as the initial operation in each step of the image processing pipeline. Unlike traditional thresholding methods where a constant threshold value is used, this function calculates the threshold on a pixel-by-pixel basis. This is achieved by computing a weighted average of the block around each pixel and then subtracting a constant. By setting the cv::ADAPTIVE_THRESH_MEAN_C method where all pixels in the block are weighted equally.[KB16]
  The adaptive thresholding technique proves particularly useful when illumination is not even on all sides of the image. By thresholding relative to the general intensity gradient, this method allows for more sophisticated image segmentation. It should be noted that this function is designed for single-channel images (Gray-scale) and

requires distinct source and destination images for processing. Adaptive thresholding is a method that calculates thresholds for smaller regions, thus catering to variations in lighting conditions across the image.[KB16]

In this project, the function call looks like: cv::adaptiveThreshold(image, image, 255, cv::ADAPTIVE_THRESH_MEAN_C, cv::THRESH_BINARY, blocksize, c). Here, the cv::adaptiveThreshold function is converting a grayscale image to a binary image based on a threshold value that is the mean of the neighborhood area (defined by blocksize) from which a constant c is subtracted. If the pixel intensity is greater than the threshold, the pixel is set to maxValue, otherwise, it is set to 0.

`Liberal Approach:` A negative value of C effectively increases the threshold level, resulting in more pixels exceeding the threshold and being classified as foreground. This may include some noise but ensures that less detail is missed.
`Conservative Approach:` A positive value of C effectively decreases the threshold level, resulting in fewer pixels exceeding the threshold and being classified as background. This approach reduces noise but may miss some details.

Chapter 5

Software Implementation

In this chapter, the Gradle dependencies used in the project are discussed followed by a detailed examination of the code within the MVVM architecture.

5.1 Gradle

The following list of dependencies were included in the project to enable usage of various libraries functionalities required of the application. **Coroutines:** Dependencies for asynchronous task management.

Room Library: Dependencies for Room provide an abstraction layer over SQLite and include support for Kotlin coroutines.

CameraX Library: Dependencies for CameraX offer lifecycle-aware camera functionalities including basic features and custom UI.

ViewModel and LiveData: Dependencies for managing UI-related data and enabling reactive data binding.

Lifecycle Runtime: Dependencies for incorporating lifecycle-aware components.

ZXing: ZXing for decoding a DMC.

Core Android Libraries: Generated by default, these dependencies provide Kotlin extensions, backward compatibility, Material Design components, and UI layout.

5.2 Model: Core Functionalities

Within the Model layer of the MVVM architecture the core functionalities are implemented in `ImageHandler` class. It manages preprocessing steps for image handling, conducts storage operations, and concludes the decoding operation of processed images. In adherence to MVVM principles, it focuses on data handling and logic. The calling of these functions will take place in the ViewModel layer.

1. **Image Cropping**
    - `cropBitmap`: Upon capturing a photo, the entire camera view is taken as an image. However, for processing, only a specific ROI(the portion highlighted by the scan overlay) will be taken into consideration. This function adjusts the captured image by cropping this centered, highlighted region.
2. **Image Processing with Magnetic Data Matrix:**
    - `magneticDataMatrixProcessor`: A method linked to an external C++ library that converts the MDMC to DMC. This method processes an input bitmap and produces an output bitmap. The specific steps of how the image is processed will be discussed in chapter 6.
3. **Sampling Images for Image Processing Refinement (Part of debugging):**
    - `saveImageToExternalStorage`: This function handles the task of taking bitmap samples of MDMC. It first checks for the existence of a folder named 'Samples' under the 'Pictures' directory of the phone storage, creating it if absent. The primary purpose behind this functionality is to enable a more streamlined development workflow: By saving pictures as samples, they can be transferred to a computer separately work on refining and improving the image processing algorithm. As this functionality is not designed for the user, calls to this function are intended to be commented out in the final release.
4. **Data Matrix Decoding:**
    - `decodeDataMatrix`: Transforms the given bitmap into an array of pixels, further converting it to a binary format suitable for decoding. Using the

`DataMatrixReader` from the ZXing library, it attempts to decode the bitmap. If successful, the method returns the decoded text; otherwise, a failure message is returned.

5.3 Image Processing Library

This section focuses on JNI-related functions; the steps for image processing are elaborated further in 6.

The image is passed from the Kotlin side to C++. To call a function in Kotlin, the naming of the function must adhere to this format:
`Java_PackageName_ClassName_MethodName`. In this specific instance, since the package name is `com.mip_technology.mdmtxdecoder`, for invoking the function in the `ImageHandler` class, the name must be:
`Java_com_mip_technology_mdmtxdecoder_model_ImageHandler_magneticDataMatrixProcessor`. (Underscore replaced the dot) The function invokes three separate functions:

- `bitmapToMat`: Converts image from bitmap format to cv::Mat format, enabling processing via OpenCV functions.
- `magneticDataMatrixProcessor`: Transforms the image from an MDMC to a DMC.
- `matToBitmap`: Invoked upon successful conversion, converts the image back to bitmap format for decoding via the `decodeDataMatrix` function in the `ImageHandler` class.

Both `matToBitmap` and `bitmapToMat` functions are also implemented within this library.

5.4 Model: Database

Within the application, a database package is organized into various components. Each component fulfills specialized roles in data persistence and manipulation. Specific implementations and operational details of each component are elaborated in the subsequent subsections.

5.4.1 Database Class

The Database Class, `DmcStringDataBase`, serves as the main access point to the app's persisted data. It is defined as an abstract class, extending `RoomDatabase`. This class is responsible for creating the instance of the database and provides a method to access the DAO. A Singleton pattern is applied to ensure that only one instance of the database is created throughout the application lifecycle.

5.4.2 Data Entities

Data Entities represent the tables within the database. In this context, the `DmcString` class delineates an entity and encapsulates three attributes: an auto-generated `primary key`, a `decoded string`, and a `timestamp`. These attributes correspond to specific columns in the database table, reflecting the exact moment the decoding operation was executed and the resulting decoded string.

5.4.3 DAO

DAOs, or Data Access Objects, enables interaction with the database. The `DmcStringDao` interface enumerates a set of operations relevant to the `DmcString` entity. Specifically, it provides methods for insertion, update, deletion, and retrieval of all records within the corresponding table. These operations are characterized as `suspend` functions, enabling asynchronous processing and preventing potential UI thread blockages. In the project the only operations used are insertion and retrieval.

5.4.4 Repository

The `DmcStringRepository` class operates as an intermediary for database operations. It uses the DAO to interact with the database. The class has methods to insert, update, and delete entries. It also provides a LiveData list of all entries. This class uses Kotlin coroutines for asynchronous tasks.

5.5 ViewModel Layer

This Layer acts as an intermediary between the View and the Model and consists of two ViewModels:

5.5.1 Database ViewModel

The `DmcStringViewModel` class is the ViewModel class that handles database related tasks:

DMC String Retrieval:

- `dmcStrings = repository.allDmcStrings`: This method initializes and retrieves all decoded MDMC strings from the repository. The ViewModel uses this data for display in the corresponding History View (History View will be explained in this Chapter under Subsection 5.6.4).

Save Decoded MDMC String to Database:

- `saveDecodedStringToDatabase(decodedString:String)`: This method saves a decoded DMC string to the local database. A `DmcString` object is created with the decoded string. The repository is then used to insert this object into the database. This operation is executed on a separate thread because database operations are time consuming.

5.5.2 Core Functionalities

The second ViewModel entitled ”CaptureViewModel” handles the core functionalities. This ViewModel mediates between the MainActivity and the ImageHandler model class, overseeing permission checks, the camera related functions and image analysis functionalities. It leverages Coroutines to manage asynchronous operations and LiveData to pass changing variables to the MainActivity and ResultActivity.

- **Camera Permission Handling:**
  - The function checkPermissionsAndStartCamera updates a LiveData variable named cameraPermissionGranted. This variable is then observed in the View, serving as the ViewModel's provision of real-time permission status of the using the camera.
- **Camera Initialization and Management:**
  - The function startCamera initializes and starts the camera.
  - The function stopCamera stops all camera-related activities.
- **Real-time Image Analysis:**
  - The function processImage conducts asynchronous image processing. LiveData variables decodedDmcResult and decodedStringResult are updated and are provided for observation in the ResultActivity.
- **Processing Status:**
  - The function processImage also updates a LiveData variable named processingStatus. This variable is then provided for observation in the view.
- **Data Matrix Decoding and Storing:**
  - The function storeDmcString calls the saveDecodedStringToDatabase from the database ViewModel to store successfully decoded DMCs.

5.6 View

The View is the part of the application responsible for rendering the user interface and handling user interactions. It serves as an interface between the user and the system. In this application, the view consists of three activities.

5.6.1 Main Activity

The MainActivity is the primary activity and is the first screen displayed when launching the application. It sets up the camera, manages permissions, handles interactions, initiates the image processing task, navigates to the HistoryActivity and launches the ResultActivity once a DMC has been successfully decoded. It is implemented as follows:

- **onCreate() Implementation:**
  - Within `onCreate`, instances of `ActivityMainBinding`, `CaptureViewModel` is initialized. Additionally, LiveData variables, namely `cameraPermissionGranted` and `processingStatus`, are observed.

    `ActivityMainBinding` serves to link the activity's Graphic User Interface (GUI) elements to the following datvariables:
    `viewFinder`: The camera preview.
    `btnMainHistory`: A button intended to navigate to the `History` activity.
  - The `cameraPermissionGranted` LiveData triggers camera initialization or permission request.
  - The `processingStatus` LiveData initiates image processing and result handling. When it returns `true` indicating the successful decoding of the DMC , the camera is stopped, and a new intent is created to navigate to the `Result` activity. The intent carries `DmtxResult` and `stringResult` as data.
  - A button for history, `btnMainHistory`, is set up to navigate to the `History` activity upon clicking.
- **onResume() Implementation:**
  - Invokes `checkPermissionsAndStartCamera` from `CaptureViewModel`. This makes sure the camera is ready as the activity comes to the foreground, relating to the `cameraPermissionGranted` LiveData.

- **onDestroy() Implementation:**
  - The `stopCamera` function from `CaptureViewModel` is executed, releasing camera resources. This acts as the conclusion of the processes started by `processingStatus` LiveData.

5.6.2 ScanOverlay

The `ScanOverlay` class is an Android custom view designed to overlay a scanning area on the camera feed in `MainActivity`.

Visual Components The ScanOverlay consists of three major visual components:

1. **Background Mask:** A semi-transparent brown background, covering the entire view except for a transparent rectangle at the center.
2. **Center Rectangle:** A rectangular ROI outlined in a thick, green stroke (same as the green from the MIP logo). The rectangle corners are rounded with a slight `cornerRadius`.
3. **Scanner Lines:** Two red lines intersect at the center of the ROI. Their alpha values are animated from 255 to 0 in a one-second loop that repeats infinitely.

Animation An instance of the `ObjectAnimator` class handles the animation of the scanner lines' alpha value.

Integration in MainActivity The `ScanOverlay` is employed in `MainActivity` as a visual aid during scanning. It overlays the camera feed, providing real-time visual cues to assist the user in aligning the scan MDMC.

Layout Design and Responsiveness The layout of `ScanOverlay` is designed in Android Studio using the `ConstraintLayout` tool. Attributes like `match_parent` and `wrap_content` are used to ensure the layout adjusts to fit different screen sizes and remains consistent across various mobile devices. The `ConstraintLayout` allows for a flexible arrangement of UI components, making the view scalable and adaptive.

5.6.3 Result

The `Result` activity focuses its logic in the `onCreate()` method. LiveData's lifecycle-aware features remove the need for code in additional lifecycle methods like `onResume()`, ensuring stable functionality when the activity transitions between foreground and background states.

- **onCreate() Implementation:**
  - The `ActivityResultBinding` is initialized to bind the layout and the activity's variables.
    - `stringResult`: This variable is connected to a `TextView`
    - `bitmapResult`: This variable is linked to an `ImageView`
    - `btnRescan` and `btnResultHistory`: These variables correspond to buttons Rescan and History.
  - Two pieces of data are retrieved from `MainActivity`: `stringResult` which corresponds to the decoded string and `DmtxResult` which corresponds to the DMC image. These are displayed in a `TextView` and `ImageView`, respectively.

5.6.4 History

The `Hitstory` activity is primarily concerned with displaying the results stored in the database. Similar to the `Result` activity, the logic is encapsulated within the `onCreate()` method.

- **onCreate() Implementation:**

- The `ActivityHistoryBinding` is initialized to link the layout with data variables. This allows for data-binding functionalities within the `History` activity.
- `DmcStringViewModel` is set up by initializing the DAO and repository configurations.
- A `RecyclerView` is configured with a `HistoryAdapter` to display the list of data.
- The `dmcStrings` LiveData variable is observed to update the RecyclerView's data. When this LiveData is updated, it triggers a refresh in the RecyclerView's dataset.

Adapter: HistoryAdapter

The `HistoryAdapter` is designed for handling the display of a list of decoded DMCs strings and their corresponding timestamps in a `RecyclerView`.

1. **Data Source:**
   - The adapter maintains a list of `DmcString` objects. Whenever this data set changes, the UI gets updated automatically.
2. **ViewHolder Inner Class:**
   - Defines how to bind the data from a `DmcString` object to the corresponding view elements in the layout. The decoded string and its timestamp are displayed.
3. **View Creation:**
   - Instantiates the layout for each item in the list by inflating an `ItemHistoryBinding` object. A new `ViewHolder` is returned with this binding.
4. **Data Binding:**
   - The `onBindViewHolder` method is overridden to populate the `ViewHolder` with data from the list based on the item's position.
5. **Item Count:**

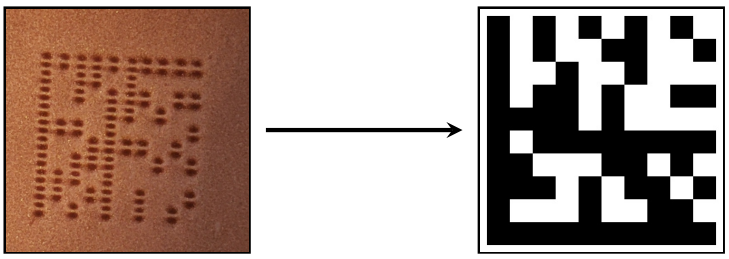
- Retrieves the number of items to be displayed, determined by the size of the data list.

In essence, `HistoryAdapter` functions as the bridge between the `RecyclerView` and the data set of decoded Data Matrix strings, ensuring efficient recycling and re-use of view components while maintaining a smooth and responsive user experience.

Chapter 6

Image Processing

The process of transforming an MDMC into a DMC as shown in Figure 6.1 demands careful attention to multiple factors that could potentially affect the accuracy of the output. Among these factors are poor lighting, contrast inconsistencies, incorrect orientation, rotational variations, and the presence of unwanted edges.



Magnetic Data Matrix Standard Data Matrix

Figure 6.1: Input and Output

6.1 Pipeline Rationale

The process for converting a captured MDMC to a standard DMC involves a sequence of seven distinct steps. The initial step focuses on corrections, the final step on image manipulation, and the steps in between on information extraction. The sequential arrangement of these steps is essential, as each depends on the successful completion of its predecessor.

The following enumerated steps are presented in their required sequential order:

1. **Preparatory Step**
2. **Rotation Correction**
3. **Region of Interest Detection**
4. **Cropping Correction**
5. **Orientation Correction**
6. **Grid Detection**
7. **Standard Data Matrix Conversion**

To understand the logic, one can envision each step of the process by tracing it back from the final result (a standard DMC) to the initial image (a captured MDMC) as shown in Figure 6.1.

To start, a perfect-condition image, shown in Figure 6.2, undergoes a thresholding process, allowing it to be transformed into a DMC. The details of this transformation will be elaborated in Section 6.9.

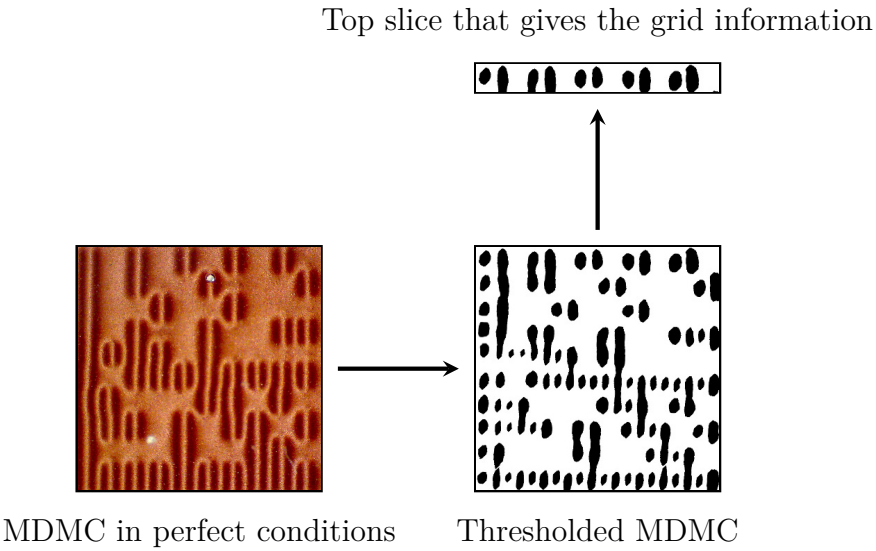


Figure 6.2: MDMC in perfect condition, thresholded and grid information isolated

Prior to reaching this final state, the captured image's grid must be determined. The number of black spots within the upper slice or left slice of the image (in the example of Figure 6.2 is 10 spots) corresponds to the grid size. This is rule applicable for every MDMC.

Correct orientation of the image is a prerequisite for reliable grid detection, ensuring that no erroneous top or left slice of the MDMC is utilized. Consequently, Orientation Borrection takes precedence over Grid Detection.

The pipeline must also address the potential rotation of the captured image and eliminate any peripheral unwanted areas surrounding it. Without addressing the image's correct rotation, cropping might still leave undesired edges, impeding the subsequent processing steps. Thus, Rotation Correction is implemented ahead of all steps.

It is noteworthy that the cropping phase is divided into two steps for optimization, discussed in greater detail in Sections 6.5 and 6.6.

Under poor brightness, none of the steps could be executed, therefore brightness correction has to be take place prior to of all of them, therefore it is implemented first in the Preparatory Step.

In summary, the pipeline is designed to prioritize stages based on dependency.

6.2 Key Attributes of the Image Processing Algorithm

Four unique aspects contribute to the success and versatility of the image processing algorithm developed for this project.

1. **Information Storage:** Each step in the pipeline will return its specific information that will be utilized in the final transformation. To manage those information, a `struct` is employed to serve as a centralized storage structure. It contains member variables such as the original `image`, `dimensions`, `rotationMatrix`, `orientation`, etc. As information is detected and processed at each stage, the corresponding attributes within the `struct` are updated.

2. **Prioritization and failure detection:** Each operation within the image processing pipeline is dependent on the successful execution of its predecessors. As explained in section 6.1, a specific order has to be followed for correct result, thus in case any stage of the pipeline encounters a failure, a Boolean variable, `addToTrashCan`, is set to true, triggering the process to restart with a new image.

3. **Modularity:** Given the complexity of image processing, a modular design is essential. The algorithm is structured into distinct classes, with each class representing one of the specific challenges faced.

4. **Consistent Starting Point:** To ensure the robustness of the information extraction process, each class in the pipeline begins its operation with the same initial processed image provided by the `struct`. This uniform starting point guarantees that every class is precisely tailored for the information it aims to extract. For instance, the cropping operation may require a higher thresholding value to eliminate noise from the edges, while the Data Matrix conversion might need more substantial black areas for a reliable result. Moreover, this consistency in the starting point facilitates the debugging process, enabling a clear view of which step might be failing or robust, thus allowing each problem to be tackled completely independently from the others.

   This pipeline of processes their corresponding stored information is depicted in Figure 6.3 as a flow diagram.

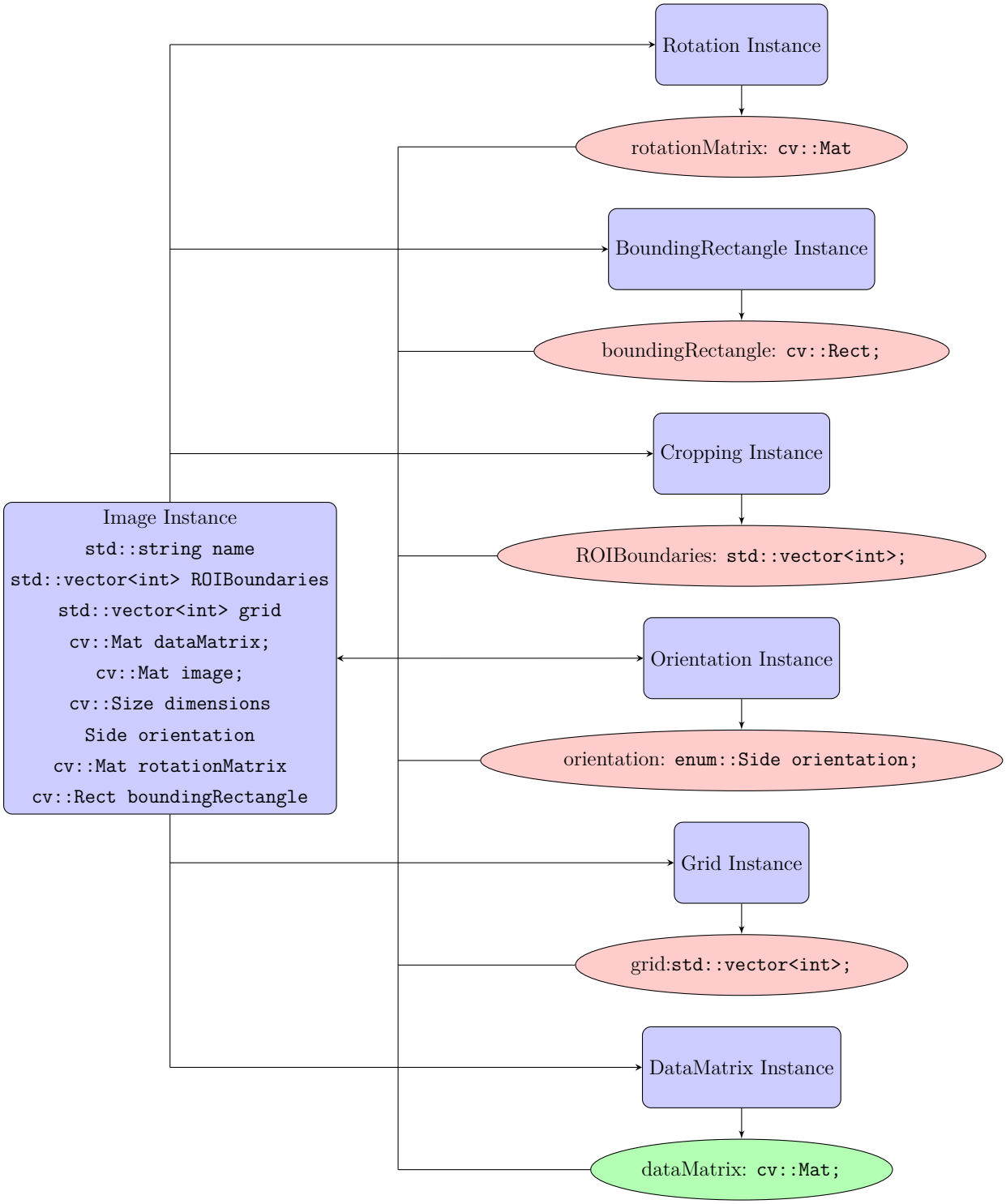


Figure 6.3: Flow diagram illustrating inputs and outputs of the image processing pipeline :rotation, cropping, orientation detection, grid detection, and the DMC conversion.

This modular design not only ensures that each step of the image processing pipeline is carried out in an orderly and prioritized manner, but it also provides a systematic way of storing and accessing the necessary information required to convert the MDMC to a standard DMC.

The implementation of each step is described in the sections below. Each section breaks down the complex process, providing a step-by-step guide on how each correction phase is implemented, ensuring a seamless transition from a raw image to a standard decodable DMC.

6.3 Preparatory Step

The initial steps in the image processing phase involved transforming the color space of the image to grayscale. This is because the color information is not relevant for our subsequent processes. This conversion also reduces the memory footprint of the image, as it reduces the number of channels from three to one.

The mean intensity of an image is quantified as the midpoint between the lowest and highest pixel values. This measure provides a gauge of the image's overall brightness or darkness. To achieve a standard brightness across all images, the target mean intensity is defined as 127.5 (The midpoint between the lowest pixel value and the highest pixel value). An adjustment factor is computed based on the difference between this target and the initial mean intensity of the image. This factor is subsequently used to scale the image's intensity, either augmenting or diminishing the brightness to align with the mean value. The final stage of the preparatory process involves noise reduction, implemented using the MedianBlur function. Figure 6.4 illustrates how the brightness is almost the same for initially different brightness images as well as the grayscaling and noise reduction.

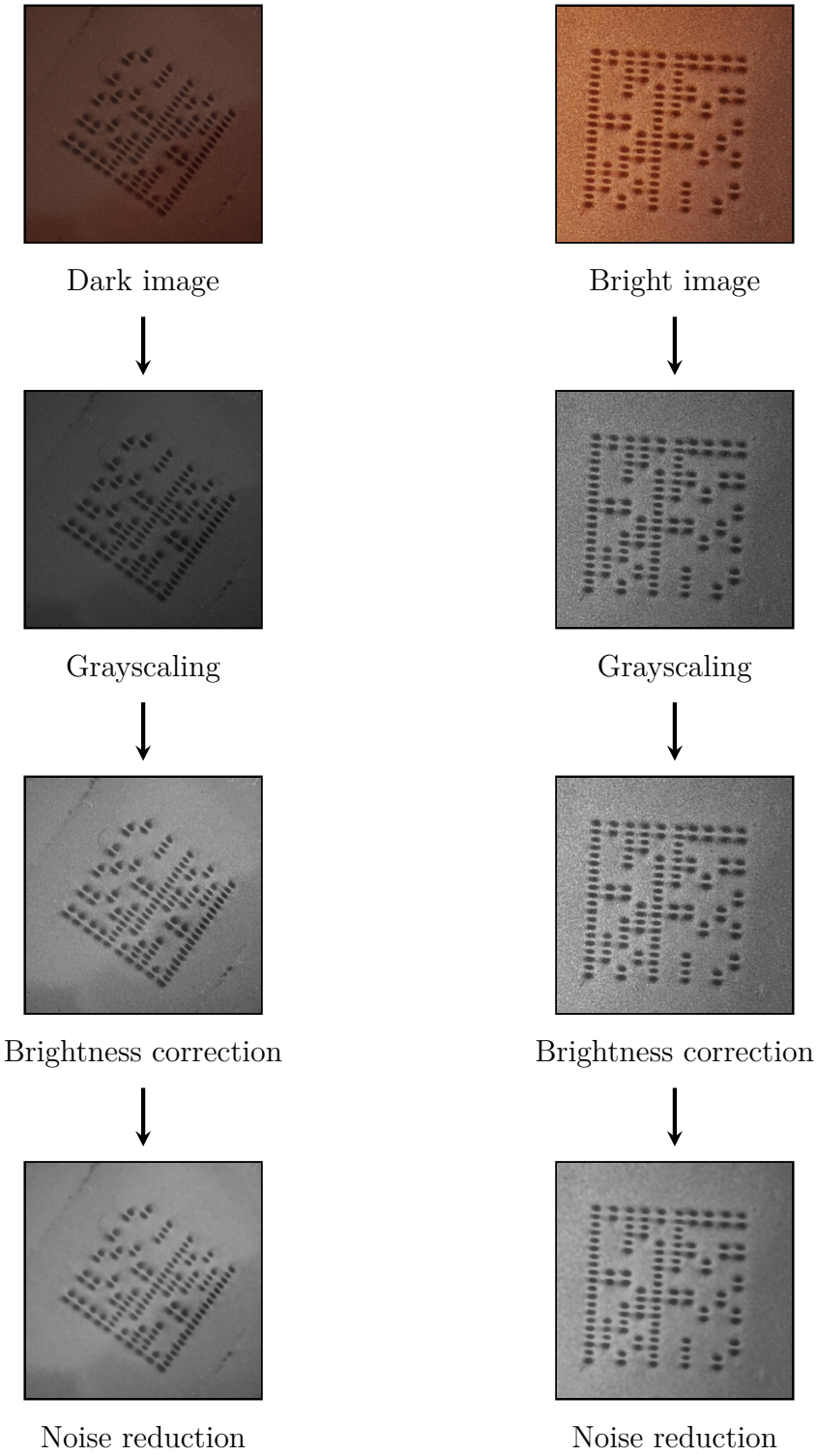


Figure 6.4: Grayscaling, standardization of Image Brightness and noise reduction across Different Inputs

6.4 Rotation Correction

Misalignment of an MDMC within a rectangle during scanning can result in a significant number of images being discarded. This problem is a common and normal occurrence in the scanning process. The objective is to identify certain image features that can facilitate the correction of rotation, thereby aligning the MDMC appropriately with the X-axis. The proposed approach is illustrated in Figure 6.5

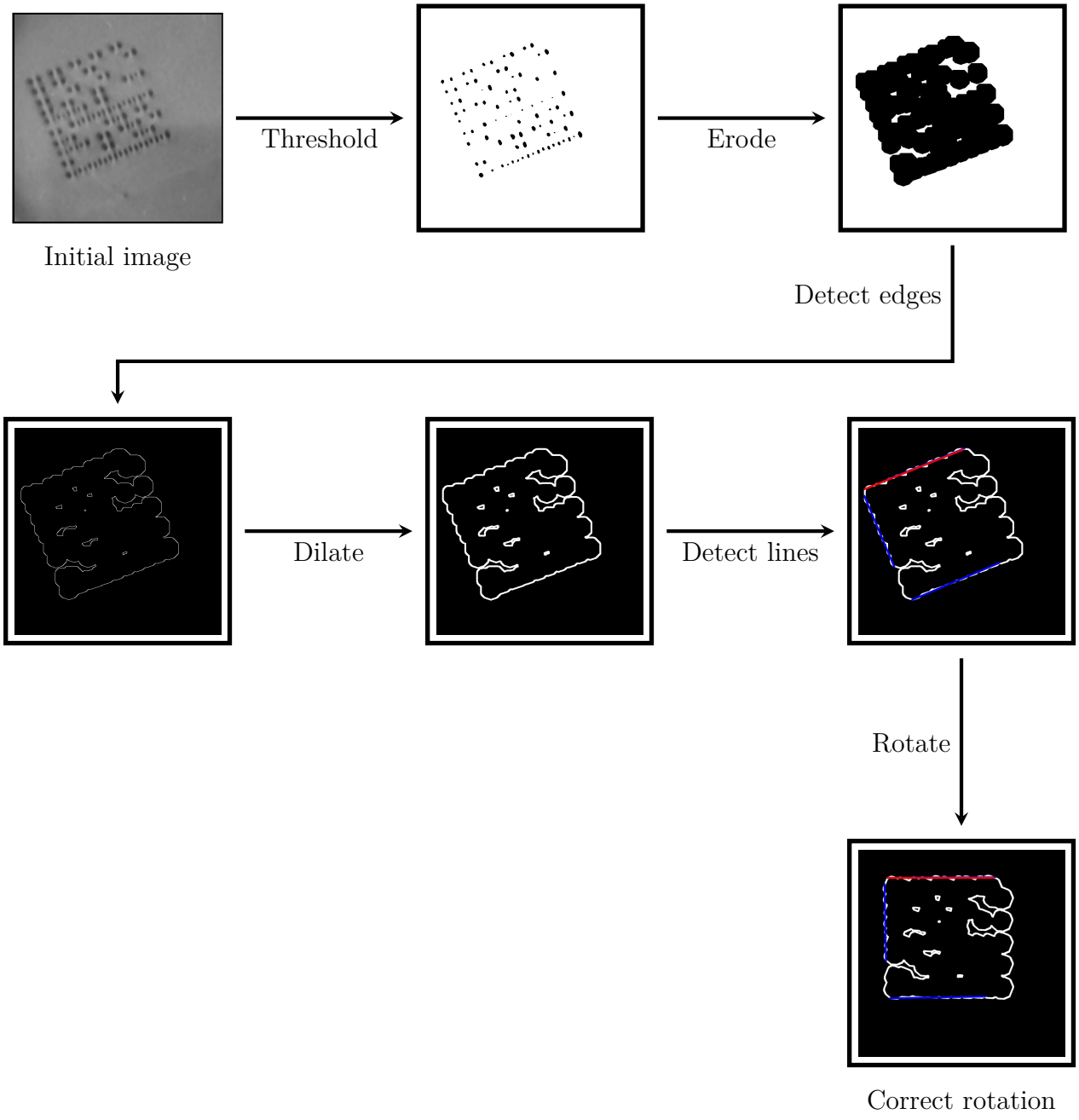


Figure 6.5: Rotation Steps

1. **Initial Image:** The process starts with the original raw image.
2. **Thresholding:** The raw image is then passed through a Utility function that applies an adaptive threshold. The adaptive thresholding here is chosen to strike a

balance between the conservative and the liberal approaches, which is suitable for the upcoming erosion and Canny edge detection stages (the second and third steps).

3. **Erosion:** The thresholded image is then subjected to erosion using a circular structuring element. This step aims to diminish any angles in the structure and render a more rectangular form to the entire shape. By doing so, it enables the computer to visualize a rectangular-like structure more effectively, thus enhancing the capability of the subsequent Canny edge detection stage to recognize the lines forming the rectangle that makes the boundaries of the magnetic data matrix.

4. **Edge Detection:** At this stage, the Canny edge detection method is applied to the eroded image. This operation highlights the square contour of our region of interest, making it stand out more prominently in the image.

5. **Dilation:** At this point, dilation is applied to the image, which has the effect of thickening the detected edges helps to join broken parts of the edges. This step facilitates the subsequent detection of the Houghlines, which serve as indicators of the image's orientation.

6. **Line Detection and Angle Calculation:** The Hough Line Transform is applied to the dilated image to detect lines. Of the lines detected, the longest one is selected and its angle with respect to the x-axis is calculated. This angle, will be used to calculate the rotation matrix which will be saved in the struct and passed to the next step.

7. **Rotation:** Using the computed angle, the image is finally subjected to an affine transformation, which effects the rotation. This final step is a visual confirmation that the images could be correctly rotated.

Hough Line Detection Failure

In the event that no Houghlines are identified within the image, the variable `addToTrashCan` will be set to `true`. Consequently, the image is discarded and the pipeline initiates processing with a new image.

6.5 Region of Interest Detection

The goal of this step is to optimize the cropping (which is the subsequent step) by eliminating all the blobs that are not part of the MDMC. This phase involves six key steps as shown in Figure 6.6

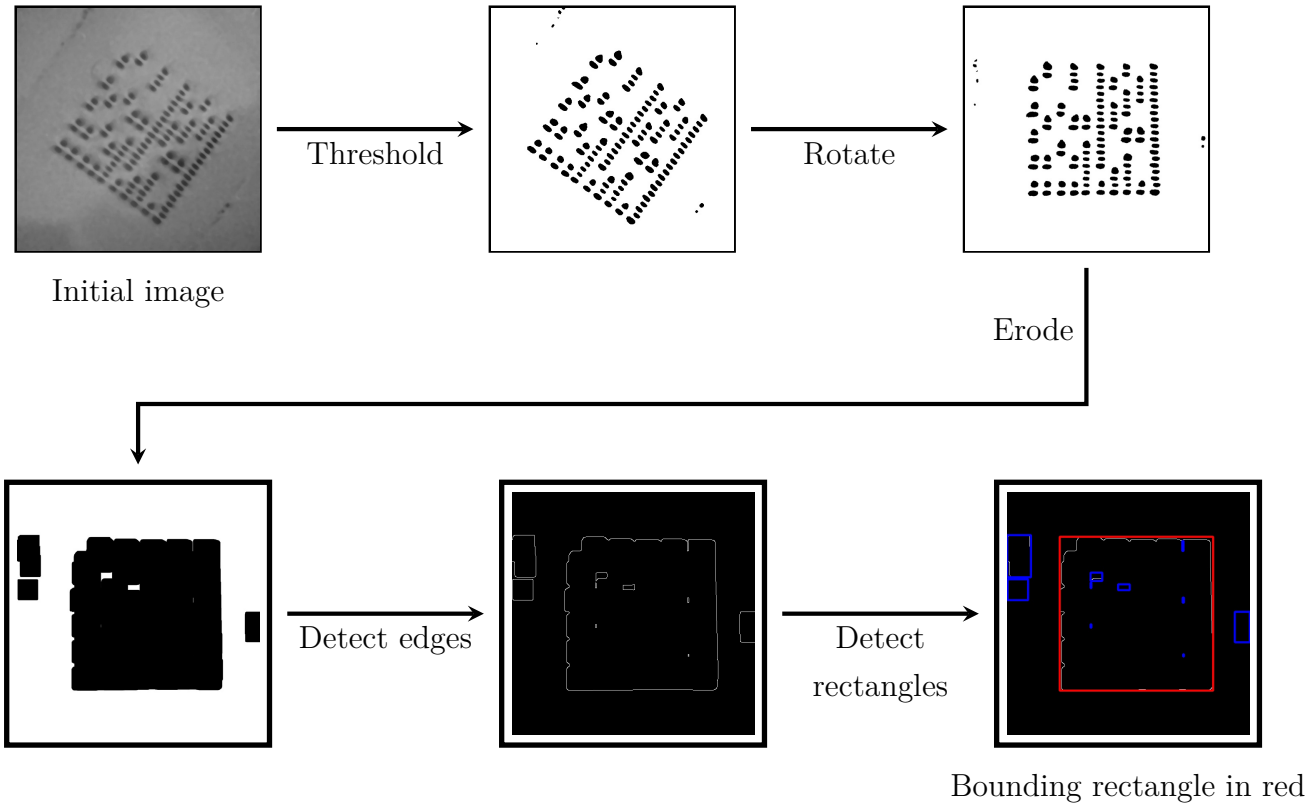


Figure 6.6: ROI detection

1. **Initial Image:** The process starts with a processed, grayscaled image.
2. **Thresholding:** The grayscaled image is subjected to an adaptive thresholding operation. This step accentuates the features of interest and prepares the image for the erosion process.
3. **Applying previous information:** In preparation for the cropping process, the image is adjusted based on the previously computed rotation matrix. This step is crucial because the cropping operation is rectangular. Therefore, the orientation of

the data matrix should be aligned correctly. If the data matrix is tilted, the cropping operation would potentially corrupt the Region of Interest.

4. **Erosion:** A rectangular structuring element is used to apply excessive erosion to the image, aiming to link the blobs together in a square shape. This helps in forming a coherent rectangular structure for further processing.

5. **Edge Detection:** The Canny edge detection method is used on the eroded image to identify the edges of the square shape, highlighting the region of interest.

6. **Bounding Rectangle Detection and Selection:** The contours are identified within the image, and bounding rectangles are computed for each one. Among these, the largest bounding rectangle is selected, which represents the MDMC. This selection is based on the understanding that the valid MDMC is expected to be the most prominent feature in the image.

Bounding Rectangle Detection Failure

In the event that no Rectangle is identified within the image, the variable `addToTrashCan` will be set to `true`. Consequently, the image is discarded and the pipeline initiates processing with a new image.

6.6 Cropping

The second cropping unfolds as shown in Figure 6.7

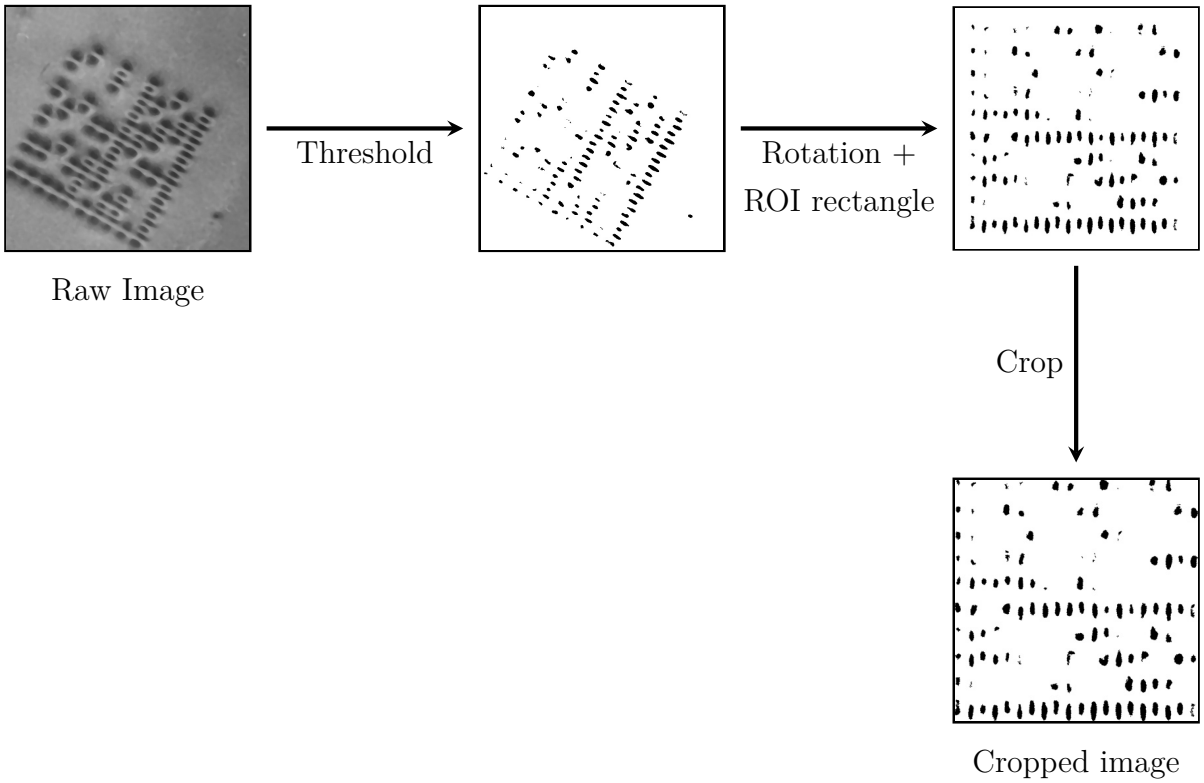


Figure 6.7: Cropping Steps

1. **Binary Threshold:** This step starts by applying a threshold function to the image, similar to the previous stage. However, the approach is more liberal as the focus is on eliminating as much noise as possible. At this stage, it's not necessary to conserve the full information - just enough to highlight the furthest points of the magnetic data matrix.

2. **Apply Previous Information:** As already established the cropping of the bounding rectangle should be prior to the begining of the cropping to eliminate the blobs taht are not part of the MDMC and prior top that operation the image has to be rotated first.

3. **Apply Previous Information:** As already established, the cropping of the bounding rectangle should be conducted after identifying and rotating the MDMC. This sequence of operations ensures that irrelevant blobs, not part of the MDMC, are eliminated, and only relevant information remains.

4. **Boundary Detection:** The main objective here is to accurately determine the furthest extremities of the data matrix, which are the leftmost, rightmost, topmost, and bottommost black pixels within the image. The procedure starts with a scanning process that runs column-wise and row-wise.

   For the leftmost boundary(x0), scanning begins from the first column and advances towards the right until the first black pixel is identified. This specific location is designated as the left boundary.

   The rightmost boundary(x0+Width) detection operates on a similar principle, but in the opposite direction. The scan commences at the last column and progresses leftwards. Upon encountering the first column with a black pixel, that particular position is marked as the right boundary.

   The topmost boundary(y0) is identified by initiating a scan from the first row and proceeding downwards until a black pixel is met, marking it as the top boundary.

   Conversely, the bottommost(y0+height) boundary is determined by starting the scan from the last row and moving upwards until the first black pixel is located, marking the bottom boundary.

   The intersection of these boundaries provides a comprehensive bounding box around the MDMC, encapsulating the entirety of the **??**.

   All these coordinates will be passed as arguments for the cropping function:
   `cv::Rect roi(x0, y0, width - x0 + 1, height - y0 + 1);`

   The final image visualization confirms the successful detection of these boundaries, ensuring that subsequent cropping operations can be performed without risking the integrity of the MDMC within the ROI.

   Cropping Failure
   The image is deemed incorrectly cropped if the difference between its height and

width exceeds 10%. This variation suggests that a part of the DMC might have been cropped, or that noise prevented the complete removal of white areas from one of the edges. In these situations, the variable `addToTrashCan` is set to `true`, and the image is discarded.

6.7 Orientation

The preparatory steps for determining the orientation is illustrated in Figure 6.8

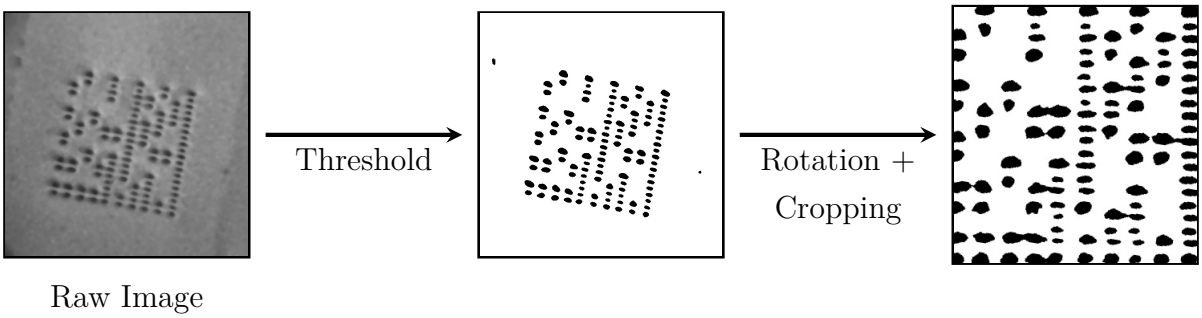


Figure 6.8: Orientation Processing Steps

- **Binary Thresholding:** The processing begins with the raw image where a conservative binary thresholding operation is performed. This conservative approach is critical in preserving the essential details along the boundaries of the image. The motivation behind this is to retain the maximum information required for the subsequent stages of blob detection, which in turn is crucial for determining the correct orientation of the image.
- **Application of Previous Information:** At this stage, the earlier computed rotation and cropping information are applied to the image. The purpose of this operation is to neatly align the magnetic data matrix into a perfect square. Moreover, it eliminates any unnecessary edges that would certainly disrupt the blob detection process.

In this part of the process, we perform operations on the boundaries of the image. Each side of the image (left, right, top, bottom) is sliced into small strips as shown in Figure 6.9,

each representing a specific edge of the image. The goal of the next operation is to detect the number of black spots in each of the strips.

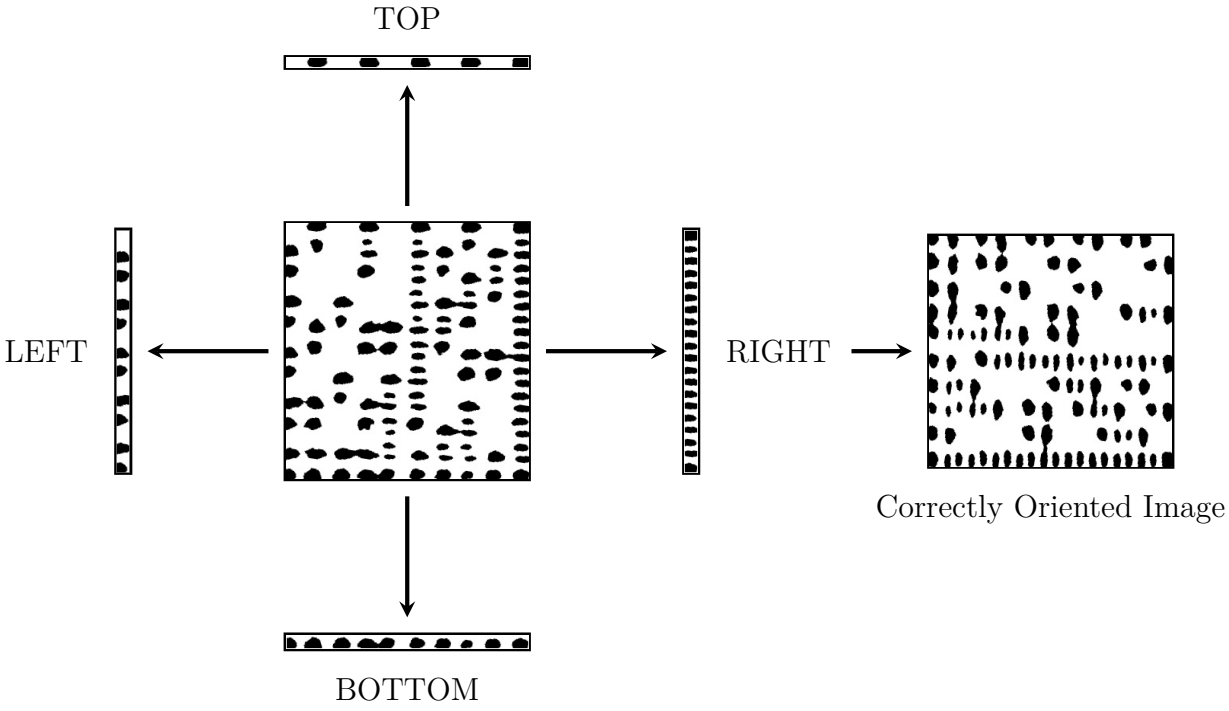


Figure 6.9: Orientation Detection Steps

- Since a blob is a cluster of pixels, it needs a well defined contour. A one pixel white border is added around each strip to isolate blobs that are touching the edges of the image, enabling them to be properly detected. The parameters for the blob in this case are a minimum distance between pixels of 0 to allow the algorithm to only count a complete isolated black spot as a blob and a minimum area of 3 pixels to avoid noise.
- Next, the blob detection function is applied to each strip. This function will return the number of blobs it found in the strip.
- An enumeration value (LEFT, RIGHT, BOTTOM, TOP) is associated with each side. This enumeration will be used later to determine the correct orientation of the image.

After these steps, the side with the most blobs is identified. This information is used to correct the orientation of the image:

- If the most blobs are found on the RIGHT side, the image is rotated 90 degrees clockwise.
- If the most blobs are found on the LEFT side, the image is rotated 90 degrees counter-clockwise.
- If the most blobs are found on the TOP side, the image is rotated 180 degrees.

At the end of this process, the image is correctly oriented, as indicated by the side with the most blobs. This step ensures that subsequent processes are working with an image in the correct orientation, enhancing the accuracy of the final results.

Orientation Detection Failure
This step would fail and will set `addToTrashCan` to `true` under the following circumstance: If the biggest number of blobs on one side is less than the double of the blobs on the side with the least blobs. This suggests there might be an issue with one side. While this criterion may seem arbitrary, it's important to note that the exact number of blobs is not critical, given that some blobs can clump together and be counted as a single blob.

6.8 Grid Detection

This step is the last information extraction step, if it is executed successfully the image is going to be transformed to a standard DMC. The preparatory steps for determining the orientation is illustrated in Figure 6.10

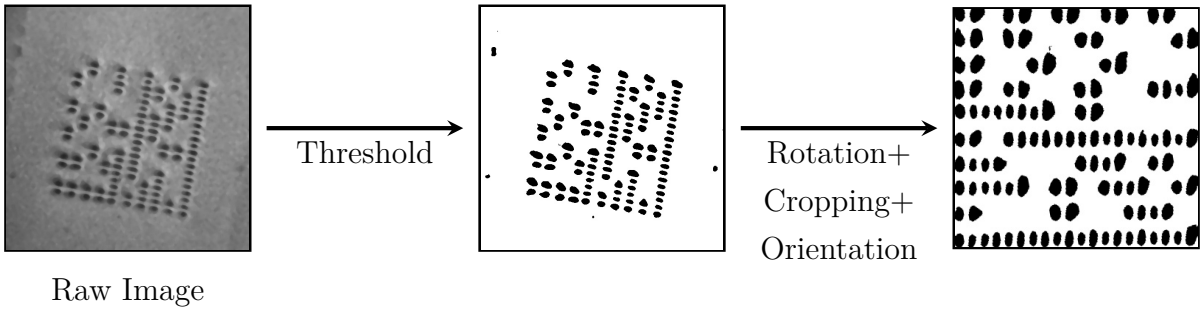


Figure 6.10: Grid Processing Steps

- **Liberal Binary Thresholding:** The image processing begins with a raw image upon which a more liberal binary thresholding operation is implemented. This liberal approach plays a crucial role in minimizing most noise while still retaining the indispensable information (like the number of blobs). This strategy is a critical factor for precisely determining the grid of the image.
- **Previous Information Application:** At this stage, the image is transformed according to the rotation, cropping, and orientation information calculated from previous steps.

Slicing and Blob Detection: Upon image preparation, a procedure similar to what was employed during Orientation is undertaken. Here, the image is divided into strips from both the left and right side as seen in Figure 6.11, and a white border is added to each strip. These steps prepare for blob detection on both sides of the image. Upon detecting the blobs, a validation check is performed to ensure that the number of blobs on the left is twice that on the right (a condition in a standard DMC). If this condition holds true, the number of blobs on the right is then returned, which will serve as the grid size.

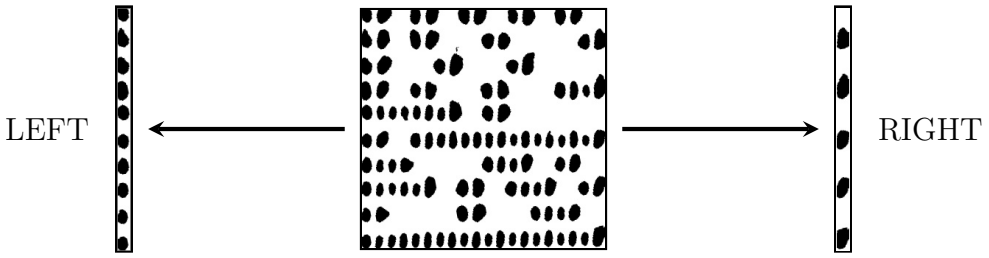


Figure 6.11: Grid Detection

Grid Detection Failure In the following events, this step is considered a failure and the pipeline starts with a new image:

- The number of the blobs on the right is not half of those on the left as established.
- The grid size is less than 10 (the minimum grid size of a DMC) or greater than 16.

  It is note worthy that the biggest grid size in a squared DMC is 144, but as this first application is still at an early stage, and due to concerns that the application might exhibit unusual behavior, it was limited to 16. Nonetheless, testing at the end was conducted with DMC of sizes 10, 11, and 12.

6.9 Standard Data Matrix Conversion

After collecting information from earlier steps, the MDMC can be converted to a DMC. Figure 6.12 shows the necessary steps for this conversion.

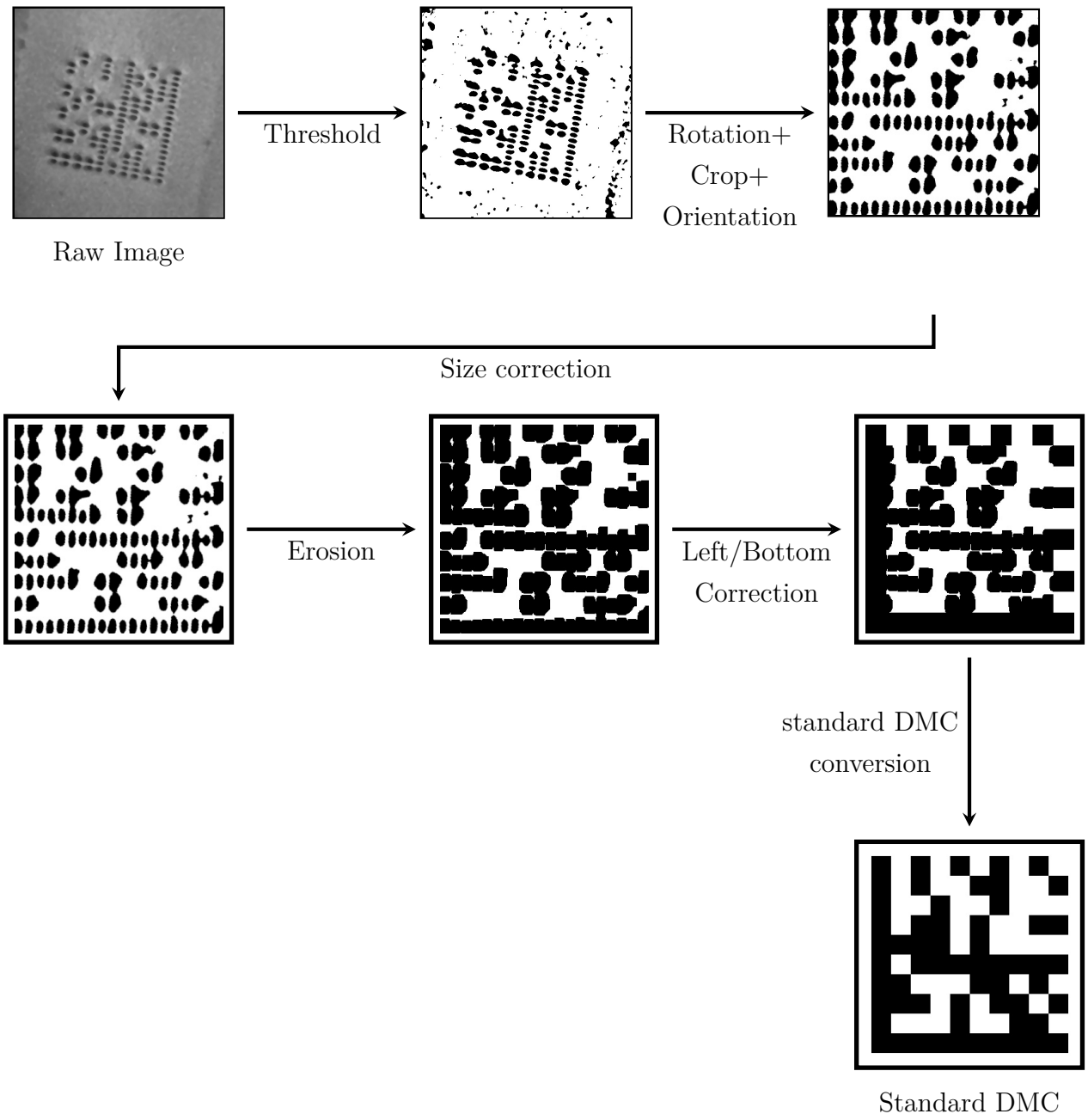


Figure 6.12: standard DMC Conversion steps

- **Conservative Thresholding:** The process begins with the raw image. A conservative thresholding technique is applied to preserve as much data as possible. The threshold value is chosen conservatively to retain maximum information, which is crucial for the subsequent stages.

- **Processing Previous Information:** At this stage, the image undergoes transformations based on the previously computed rotation, cropping, and orientation information. The transformations align the data matrix neatly into a perfect square, eliminate any unnecessary edges, and orient the image correctly. This prepares the image for the last stage of the process.
- **Size Correction:** The data matrix may lose its square shape due to the cropping operation, or even earlier during the capture of the raw image. In this step, the algorithm identifies the larger dimension (rows or columns) and adjusts the size of the smaller one to match it, restoring the perfectly square shape of the data matrix.
- **Border Adjustments:** The left and bottom borders of the data matrix are converted to black to help the algorithm produce more accurate results.
- **Erosion:** In this step, an erosion operation is applied to the image. This process refines the blobs, rendering them with a more squared shape, which aids in the accuracy of the subsequent steps.
- **Data Matrix Generation:** Finally, the data matrix is generated by iterating over the image and segmenting it into smaller squares based on the grid size. All the pixels in each square will be converted to its dominant color. This final step converts the identified blobs into interpretable data matrix information.

After this last step, the image is sent back to the Kotlin side of the application. Here, an attempt is made to decode the DMC, as detailed in chapter 5. If the decoding is successful, the result is handed over to the ViewModel. If not, the entire pipeline restarts with a new capture.

Chapter 7

Challenges and Results

This chapter provides an overview of the challenges that faced the development of this application, as well as the results achieved in the final product.

7.1 Challenges

The development of this application faced several challenges that had to be overcome. Below is a list of the most pertinent ones:

1. **Time Constraints and Learning Curves**:
    - The project's complexity, combined with a limited timeframe, presented challenges in thoroughly addressing each detail. Additionally, the learning curve for understanding Android and Kotlin posed delays in the development process.
2. **Asynchronous Programming Challenges**:
    - Balancing image capturing and processing tasks is crucial for real-time applications. While capturing is continuous, processing, being intensive, can introduce lags or delays.
    - Asynchronous programming using Kotlin coroutines was employed to maintain UI responsiveness alongside heavy image processing.

- Coordinating asynchronous tasks to avoid race conditions, deadlocks, and handling exceptions was critical.

3. **Integration between C++ and Android**:
   - Bridging C++ libraries with Android Java/Kotlin code requires tools like JNI, which have their complexities.
   - Different memory management paradigms between Android and C++ pose challenges, potentially leading to inefficiencies or memory leaks.
4. **Camera View Challenges**:
   - Ensuring consistent image quality across different Android devices.
   - Addressing issues arising from diverse lighting conditions, motion blur, or obstructions.
   - Managing varied resolutions and formats across devices.
5. **Platform Limitations**:
   - Different Android devices have varying computational power, making universal efficiency a challenge.
   - Managing the application's battery efficiency during intense processing.
6. **Testing and Validation**:
   - The wide range of Android devices with distinct hardware capabilities complicates consistent performance testing.
   - Validating the correct functionality of C++ algorithms when integrated with Android.
7. **Scalability and Maintenance**:
   - Preparing the application for updates or improvements in image processing techniques.
   - Ensuring compatibility with future Android updates or changes.

8. **User Interface Consistency**:
    - Ensuring that the GUI remains consistent across all Android platforms and screen sizes.
    - Utilizing design elements like ”match_parent” attributes to ensure elements adapt to different device screens.

7.2 Results

As the new application aims to address the limitations of an existing one, the results of its development could be thoroughly observed by comparing the two. The following subsections showcase the improvements made both in functionality and on a technical level.

7.2.1 Functional Comparison

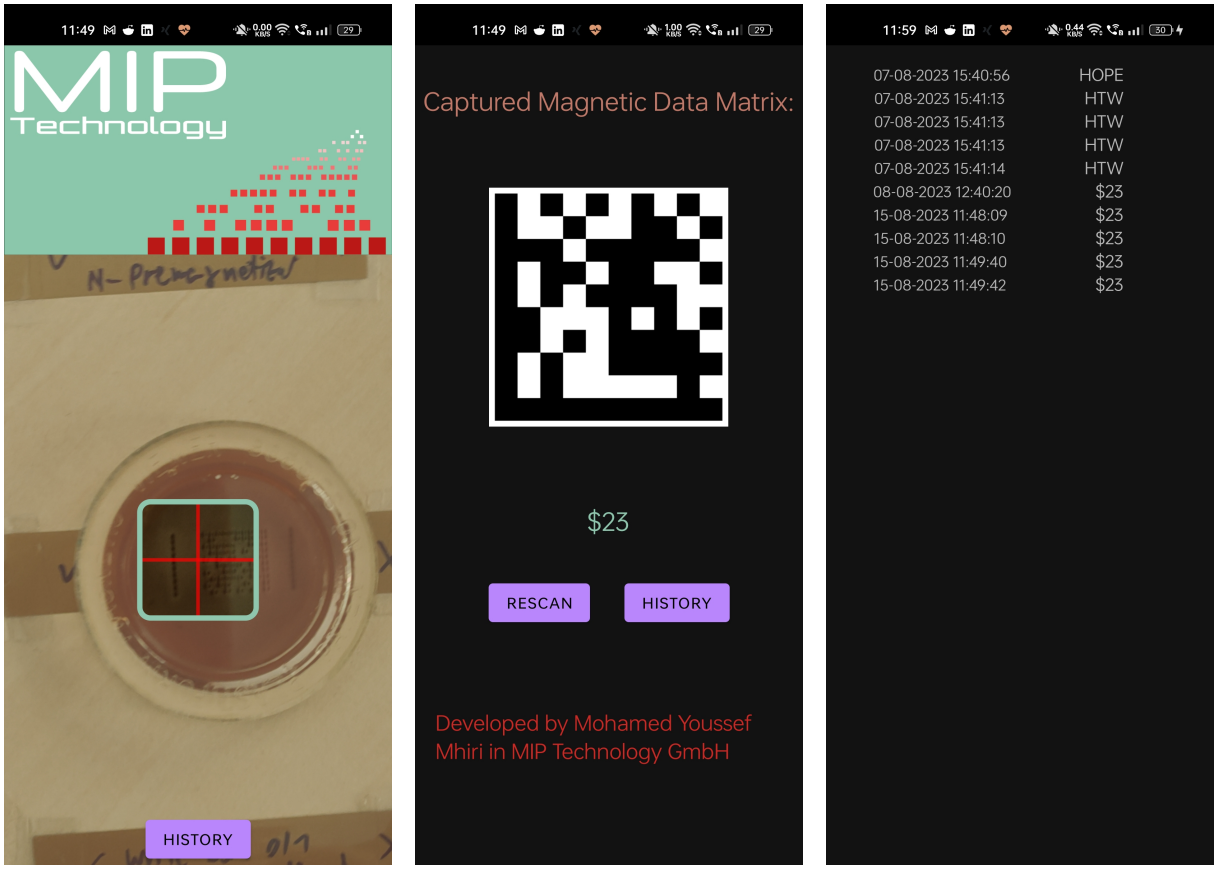
Table 7.1 compares the old and new applications in terms of functionality.

Area of Improvement	**Limitation of Existing Software**	**Improvements in the New Software**
Overall functionality	Lack of a data matrix decoder: The application stopped at generating Data Matrices without decoding them.	New software implements a Data Matrix Decoder.
Lighting Condition Versatility	Light Condition Sensitivity: The application is highly sensitive to lighting conditions.	Based on the mean value of the input image, the application would adjust its brightness in order to ensure the proper decoding of MDMC even under suboptimal or adverse lighting conditions.
Responsiveness	Suboptimal performance: Noticeable lag during the scanning process led to an inferior user experience.	The use of Asynchronous Programming, as well as the switch from an Interpreted Language to a Compiled language reduced lag times substantially.
Accuracy	High inaccuracy: The majority of Data Matrices generated by the application are incorrect.	The improvements in the image processing pipeline leading to better Light Condition Versatility were enough to substantially improve the application's accuracy. More importantly, the adding of the decoding component to the application made for a way to verify the correctness of each generated Data matrix, eliminating the possibility of inaccurate results altogether.

Table 7.1: Functional Comparison

Lifecycle management	No lifecycle functions employed: every time an event might cause the app to go to onStop the application process must be killed because on onRestart() because Kivy lacks native Android Activity lifecycle methods and they were not implemented by the developer.	The activities are lifecycle aware and all functionalities are implemented inside a lifecycle method.
Feedback	Lack of feedback: The only type of feedback provided by the early version of the application was visual.	A haptic feedback and a toast message indicating successful capture are implemented.
Logging	No persistent data: The application does not have a database.	The implementation of a Database allows for saving a log of previously decoded Data Matrices as shown in Figure 7.1a.
User Interface	The user interface was lacking in terms of input and feedback methods. The single button used was crude and aesthetically unpleasing.	A new user interface was designed, aiming to strike a balance between functionality, accessibility, aesthetic appeal, and minimalism. Based on the MIP color scheme, it distributes redesigned buttons across three distinct screens that cover all the new functionalities of the application as shown in Figures 7.1a, 7.1b, 7.1c.

Table 7.1: Functional Comparison (continued)



(a) Scanning screen (b) Result screen (c) Scanned MDMC list

Figure 7.1: Different screens of the application

7.2.2 Technical Comparison

Table 7.2compares the old and new applications on a technical level and includes comments on the purpose of the changes that were made.

	Older Application	**New Application**
Architecture	No architecture was considered while developing the code.	Adherence to the MVVM architectural pattern ensures separation of concerns. It also enables other engineers to further develop the application.
frameworks and IDEs	Kivy and Buildozer. This set is a cross-platform, community-driven, offers less standardization, limited built-in features, and a lack of documentation	Kotlin with Android Studio This IDE is backed by Google, providing a more standardized environment with extensive libraries and support. Moreover, they are optimized specifically for Android development.
Resources Management	The application runs on a single thread for both image processing and camera feed. This makes the image processing more time consuming, causing noticeable lag.	Asynchronous Programming makes use of the multithreaded architecture of modern smartphones. With this, the image processing uses a background thread independent from the main thread, which in turn is used for the scanning (cameraview). This reduces lag and allows the application to run smoothly. A third thread is for saving the successfully decoded MDMCs in the database.
Image Processing Programming Language	Python, an interpreted language is used	C++, a compiled language is used for enhanced code execution speed

Table 7.2: Technical Comparison

Optimisation of the Image Processing Pipeline

In the current library, the `adaptiveThreshold` technique is utilized instead of `threshold`, as detailed extensively in Section 4.3. While both basic thresholding and adaptive thresholding aim to binarize an image based on pixel intensity, the latter's dynamic adjustment of threshold values, contingent on localized pixel neighborhoods, offers distinct advantages. The method's adeptness in handling varying lighting conditions and shadows, as contrasted to basic thresholding, has been observed. Consistently, adaptive thresholding delivered more accurate and dependable outcomes in the implementation, underlining its superiority and the rationale behind its selection for the library.

In the current implementation, a significant preprocessing step is introduced which sets it apart from the Python version. While both implementations utilize grayscale conversion, their application points differ. In the Python approach, the image is passed as a BGR format to every subsequent step, being converted to grayscale at the beginning of each of those steps. This repetitive conversion not only increases computational overhead but can also inflate memory usage due to the retention of the BGR channels. In contrast, the new image processing algorithm optimizes this by performing the grayscale conversion only once in the begining. Once converted, the image is then consistently passed as a grayscale to the processing processing pipeline. This singular conversion not only reduces redundant operations but also offers enhanced memory efficiency. Coupled with this, the introduced mean intensity adjustment ensures consistent lighting across different images, and the median blurring aids in minimizing noise, ensuring a streamlined and efficient processing pipeline.

Chapter 8

Future Work and Conclusion

While the development of this application achieved its initial objectives, it also made me think of many new potential improvements and additions to the program that could be added in a future update.

The following sections mention the ideas that appear to be the most interesting, including new Android features, as well as Image Processing optimizations.

8.1 Proposed Android Features

GUI Themes:
Given the importance of user experience and aesthetics in the current mobile application environment, the user interface design contributes to the success of the product. By offering a diverse set of visual themes, the application can cater to a broad range of user preferences. Notably, integrating options such as light and dark modes, which are prevalent in modern app design, can greatly enhance user engagement and satisfaction.
Universal Decoder:
To make the application more versatile, one proposed feature is the addition of a function that can check if an image already represents a code decodable by the existing zxing library. The application becomes a one-stop solution for decoding, reducing the need for users to juggle multiple decoding apps.

Supported Code Formats by ZXing:

- QR Code
- Standard DMC
- Aztec
- UPC-A and UPC-E
- EAN-8 and EAN-13
- Code 39
- Code 93
- Code 128
- ITF (Interleaved Two of Five)
- RSS-14 (GS1 DataBar)
- RSS Expanded (GS1 DataBar Expanded)
- Codabar
- PDF417
- MaxiCode (mode 2 and 3)

Object detection feature:
Incorporating a dynamic object detection capability directly in the camera view offers a real-time feedback mechanism. This feature could rely on cv::RotatedRect for highlighting the MDMC, ensuring that the MDMC is identified correctly on the screen. By visualizing the detection in real-time, users can instantly verify the accuracy of the detected MDMC, making the scanning process more intuitive and efficient.

Diagnostic Feedback and Guidance:
Should the scanning process face challenges or failures, the application could provide users with diagnostic feedback, in the form of context-specific hints or guidance. For instance, if a step in the scanning process is failing, the system could highlight the problematic area or provide textual feedback, such as ”Please adjust lighting” or ”Magnetic Viewer appears agitated; please let it settle”, etc.

This diagnostic feedback not only serve as a troubleshooting mechanism, but also as a user-friendly interface enhancement, reducing user frustration, further involving the user in the scanning process and potentially increasing the rate of successful scans.

8.2 Optimizing Image Processing

Merging Rotation correction and Region of Interest Detection into a single class:
In the present implementation, the bounding rectangle class and rotation class are used separately to detect the ROI and perform the necessary rotation. A potential enhancement could be using the `cv::RotatedRect` class instead. This class can combine both the bounding rectangle and rotation information in one single information, which might simplify the code and possibly reduce computational demands since the Hough transform used in the rotation class uses more computational power.

MDMC transformation class improvement:
The standard data matrix conversion stage (6.9) did not undergo the same rigorous testing as the preceding steps (with around a 1000 images). One primary reason was the limited time allocated to the project, which inevitably led to prioritizing certain tasks over others. The classes prior to the MDMC conversion demonstrated a high success rate during the debugging, while a significant number of erroneous DMCs during the standard data matrix conversion. This makes further extensive testing a requirement to improve the decoding speed and robustness of the image processing Algorithm. By testing with larger and more varied image dataset and employing different functions and variables, the number of misclassified images could be reduced, thus enhancing the overall system performance.

8.3 Conclusion

In the field of digital identification, robust and dependable decoding methods are paramount. The introduction of MDMC technology serves as an addition to this domain, providing an identification mechanism resilient to mechanical stresses, elevated temperatures, and

challenging environments.
The development of the Magnetic Data Matrix Decoder Application serves to showcase the use of the MDMC as a solution for identification.
Throughout the project, a previous incomplete attempt served as the basis into refined application that mitigated many of the initial version's limitations, guaranteeing not only better functionality but also increased efficiency and improved user experience.
The updated application showcases notable improvements in image processing, effectively adapting to diverse lighting scenarios, provide intuitive feedback mechanisms and keeping records of previous scans. The new application notably outperforms its predecessor. Its structure, based on the MVVM architectural pattern, enhances code legibility and sustainability. Employing Kotlin as the programming language, coupled with the Android Studio IDE, optimizes the development trajectory and embraces advanced methodologies. Moreover, the deliberate utilization of C++ for the image processing algorithm ensures swift runtimes, leading to very low latency.
While working on this project was very fulfilling, the path to its completion was not free of hurdles. Between time constraints, platform limitations and camera view challenges, I was forced to adapt, expand my knowledge, and think outside the box. Yet it was these moments that made for the most enriching and satisfying moments in my journey.

List of Figures

Acronyms

DAO Data Access Object. III, 20, 35, 41

DMC Data Matrix Code. 4, 5, 6, 8, 9, 10, 11, 26, 27, 32, 33, 34, 36, 37, 38, 40, 41, 43, 44, 47, 48, 57, 59, 60, 61, 62, 63, 73, 74, 76, 77

GUI Graphic User Interface. 38

IDE Integrated Development Environment. 9, 13, 70

JNI Java Native Interface. 22

MDMC Magnetic Data Matrix Code. 4, 5, 6, 8, 10, 11, 13, 18, 26, 27, 33, 34, 36, 39, 43, 44, 45, 48, 53, 54, 55, 56, 61, 69, 70, 73, 74, 75, 76

MIP Magnetic Information Platform. 3, 25, 39, 76

MVVM Model View View-Model. 10, 18, 19, 32, 33, 70, 75

NDK Native Development Kit. II, 22

RFID Radio Frequency Identification. 2, 3

ROI Region Of Interest. 9, 10, 33, 39, 53, 56, 74, 76

UI User Interface. 10, 13, 17, 19, 22, 32, 35, 40, 41, 64

Bibliography

[Arn] Arnold Magnetics. *Magnetic Viewer B-1022*. Accessed: 2023-07-28. URL: `https://store.arnoldmagnetics.com/product/284/magnetic-viewer-b-1022`.

[Car03] Mario Cardullo. "Genesis of the Versatile RFID Tag". In: *RFID Journal* (Apr. 2003). URL: `https://www.rfidjournal.com/genesis-of-the-versatile-rfid-tag`.

[Car05] Mark C. Carnes. *American National biography Supplement 2*. The American Council of Learned Socities, 2005. ISBN: 9780195222029.

[Den90] Robert S. Cymbalski Dennis G. Priddy. "Dynamically variable machine readable binary code and method for reading and producing thereof". United States Patent. July 1990.

[Dev23] Android Developers. *Save data in a local database using Room*. Accessed on 2023-08-26. 2023. URL: `https://developer.android.com/training/data-storage/room`.

[DHL17] Berend Denkena, Katja Hasenfuß, and Christian Liedtke. "Gentelligente Bauteile – Genetik und Intelligenz in der Produktionstechnik". German. In: *Journal Name* (2017).

[DiM16] J. F. DiMarzio. *Beginning Android® Programming with Android Studio*. John Wiley & Sons, Inc., 2016. ISBN: 9781118705599.

[DMC99] Eric D Daniel, C Denis Mee, and Mark H Clark. *Magnetic recording: The first 100 years*. IEEE Press, 1999. ISBN: placeholder.

[Eve+96] David F. Everett et al. "Identification system and method with passive tag". US Patent 5,491,468. Feb. 1996.

[Faz21] Michael Fazio. *Kotlin and Android Development featuring Jetpack: Build Better, Safer Android Apps*. The Pragmatic Programmers, 2021. ISBN: 978-1680508154.

[Gar11] R. Garofalo. *Building Enterprise Applications with Windows Presentation Foundation and the Model View ViewModel Pattern*. O'Reilly Media, Inc., 2011.

[GW07] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing*. English. 3rd ed. Prentice Hall International, 2007. ISBN: 978-0131687288.

[Has11] Z. M. Hassan. "Credit System in the First Abbasid Period". In: *Uruk For Humanities* 4.2 (May 2011). pdf link: https://www.iasj.net/iasj/issue/2912 Accessed on 23/07/2023.

[KB16] Adrian Kaehler and Gary Bradski. *Learning OpenCV 3: Computer Vision in C++ with the OpenCV Library*. English. O'Reilly Media, 2016. ISBN: 978-1491937990.

[Liu13] Feipeng Liu. *Android Native Development Kit Cookbook*. Packt Publishing, 2013. ISBN: 978-1849691505.

[Meg98] Philip B. Meggs. *A History of Graphic Design*. John Wiley & Sons, Inc., 1998. ISBN: 978-1118772058.

[Nel97] Benjamin Nelson. *Punched Cards To Bar Codes: A 200-year journey*. Helmers, 1997. ISBN: 978-0911261127.

[Phi+15] Bill Phillips et al. *Android Programming: The Big Nerd Ranch Guide*. 2nd ed. Big Nerd Ranch, LLC, Aug. 2015. ISBN: 978-0134171494.

[Rog+09] Rick Rogers et al. *Android Application Development*. O'Reilly Media, Inc., 2009. ISBN: 9780596521479.

[Smy18] Neil Smyth. *Android Studio 3.2 Development Essentials - Kotlin Edition: Developing Android 9 Apps Using Android Studio 3.2, Kotlin and Android Jetpack*. Payload Media, Inc., 2018. ISBN: 9780960010929.

[Spä18] Peter Späth. *Pro Android with Kotlin: Developing Modern Mobile Apps*. Leipzig, Germany, 2018. ISBN: 978-1-4842-3819-6.

[SS00] Linda G. Shapiro and George C. Stockman. *Computer Vision*. 2000. ISBN: 978-0130307965.

[Sta23] StatCounter. *Mobile Operating System Market Share Worldwide*. Accessed on 2023-08-26. 2023. URL: `https://gs.statcounter.com/os-market-share/mobile/worldwide`.

[Str05] Gilbert Strang. *Linear Algebra and Its Applications*. 4th. Cenage Learning, 2005. ISBN: 978-8131501726.

[tea23] Google's Android team. *official resource for Android developers*. Accessed: 2023-08-27. 2023. URL: `https://developer.android.com/build`.

[Tec] MIP Technology. *MIP Technologies Website*. `https://mip-technology.de/en/`. Accessed on 22/03/2023.

[Tig22] Jomar Tigcal. *Simplifying Android Development with Coroutines and Flows: Learn how to use Kotlin coroutines and the flow API to handle data streams asynchronously in your Android app*. English. Packt Publishing, 2022. ISBN: 978-1801816243.

[Tri20] Hardik Trivedi. *Android application development with Kotlin*. English. BPB Online LLP, 2020. ISBN: 9789389423518.

[Wah12] E. L. Wahlberg. "The Wine Jars Speak: A text study". Uppsala University, Sweden, 2012, pp. 1–2.

[Wal83] Charles A. Walton. "Portable radio frequency emitting identifier". US Patent 4,384,288. May 1983.

[WS52] Norman J. Woodland and Bernard Silver. "Classifying Apparatus and Method". US-2612994-A. 1952.