Mobile applications for high-throughput seed characterization

by

Siddharth Amaravadi

B.S., Vignan Institute of Technology and Science, 2013

A THESIS

submitted in partial fulfillment of the
requirements for the degree

MASTER OF SCIENCE

Department of Computer Science
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2018

Approved by:

Major Professor
Dr. Mitchell L. Neilsen

Copyright

Abstract

Kansas State University is a world leader in the study of small grain genetics to develop new varieties which tolerate a wide range of environmental conditions. A phenotype is a composite of a plants observable traits. Several mobile applications, called PhenoApps, have been developed for field-based, high-throughput phenotyping (HTP) to advance plant breeding programs around the world. These applications require novel image analysis algorithms to be developed to model and extract plant phenotypes. Some of the first algorithms developed were focused on using static image analysis to count and characterize a wide variety of seeds in a single image with a static colored background.

This thesis describes both a static algorithm and development of a hopper system for a dynamic, real-time algorithm to accurately count and characterize seeds using a modest mobile device. The static algorithm analyzes a single image of a particular seed sample, captured on a mobile device; whereas, the dynamic algorithm analyzes multiple frames from the video input of a mobile device in real time. Novel 3D models are designed and printed to set a steady flow rate for the seeds, but the analysis is also completed to consider seeds flowing at variable rates and to determine the range of allowable flow rates and achievable precision for a wide variety of seeds. Both algorithms have been implemented in user-friendly mobile applications for realistic, field-based use. A plant breeder can use the applications to both count and characterize a smaller sample using the static approach or a larger sample using the dynamic approach, with seeds sampled in real time without the need to analyze multiple static images. There are many directions for future research to enhance the algorithms performance and accuracy.

Table of Contents

List of Figures

List of Tables

Acknowledgments

I cannot thank Dr. Mitchell L. Neilsen, but only be grateful to him. For all that I've learn't from him, the things that he has taught me, guided me both professionally and personally, I consider him as my ***Godfather***.

Dedication

Dedicated to Dr Mitchell L. Neilsen

Chapter 1

Introduction

Over the past century the field of agriculture and plant sciences have seen great changes. Technology has been adopted into these fields in the pursuit of greater yields[1], to meet the requirements of a rapidly increasing world population.

The introduction of computer-based systems to monitor and harvest crop yields in agriculture and the ability to study plant genomics and phenomics to understand plant traits has contributed to significant growth in these fields. A lot of research and development goes into these fields to build new and advanced technologies.

One such research effort is a new project at Kansas State University, called BreadPheno, which seeks to leverage novel advances in image processing and machine vision to deliver transformative mobile applications through several existing breeder networks (http://www.wheatgenetics.org/phenoapps). The initial application, called FieldBook[2], is used to log field data.

It has more than 5000 downloads, from the Google Play Store[3]. Building on this success, user-friendly mobile apps for field-based, high-throughput phenotyping (HTP) are being built and deployed for wide distribution. The latest addition is an app called Verify.[4]

Figure 1.1: *Field Book*

Figure 1.2: *Verify*

Novel image analysis algorithms are deployed to model and extract plant phenotypes. A robust development pipeline is assisted by both real-time field/laboratory/greenhouse trials by breeding collaborators and the broader community which includes middle/high-school students and independent breeders using the apps to explore both qualitative and quantitative differences under genetic control. A diverse set of crops and breeder networks are engaged to provide a diverse set of target plant phenotypes, environments, breeding programs, and working cultures.

1.1 Challenges

Dramatic increases in the speed and ability to collect precision phenotype data is needed to decipher plant genomes and accelerate plant breeding. Over the past decade, the availability of genomic data has exploded while the methods to collect phenotypes which can efficiently distinguish plants exhibiting rare or optimal genotypes from large populations[5], have made minimal advancements. This has led to a dramatic imbalance in data sets connecting genotype to phenotype and highlighting phenotyping as the remaining major bottleneck in plant breeding programs.

Food and nutritional security is a another grand challenge in the coming decades. The global population will increase to over nine billion and food demand will grow by more than

50%. Adding to this, climate change in the past few years is leading to drastic climatic conditions with plants changing the dates of activity, such as trees budding their leaves earlier in the spring and dropping them later in the fall[6].

1.2 Solutions

To address these challenges, novel advancements to leverage genomic information and expedite the improvement process to produce high yielding plant varieties that can adapt to the future climatic conditions. While genomic information has become inexpensive and readily available,the complementary phenotypes needed to understand the function of plant genomes and make selections in breeding programs has remained static, with insufficient development, particularly for phenotypes collected from field trials in breeding programs.

1.2.1 Technologies

This project seeks to advance the field of 3D graphics and modeling, data mining and deep learning through integration of simultaneous ground truth phenotypic measurements and imaging with mobile technology. By combining data from multiple research programs with ground-truth breeder knowledge, this project will lay the foundation for collecting training set data that can subsequently be used to extract and quantify complex phenotypes using deep-learning.

1.2.2 Platform

With smart phone ownership rates around the world, expected to reach 66% in 2018 from 58% in 2016 and in emerging and developing countries, it is rapidly climbing from a median of 21% in 2013 to 37% in 2015[7;8]. By focusing on novel algorithms delivered through mobile apps, innovative portable phenotyping tools can be rapidly deployed through readily available and highly penetrating smart phone and mobile technology.

This approach will enable rapid dissemination and broad usability. Collectively equipping thousands of breeders around the world with tools for rapid collection, processing and analysis of complex phenotypes will provide the foundation for increasing genetic gain that will ultimately result in improved productivity, food security, nutrition and income of small holder farmers and their families in emerging and developing countries.

1.3 Current Work

Previous seed counting algorithms built at Kansas State University have extended the classical Watershed Segmentation Algorithm to increase the accuracy of cluster classification when seeds overlap. The classical Watershed Algorithm is a flooding simulation which separates objects that are very close, or overlapping one-another. The extended Watershed Algorithm helps this by splitting estimated clusters using cross product calculations to measure concavity and identify logical convex points to be used to split clusters.

This thesis compares the previous algorithm to the new approach which accounts for the lack of processing power and memory on mobile devices while maintaining an accurate analysis.

Chapter 2

State of the art

2.1 High Throughput Phenotyping (HTP)

The ability to rapidly collect and analyze large amounts of morphological or physiological data of different plant varieties is known as high-throughput phenotyping (HTP). One of the major advancements in the field of high throughput phenotyping is to capture plant traits in a non-destructive manner[9]. Image-based methods for computing phenotypic measurements provide a promising solution for such high-throughput systems in plants,and is defined as the technology to record plant traits by capturing images with minimum to no invasion.

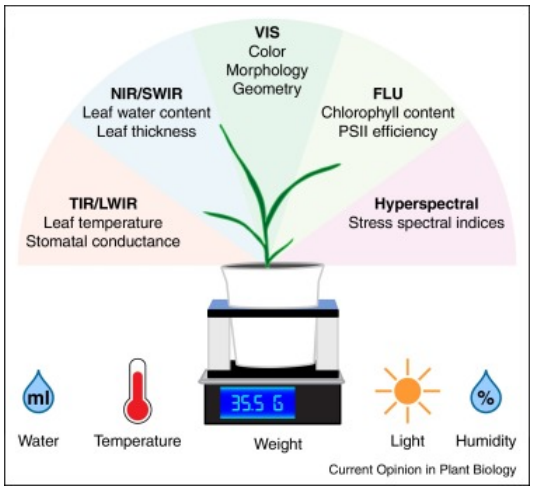

Figure 2.1: *Phenotypes*[9]

There have been many successful applications of image analysis to plant phenomics. However, these applications of image-based HTP have been largely concentrated on laboratory or controlled environment facilities. While providing highly accurate phenotypes, the ability to make these tools portable and practical enough for trials in field conditions remains limited. Moreover, the limited capacity and infrastructure needed to develop controlled environment HTP platforms makes this approach unpromising for breeding programs that currently manage population sizes much larger than even the worlds largest facilities.

2.2 Phenotyping bottleneck

Field-based phenotyping remains the major bottleneck in most breeding and genetics programs, including cassava and wheat. The bottleneck in field-phenotyping has resulted in the rise of interest in applying remote sensing technologies to field crop monitoring and in this regard field phenomics is more advanced than controlled environment[10]. As recognized by the breeders, this bottleneck can be overcome by adopting and/or developing innovative, high-throughput, and accurate hand-held HTP tools that can greatly increase the ability of breeders to generate useful data more cheaply and rapidly. As such, there is considerable willingness and motivation in the breeding community to test and adopt new technologies that can make the breeding program more efficient.

The various image analysis algorithms and their extensions that are developed have been implemented in user-friendly mobile applications. These apps are deployed and distributed using various platforms. They are adopted by hundreds of breeding programs around the world. They are available online at no cost and have been designed to be standalone, intuitive and effective. They are delivering a significant increase in breeding productivity and are the foundation for the current project. These tools focus on simplifying and digitizing data collection by focusing on procedures that are common to all breeding programs.

2.3 PhenoApps

2.3.1 OneKK

A beta version is available for OneKK, an app that can give accurate size and shape parameters for seeds and tubers/roots. With the namesake of OneKK for capturing a thousand (OneK) kernel (K) traits, the app extends to any size and scale of seeds Figure 2.2. The app uses integrated image processing algorithms based on OpenCV[11;12], to estimate length and width while attempting to count seeds or roots that are spread on a colored mat that includes reference circles of a known size to convert pixels to absolute size in mm[13] .

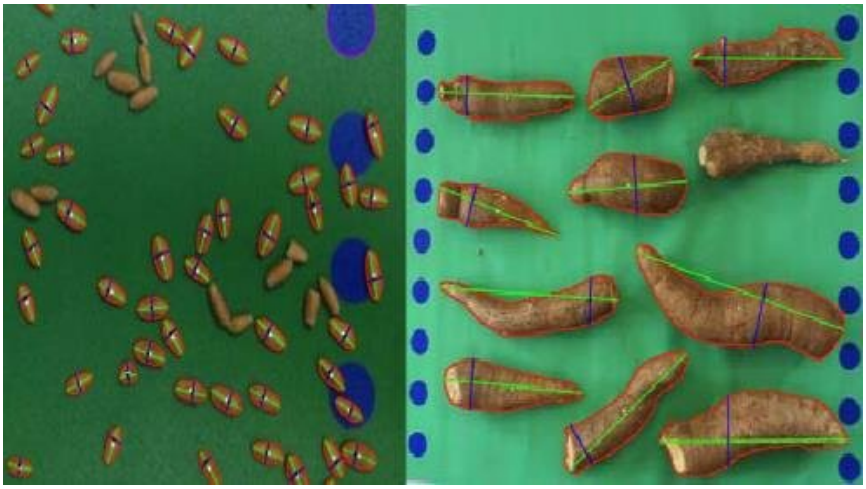

Figure 2.2: *Wheat and Cassava roots*

The input image sample is processed by separating the three regions of interest: the green background, the blue reference circles, and the golden grain seeds. This is done using Open CV, by using HSB color space we can easily separate the wheat seeds from the image by selecting a hue range of 0 - 50 and leaving the saturation and brightness as 0 - 255. An abundant number of pixels are green and denote the background. Then we have brown or golden colored pixels which denote the grains and a few blue colored pixels denoting the reference circles. The image is then converted into binary and then watershed segmentation based on Euclidean Distance Map is applied. The results were accurate and this approach was also extended by splitting the cluster of seeds to get even more accurate results.

A newer version of OneKK is under development which implements the static algorithm, another extension of watershed segmentation, where the seeds or roots are spread on a light-box bounded by four same currency coins. These coins serve the same purpose as the reference circles in the previous approach i.e to convert pixels to absolute size in mm. This approach also addresses the limitation of watershed, to efficiently separate small clusters of seeds.

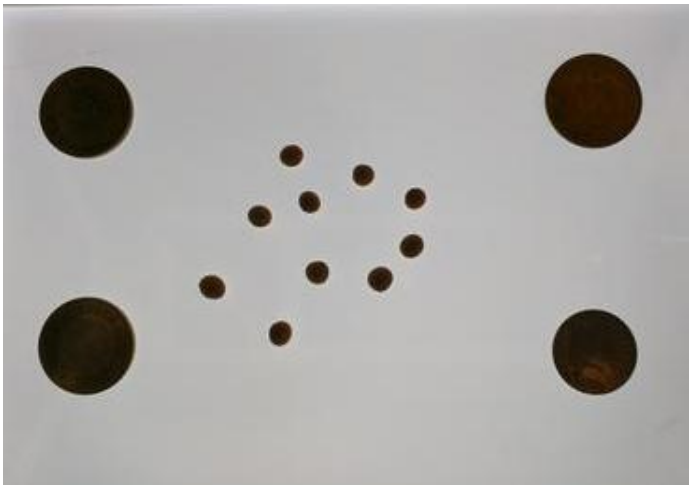

Figure 2.3: *OneKK light-box input sample*

This approach also supports the sample images captured on a normal background instead of a light-box. In case the sample images are captured with a normal background, it requires a little additional preprocessing before the actual processing. Thus making the algorithm more generic and makes OneKK more flexible to use.

2.3.2 Abacus

The other application currently under development with code name Abacus, which signifies counting, implements the dynamic, real-time algorithm to count large samples in real time. A custom test bench is designed to easily setup and perform large sample counting. The designs of the test bench can easily be replicated or modified.

Chapter 3

Static Algorithm

3.1 Introduction

The new static algorithm is a novel extension of watershed segmentation but improves over the previous extensions of it by generalizing the background mat over which the seeds or roots are spread. The algorithm uses a white light-box background instead of a colored one. Thus enabling it to be easily implemented on portable, memory and processing power-constraint devices like mobile phones. The first step is to capture the sample image over a white light-box background (Figure 3.1)

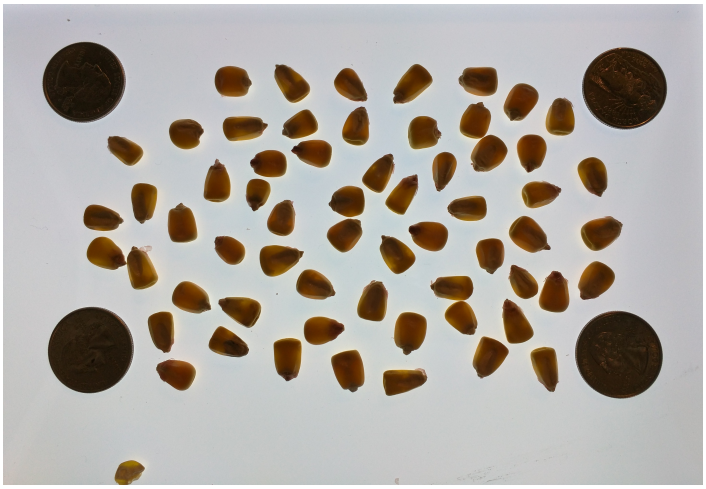

Figure 3.1: *Maize sample over light-box background*

3.2 Preprocessing

The next few steps are part of preprocessing before watershed segmentation is applied

Step - 1 : a histogram from the actual image is calculated to estimate a threshold value

Step - 2 : a binary threshold transformation is applied on the image using the above calculated threshold value (Figure 3.2)

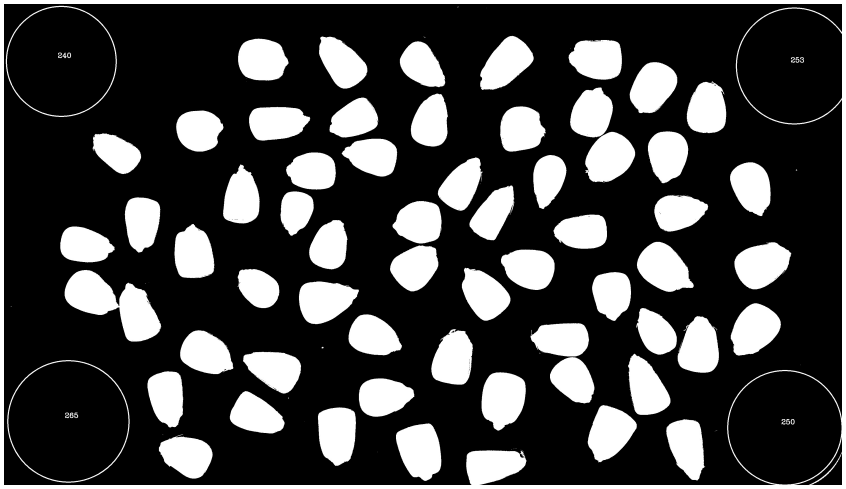

Figure 3.2: *Binary threshold*

Step - 3 : the binary image is then converted to gray scale and sure background is extracted using a kernel of size 3 and dilating for three iterations (Figure 3.3)

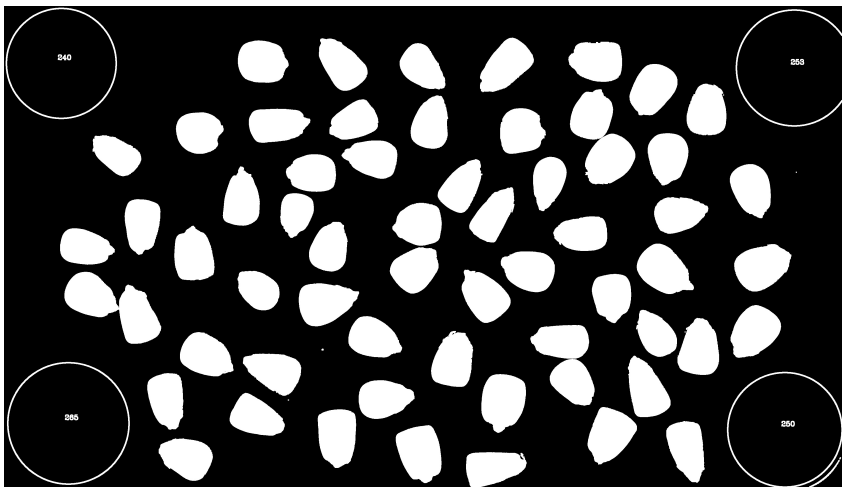

Figure 3.3: *Sure background*

Step - 4 : an opening mat is obtained by a morphological transformation of MORPH_OPEN on the binary mat, which is essentially an erosion followed by dilation for noise removal. distance transform is applied on the opening mat to calculate the Euclidean Distance Map (Figure 3.4)

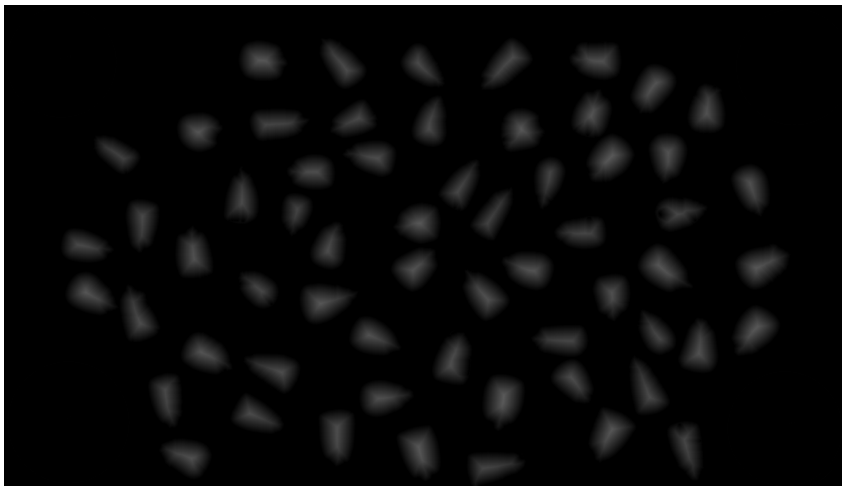

Figure 3.4: *Euclidean distance map*

Step - 5 : the distance transform is used to calculate the sure foreground by thresholding the distance transform with a lower bound of 8 and upper bound of 255 and the extracted foreground components are used as markers (Figure 3.5)

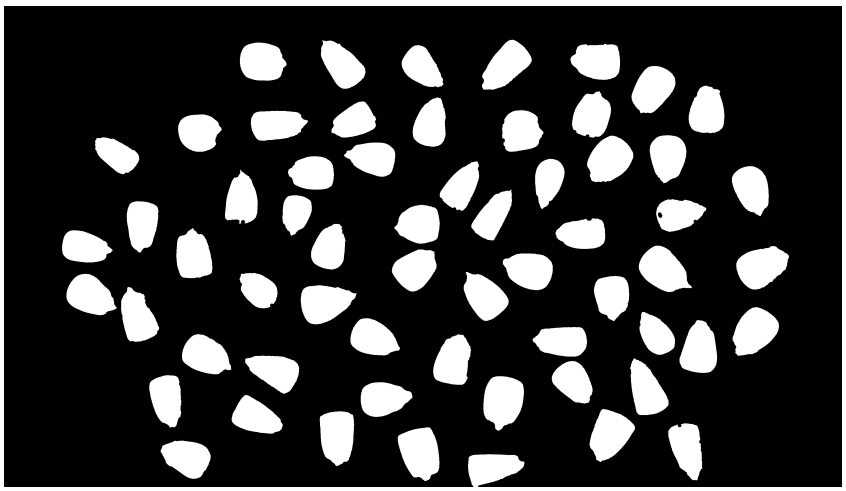

Figure 3.5: *Sure foreground*

Step - 6 : the region other than the sure foreground and sure background is considered unknown and is calculated by subtracting sure foreground from sure background (Figure 3.6)

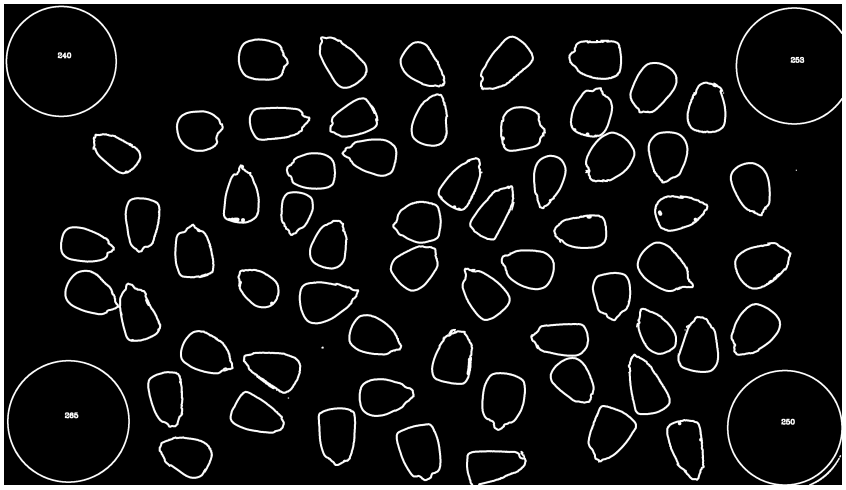

Figure 3.6: *Unknown*

Step - 7 : using a connectivity of 4 or 8 we calculate the connected components with stats using the sure foreground mat. The connected components give us the number of labels, the label matrix, stats matrix and centroid matrix. (Figure 3.7)

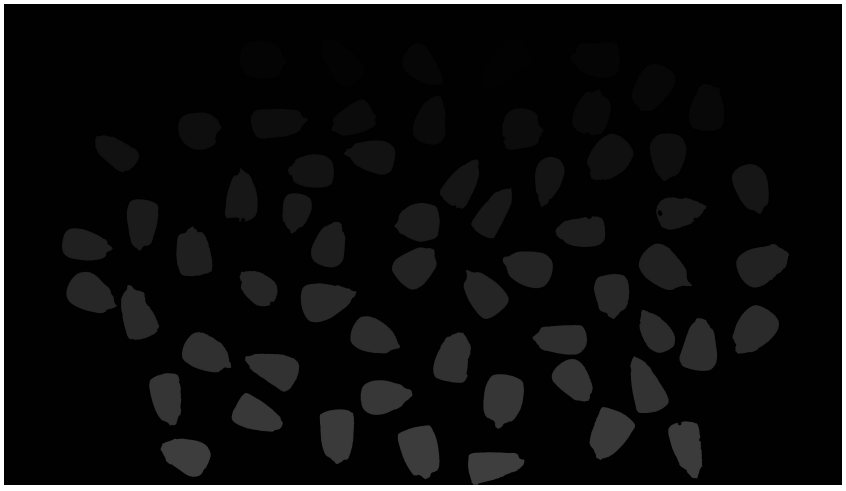

Figure 3.7: *Labels*

3.3 Watershed Segmentation

Once the preprocessing is done we proceed with Watershed segmentation with the following steps,

Step - 8 : ignoring all the markers related to the background (marked as 0), we apply watershed segmentation on the image using the markers.

Step - 9 : loop over unique segments returned by watershed segmentation and determine the contours to compute the minimum and maximum contour area thresholds (Figure 3.8)

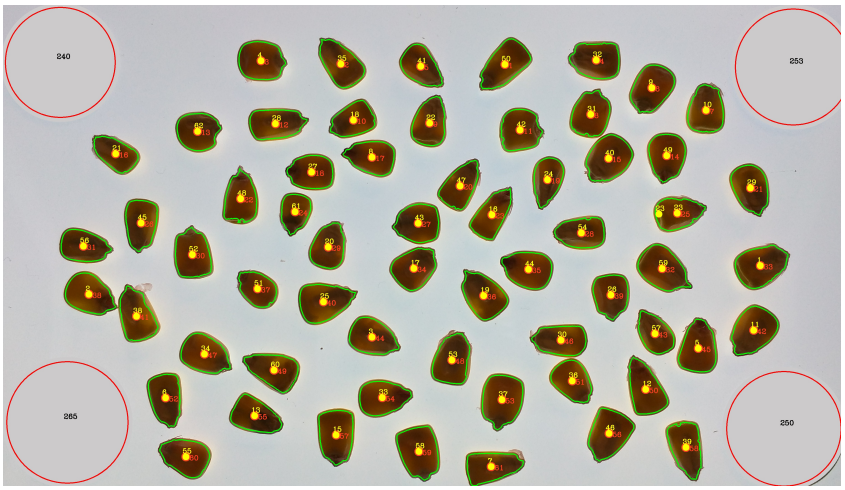

Figure 3.8: *Contours*

Step - 10 : iterate over the number of contours skipping the contours below or above the determined threshold estimated in Step - 1. If the contour is within the threshold bounds then we consider that as a seed and calculate the area, perimeter, moments to get the centroids of the contour and also the bounding box

Step - 11 : estimate the number of seeds based on the size of contours, by iterating to get a better estimate for average seed size, adjusting the average seed area and seed count accordingly

3.4 Extension of Watershed Segmentation

After the watershed segmentation is completed, we implement the extension to it, which checks for cluster of seeds and makes the necessary corrections to the count. The process is done with the following steps:

Step - 12 : apply our algorithm extending the watershed segmentation, without actually re-running the watershed segmentation.

Step - 13 : check if we need to segment the seeds with both size greater than the estimated average area and perimeter, at least 1.5 times the average.

Step - 14 : calculate the dot sum of the points and also compute the absolute value of negative dot products and sum. We make sure we check small seeds with relatively large dot sums (dot sum is greater than or equal to 550) and update them accordingly.

Step - 15 : check if the estimated seed count matches with the segmented area counts adjust the seed count accordingly. (Figure 3.9)

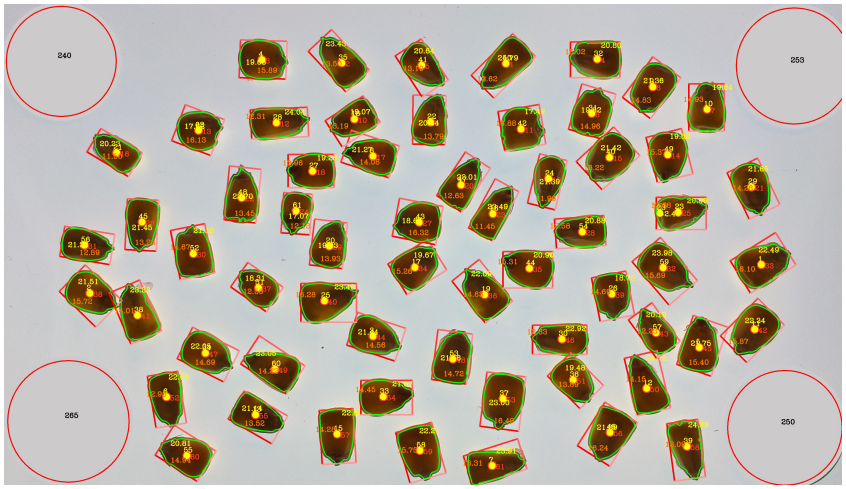

Figure 3.9: *Analyzed sample*

The above steps finally yield the seed count in the sample captured. In other case where in the sample image is captured with a normal background and not a light-box, we add an additional step to preprocessing of performing a RGB color threshold based on the background to black. We then perform all the steps as above to get the seed count.

Chapter 4

OneKK

4.1 Introduction

The above described algorithm is implemented in the latest version of OneKK. The app is used to capture the image of the sample placed on the light-box, bounded by four same currency coins, used as reference markers. The coins are detected using a coin detection mechanism, implemented using Circle Hough Transform(CHT)[14]. It is also used to estimate the centroid and bounding box of the coins used to estimate the area of the coin and determine the four corners of top-left, top-right, bottom-left and bottom-right respectively used to limit the search space. Once all the four coins are recognized the user then captures the image of the sample and processing begins.

4.2 Application architecture

OneKK constitutes of variety of intensive functionalities with respect to UI, Computing and Data. UI has a camera preview with a dynamic grid overlay. An experimental real-time coin recognition is also available, which works similar to that of face recognition. This is a graphic intense feature. Computing involves a lot of image processing on high resolution images and heavy computations like dot products are done on large mat's.

The architecture of the application is shown in Figure 4.1,

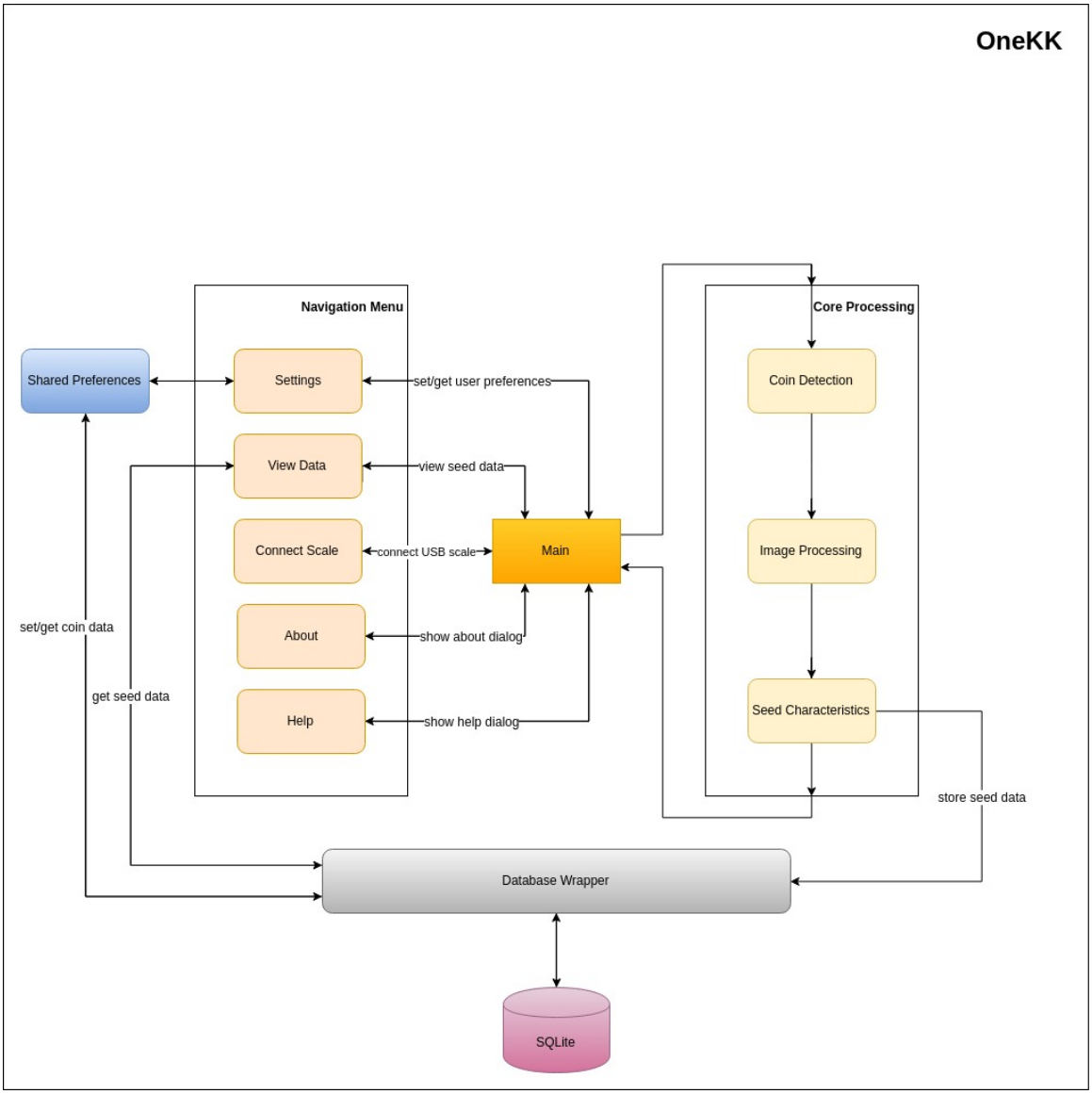

Figure 4.1: *Application Architecture*

4.2.1 User Interface (UI)

The UI functionality is performed using an Android entity called Activity. It is distributed among different activities to keep the work load balanced. An Activity is an Android entity used to perform a single focused task. It is primarily used to implement functionality having some user interaction. The Activity class takes care of creating a window and placing

the UI and its components. Almost every subclass of Activity class implements two methods **onCreate** and **onPause**. The way to start a new Activity is by calling the method **startActivity(Intent).**

There are four activities implemented in OneKK,

1. Main Activity
2. Settings Activity
3. View Data Activity
4. Coin Data Activity

1. Main Activity

Main Activity is the primary thread on which the entire application runs on. It is defined as a **LAUNCHER** activity, indicating that this is the activity that gets triggered when the application is launched. It is also the major UI thread, handling all the operations of rendering components on the main screen. The Main Activity activity diagram is shown in Figure 4.2

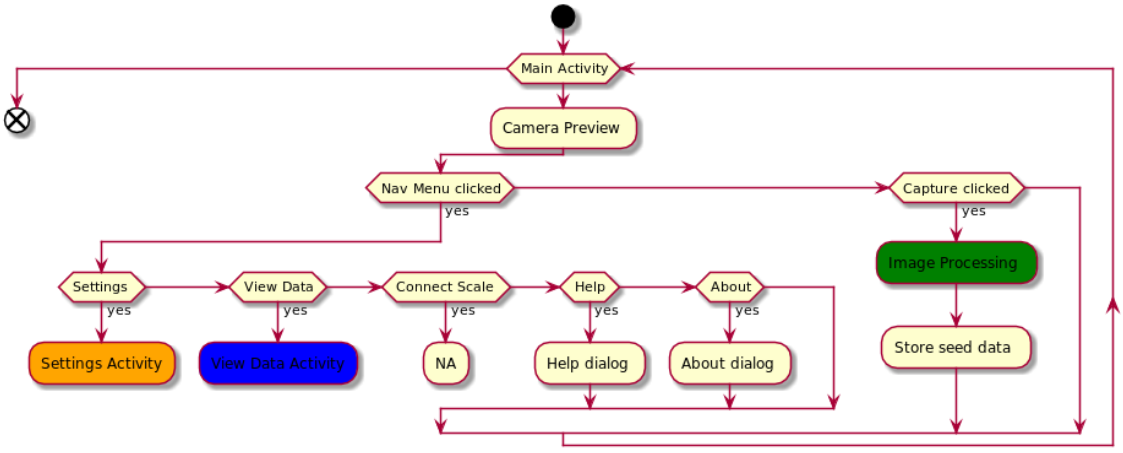

Figure 4.2: *Main Activity Diagram*

As the application is launched, the Main Activity gets triggered and the **onCreate** method is called. This initializes all the UI components and renders them on the main

screen. It contains three major components Camera Preview, Navigation Menu and Image Capture.

- Camera Preview : It is initialized and handled by Main Activity. A one time process on a new install is done, where, it requests the user for permissions to use the Camera hardware. It also sets different camera properties and finally starts the camera preview.
- Navigation Menu : It consists of different user options and is initialized by Main Activity. There are a couple of options like Settings and View Data, which are handled by their respective activities (discussed in later subsections).
- Image Capture : It is a button interface for the user to capture the sample and start the sample processing. The entire processing is handled by a background thread and at the same time interacting with the UI to update the progress. (discussed in later subsection). After the processing is completed the Main Activity uses the database wrapper to store the analyzed seed data in the database.

2. Settings Activity

Settings Activity is another Android activity component which handles all the user Settings in the application right from the first usage of the app. It uses an Android concept called **Shared Preferences** to store and retrieve all settings related to the application. The activity diagram is shown in Figure 4.3

- User Name : It is used by the user to set his/her first and last name.
- Display Analysis Preview : If enabled shows the analysis preview after each sample with the processed image and the seed count.
- Processing : It is a another panel inside the Settings window, used to set options specific to processing. The activity diagram for this is shown in Figure 4.4

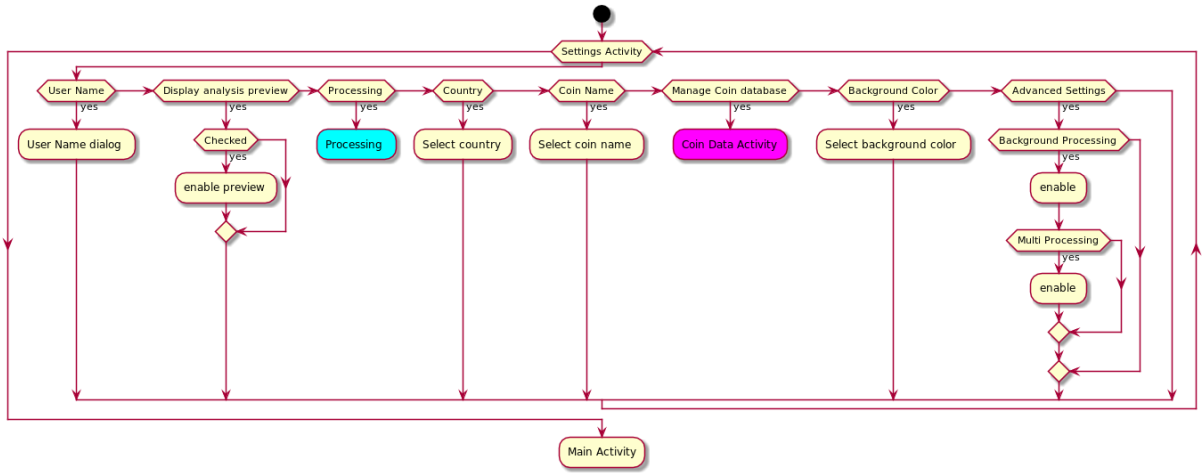

Figure 4.3: *Settings Activity Diagram*

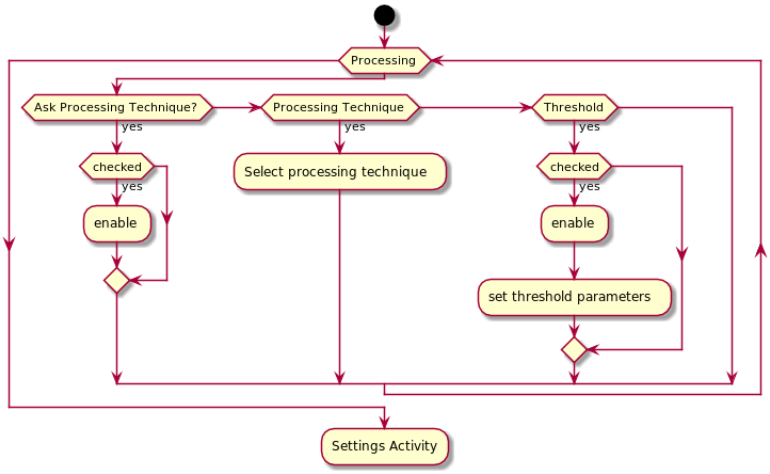

Figure 4.4: *Settings Activity Processing Diagram*

- Country : It is a list of predefined countries retrieved from the database. The user selects a country name, whose coin is used as a bounding marker.
- Coin Name : It is a list of predefined coin names retrieved from the database based on the Country selected. The user selects the specific coin name of the specific country which is used as a bounding marker
- Manage Coin database: This is handled by a separate activity to manage coin related data

- Background color : It lets the user to choose the background color on which the seeds are spread.

- Advanced Settings : These options user to enable or disable Background and Multi processing.

3. View Data Activity

View Data Activity retrieves seed data from a SQLite database using a database wrapper class and populates the seed data table defined in the UI of this activity. It also provides the user with the options to edit, export and delete the table data. The activity diagram of View Data Activity is shown in Figure 4.5

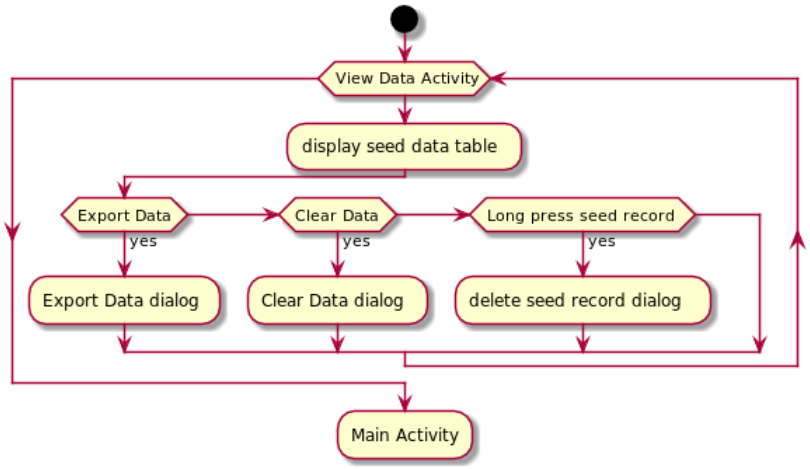

Figure 4.5: *View Data Activity Diagram*

- Seed Data Table : This is a five column table consisting of Sample Name, Width, Height, Count, Weight of each sample that was processed.

- Export Data : This option helps the user to export all the seed related data from the database into a user friendly CSV format.

- Reset Data : The user can reset all the data that has been stored so far in the database.

- Delete record : The user can long press on any of the record and delete that specific record from the database.

4. Coin Data Activity

Coin Data Activity is a child activity under Settings Activity. It primarily handles the operations of fetching or storing coin related information to and from the database. It has a coin table defined in its UI to display the coin data. There are options to add, edit, delete, export and reset coin data.

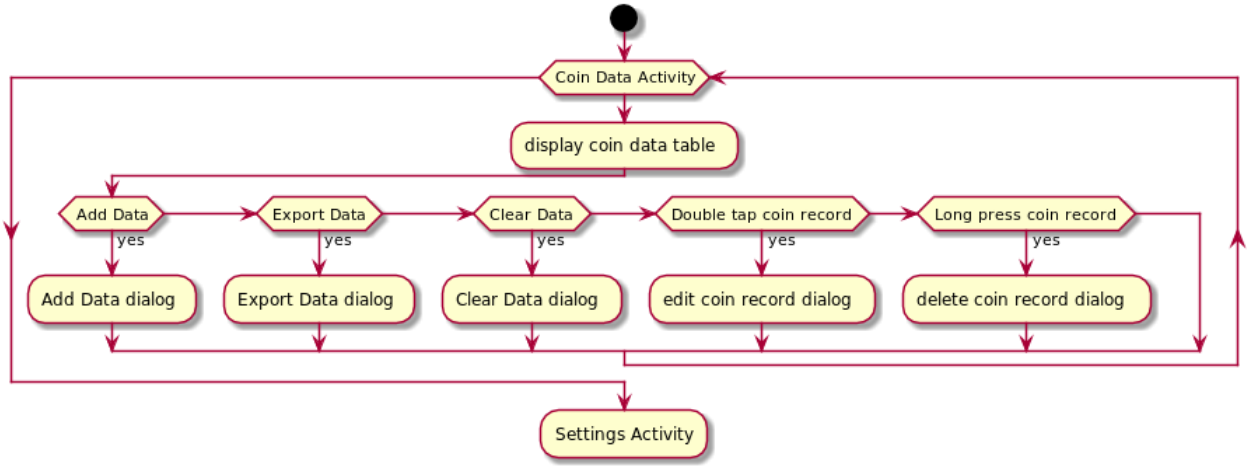

Figure 4.6: *Coin Data Activity Diagram*

- Coin Data Table : This is a three column table consisting of Country Name, Coin Name and Diameter of each coin record in the database.
- Add Data : The user can add new coin data based on the requirement. The user has to provide the country, coin name and the diameter of the new coin.
- Export Data : This option helps the user to export all the coin related data from the database into a user friendly CSV format.
- Reset Data : The user can reset all the coin data to the original version.

- Edit record : The user can double tap on any of the record and edit that specific coin record.
- Delete record : The user can long press on any of the record and delete that specific record from the database.

4.2.2 Core Processing

As, the application has some heavy computing functionality, we've leveraged the android feature of AsyncTask and Thread pool, to move all the computing to a background thread(s) for better performance and a smoother UI.

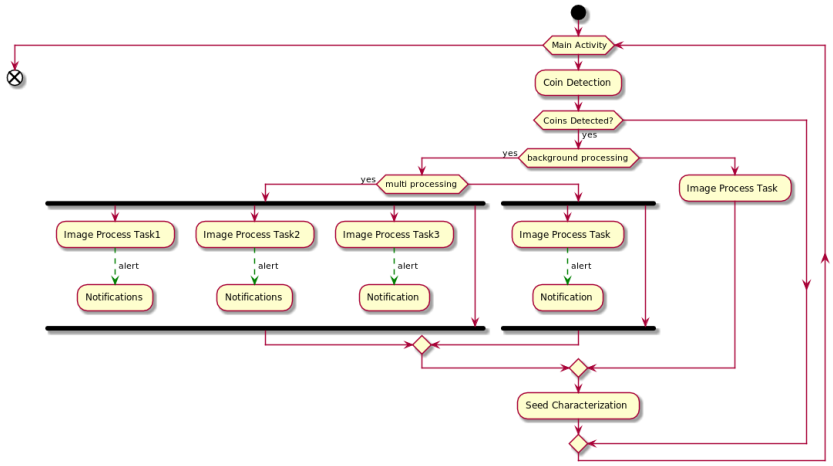

Figure 4.7: *Core Processing Activity Diagram*

The core processing consists of three processes,

1. Coin Detection
2. Image Processing
3. Seed Characterization

1. Coin Detection

The first step is Coin Detection. As the user captures the sample image, which is processed to detect the coins, used to bound the sample search space. The results are used as a pre-check before the actual image processing begins. If all the four coins are detected and are within a specific threshold of acceptance, then the coin parameters are saved (for later use in Seed characterization) and Image processing begins.

2. Image Processing

The application has three modes of processing, where in the user can process

1. One sample at a time
2. multiple samples in background, by queuing them
3. multiple samples in background, at the same time

The maximum number of samples that can be processed at the same time is limited by an Android constraint given by,

$$MAXIMUM\_POOL\_SIZE = CPUCores * 2 + 1$$ [15]

Both the multiple sample processing approaches free up the UI, enabling the user to collect more samples, as others are processed in the background. The progress updates related to each of the sample analysis being done in the background are pushed to the Notification Drawer of the phone. This makes the user aware of the samples analysis that are in queue, running and completed. This type of processing contributes even more to the nature of high throughput phenotyping.

3. Seed Characterization

We use the coin parameters obtained in Coin Detection, to establish a relation between pixels and physical measurements (mm). Then we use the seed related results from Image

Processing and convert them to physical measurements using the earlier relation established. The major characteristics that are extracted and stored are seed count, width, height and area. This data is stored in the database by Main Activity. The data related to user, the captured sample image name and other meta information is also stored in the database.

4.2.3 Database

The database used for the application is SQLite. This is a lightweight database that comes with Android and provides powerful APIs with helper classes to interact with a SQLite database. The database is created as the application is installed and all the necessary tables with their schema are created.The database in the application primarily consists of three tables Sample Table, Seed Table, Coin Table.

The below sections describe about the purpose of each table and the type of data stored in each of them,

1. Sample Table

The sample table is used to store data about each sample that is captured and processed using the app. The primary key, "**id**", in the table is an auto incremented integer value. The other columns are all of type "**TEXT**" and are as follows,

- **sample_id** - is a barcode id for each sample
- **photo** - is the image of the sample captured,
- **person** - is the first and last name of the user who collected the sample
- **date** - is the date when the sample was collected
- **seed_count** - is the seed count in the sample
- **weight** - is the total weight of the sample

- **length_avg , length_var , length_cv** - are the length values of the sample
- **width_avg , width_var , width_cv** - are the different width values of the sample
- **area_avg , area_var , area_cv** - are the different area values of the sample (**avg** - average, **var** - variance, **cv** - coefficient of variance)

2. Seed Table

The seed table is used to store data about each seed in a sample, that is captured and processed using the app. The primary key, "**id**", in the table is an auto incremented integer value. The other columns are all of type "**TEXT**" and are as follows,

- **sample_id** - is a barcode id for each sample
- **length** - is the length of a seed in the sample
- **width** - is the width of a seed in the sample
- **circularity** - is the roundness of a seed in the sample
- **area** - is the area of a seed in the sample
- **color** - is the color of a seed in the sample
- **weight** - the total weight of the sample

3. Coin Table

The coin table stores a predefined list of currency values from different countries used as reference markers during processing. The table is loaded with the required data as the application is installed from a CSV file. The primary key, "**id**", in the table is an auto incremented integer value. The other columns are all of type "**TEXT**" and are as follows,

- country - is the Country name
- primary_currency - is the primary denomination of the currency

- value - is the actual value of the currency
- secondary_currency - is the secondary denomination of the currency
- nominal - is the face value of the currency
- diameter - is diameter of the coin in mm
- name - is the assigned name of a particular coin

4.3 Implementation

The implementation of the application is done in two stages.

4.3.1 User Interface

The components involved in the implementation of the UI utilizes two types of resources, UI components and the respective classes handling the functionality. The UI components are written in XML format in files called "Resource Files" and the classes are written in Java. The application consists of four major screens with which the user can interact

1. Main
2. Settings
3. View Data
4. Coin Data

1. Main

The main screen consists of five UI resources Navigation Menu, Camera Preview, Image Capture button, Sample input and Last sample view. The class handling all the functionality of this screen is Main Activity class. When the app is launched, the Main Activity being

Figure 4.8: *Main Activity Class Diagram*

the launcher activity is triggered. It then initializes all the necessary components and loads it on the screen. The class diagram is shown in Figure 4.8

- **Navigation Menu** - is a UI component implemented using an Android Widget resource called **Navigation View**. In the widget declaration, we also provide another resource file which consists of the menu items. When the app starts the Main Activity class attaches a navigation drawer to the navigation menu and initializes them in a function **setupDrawerContent()**. This method attaches each of the menu item with their respective functionality handler.

- **Camera Preview** - is the UI component which provides the live preview from the

camera hardware. The camera preview is embedded in a frame layout. Main Activity is the handler for all the operations related to this. The methods **startCamera** and **getCameraInstance** are used to get and initialize the camera instance. A package manager is used to set the required camera parameters like focus, rotation, orientation etc. We also define another class called **Camera Preview** which extends Android's Surface View and implements a **SurfaceHolder callback** acting as a wrapper class for the preview. We create a new object for the Camera Preview class and attach it to the application Main screen. We also register a picture callback listener, which handles the processing once the image is captured. The class diagram is shown in Figure 4.8

- **Image Capture Button** - is an Android widget **Image Button**, which is the trigger to capture the sample image and start the processing. The button is embedded in the layout file. It is initialized and loaded in the Main Activity class. The button triggers the picture callback listener, defined as part of initializing the camera. The callback saves the image on the device and then invokes the method for core processing. The processing is carried out on a background thread either blocking or non blocking UI, based on the user settings.

- **Sample Input** - is a user input layout, with two labels and text boxes implemented using Android widgets **TextView** and **EditText**. Two pairs of these two widgets are laid out in a linear layout. The inputs are used for manual user input or barcode entry of Sample Name and the other for the weight of the sample which can also be a manual entry or directly obtained if a weight is connected to the device and is configured in the app. The initializing and loading of the components is done by the Main Activity and the values from these fields are available in the Main Activity itself.

- **Last Sample View** - is a table layout displaying the data of the last sample that was processed. The data consists of two fields, Sample Name and Count. This is implemented using an Android widget **TableLayout**, laid inside a linear layout. The layout is initialized and loaded inside the Main Activity. The data is populated using

the SQLite wrapper class (discussed in later section). After each sample is processed this table is refreshed and the latest data is populated.

2. Settings

It is another user interface screen which provides various flexible option panel, for the user to configure. This is implemented using Android's **Preference Fragment and Preference Activity**. The creation and loading process of the Settings panel is taken care by the class **SettingsActivity** which extends Preference Activity, where as the initialization and definition of the individual preferences in the Settings panel is done in the **SettingsFragment** class which extends the Preference Fragment. The class diagram is shown in Figure 4.9

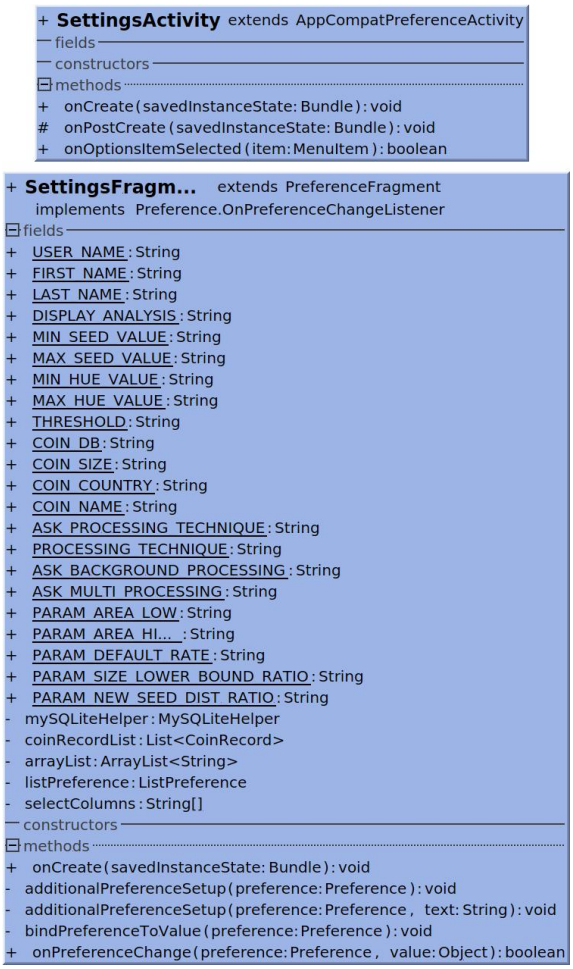

Figure 4.9: *Settings Class Diagram*

A XML resource file is declared using Android's Preference screen specific widgets. This file consists of the layout in which the Settings panel and its components have to be displayed. The Settings panels is created in the SettingsActivity class when the **onCreate** method is triggered. This method then creates a new object of SettingsFragment class.

The SettingsFragment class loads the earlier described XML resource file and initializes all the components inside it. The preferences have some additional setup, where change listeners are overridden to handle the change processing. The SettingsFragment class also used the SQLite wrapper class to load the coin data preferences of Country, Coin Name and Coin Diameter from the database. These preferences also have some additional setup where the Coin Name is loaded based on the selected Country.

3. View Data

This screen provides a table consisting of the sample processing results. This is similar to that of Main screen i.e it is implemented using Android entity Activity. The only difference being, this isn't a launcher activity. This has its own UI screen and respective functionality, which is handled by the class **ViewDataActivity**. The class diagram is shown in Figure 4.10

The entire screen layout is defined in separate XML resource files. The UI screen is laid out using a linear layout, consisting of the Android widgets **Tool Bar, Table Layout and a Nested Scroll View**. The tool bar consists of the screen name, navigation to the previous screen. The tool bar also consists of user options, to export or clear data displayed in the table. The table layout consists of the headers of the data that is displayed on the screen. The nested scroll view is used to load a table layout within it with the data, providing the user with the ability to scroll through the data.

The components and the handling of screen activity is done by **ViewDataActivity** class. This class extends the Android AppCompatActivity class and overrides the method **onCreate**. The layout resource files mentioned above are loaded and the components are initialized inside this method. These menu options are defined in a resource file and are loaded in another overridden method **onCreateOptionsMenu**. This class also uses the

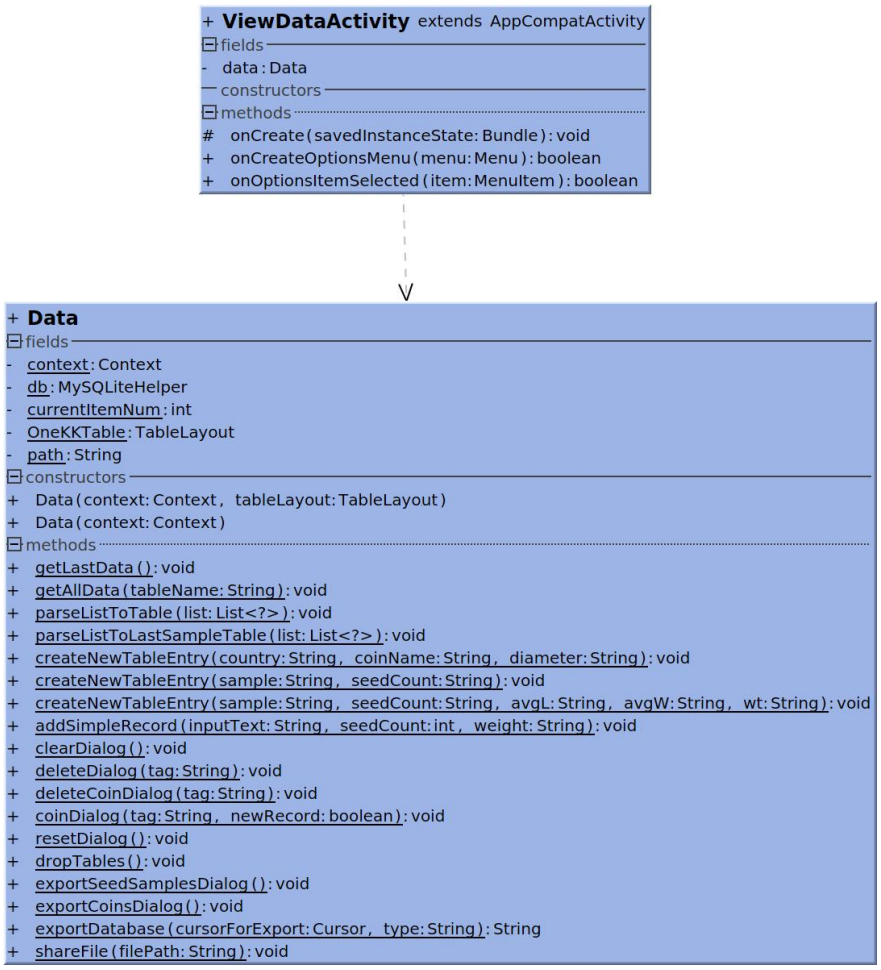

Figure 4.10: *View Data Class Diagram*

SQLite wrapper class to populate the seed data from the database into the table. The user options export or clear are handled by ViewDataActivity itself

4. Coin Data

This is a sub screen of the Settings screen and is used to manage the coin data. It similar to that of Main Activity i.e it is implemented using Android entity Activity. The only difference being, this isn't a launcher activity. It has its own UI screen and respective functionality, handled by **CoinDataActivity** class. The class diagram is shown in Figure 4.11

The implementation is similar to that of View Data screen In addition to that, the menu also has the option to add new coins to the coin list. Each individual coin record added to

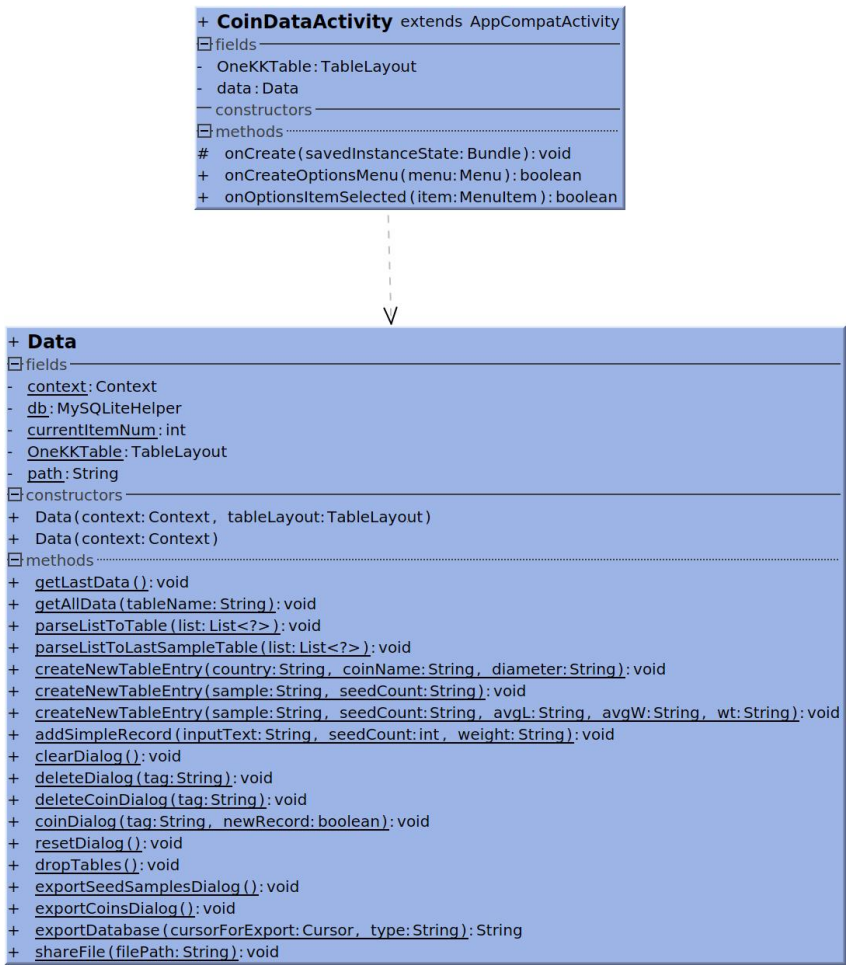

Figure 4.11: *Coin Data Class Diagram*

the table, has a couple of options to it, one is, a long press, which lets the user to delete the record and the second is a double tap which lets the user edit the record.

4.3.2 Core Processing

The core processing is implemented in the **CoreProcessingTask** class which extends the AsyncTask. This class is triggered from the Main Activity after the image capture button is clicked. It involves three steps, Coin Detection, Image Processing and Seed Characterization. Coin Detection is run on the main thread, where as Image Processing and Seed Characterization are run on the background thread. The class diagram is shown in Figure 4.12

The Main Activity can execute the Image processing tasks in three modes,

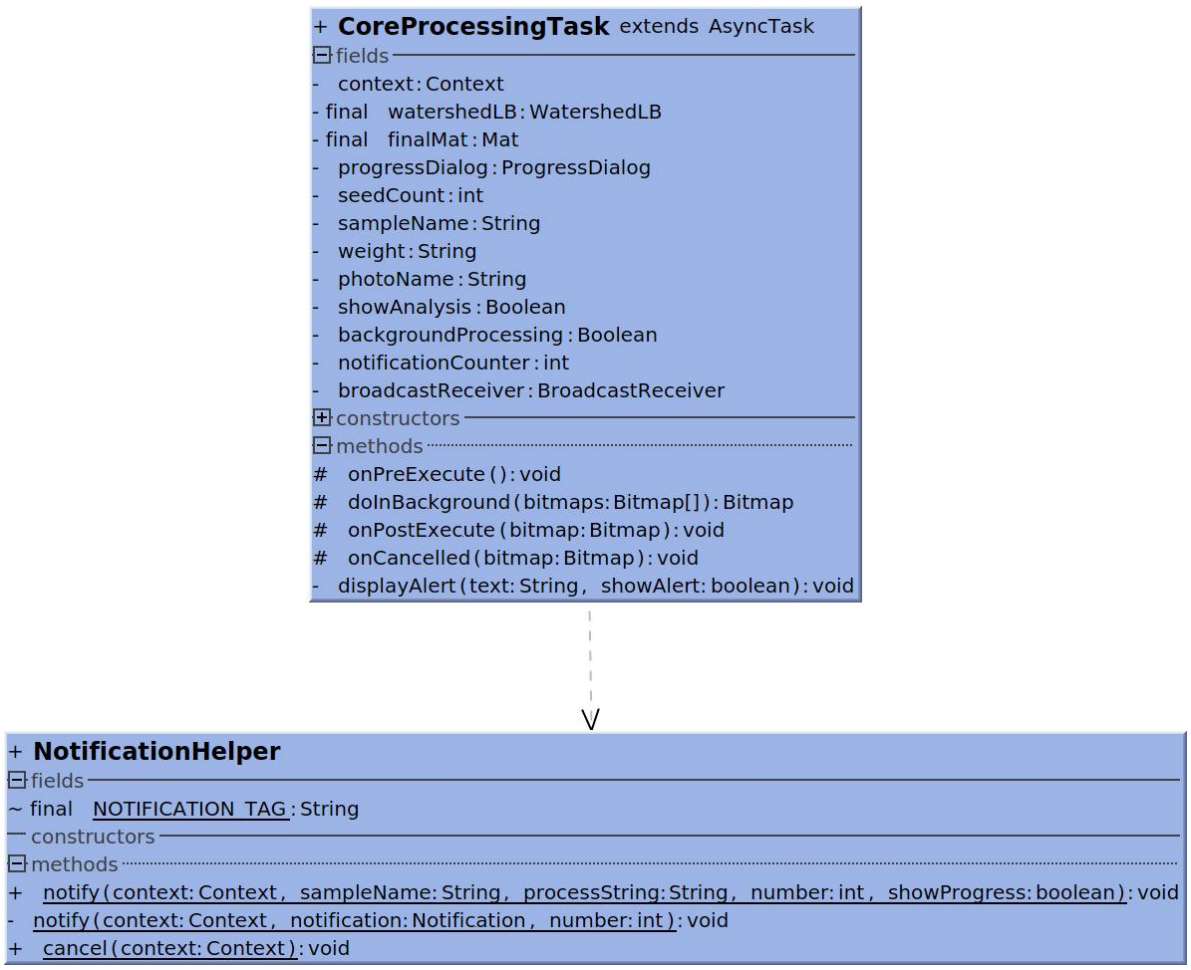

Figure 4.12: *Core Processing Class Diagram*

- **UI blocking, single task, background processing** - In this mode, the UI is blocked and the processing is pushed to a background thread using an AsyncTask. A new UI thread is spawned to display the progress of the process on the UI. Once the processing is complete, the control is returned to the Main Activity.

- **UI non-blocking, single task, queued background processing** - This is a UI non-blocking mode and the processing is pushed to a background thread using an AsyncTask. The Main Activity triggers an AsyncTask, sends the execution on to the background thread and the control is immediately returned to the Main Activity. This lets the user to capture another sample image. In this mode the subsequent processing tasks that are triggered are queued and are executed one after the other in a serial

fashion.

We also have a **NotificationHelper** class which is used to create and display Notifications. The Main Activity creates a NotificationHelper object and passes it to the AsyncTask. This object is used to display the progress of the process, by creating different notifications for different events and displaying them in the Notification drawer. All the notifications are stacked in the Notification drawer and remain there with their final processing status, unless the user manually discards them.

- **UI non-blocking, multi-task, queued background processing** - This mode is similar to that of the previous one, i.e its a UI non-blocking mode with processing done on a background thread and the progress displayed in the form of notifications using the **NotificationHelper** class. The only difference being, at a time multiple tasks can be spawned using Android *THREAD POOL*. The maximum number of multiple tasks that can be spawned at a time is dependent on the device's CPU count. Once this limit is reached the subsequent tasks are queued and executed as threads complete their current processing.

1. Coin Detection

Coin Detection is implemented using OpenCV functions, in a class called **CoinRecognitionTask**. The class also extends the AsyncTask, which is used in implementing the experimental version of real time coin recognition. A **Coin** class is defined to store different properties of the coin like center, radius etc. The class diagram is shown in Figure-4.13.

As the static coin detection begins in **process** method, the image is converted into a mat. The colored mat is then converted into a grey mat. A median blur of size 5 is applied on this grey mat. We then apply an OpenCV function called **HoughCircles**, on this mat to determine the fully circular objects in the mat. This function takes some parameters among which we configure two of them to fit our specific needs. The two parameters are the minimum and the maximum radius of the circles to look for. This is crucial as some of the seeds are circular as well and so we have to make sure we set the right values to just get

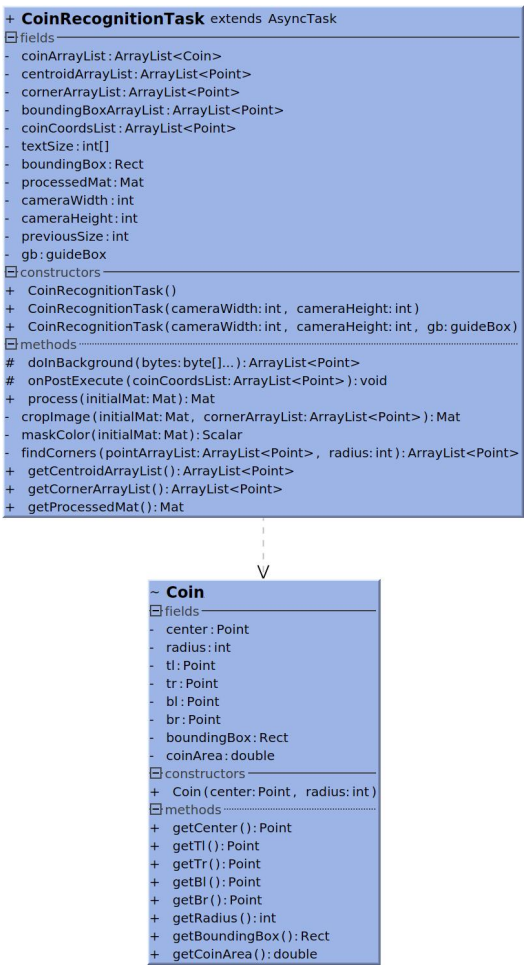

Figure 4.13: *Coin Detection Class Diagram*

the circles of the coins. If the process is unable to detect the four coins, the processing is interrupted and a alert is displayed to the user to recheck the coins.

If the circles of the four coins are determined, we then use another OpenCV function **sumElems** to determine the background color and use the color to mask the coins, to prevent the main algorithm from considering the coin contours in processing. The circularity of the four coin circles is checked to see if its within a specific threshold and accordingly continue the processing or interrupt and send an alert to the user.

The processing continues by taking the centroids of the circles and then sorting them in

the order top-left, bottom-left, top-right and bottom-right respectively. The sorted centroids are used to determine their respective corners. and a new object of such type is created for each of the detected coins. These objects are stored in an Array List. The corners are also stored in an Array List and then are used to crop the image, to limit the search space. If the coins are successfully determined, the cropped mat is returned to the Main Activity. The Main Activity uses this mat and executes background tasks to start the Image processing.

2. Image Processing

Image processing is done in two stages Preprocessing and the actual Image Processing. All the classes related to Image Processing are put in the package **imageprocess**. The primary class used from the package is **WatershedLB**. WatershedLB class does all the heavy lifting and consists of all the methods to perform preprocessing and processing. The class diagram of the classes is shown in Figure-4.14

2.1 Preprocessing Even before the actual processing starts, some preprocessing is done. The Main Activity creates a new object for the WatershedParams class, specifying the various parameters required for processing. This parameters object is then passed as a parameter to the WatershedLB constructor to initiate the preprocessing. As part of preprocessing the input bitmap is converted to a mat and then is converted into a gray mat using OpenCV function **cvtColor**. We then use this gray mat to determine the histogram using the function **calcHist**. The histogram values are used to calculate a threshold value which is used in processing the sample image.

2.2 Image Processing The processing uses a lot of OpenCV functions from the packages Imgproc and Core.

- The gray mat obtained from the previous step is used and a **threshold** function is applied. The minimum threshold value is set to the above calculated threshold value and the maximum is set to *255*. The type of threshold done is *BINARY_INV*.

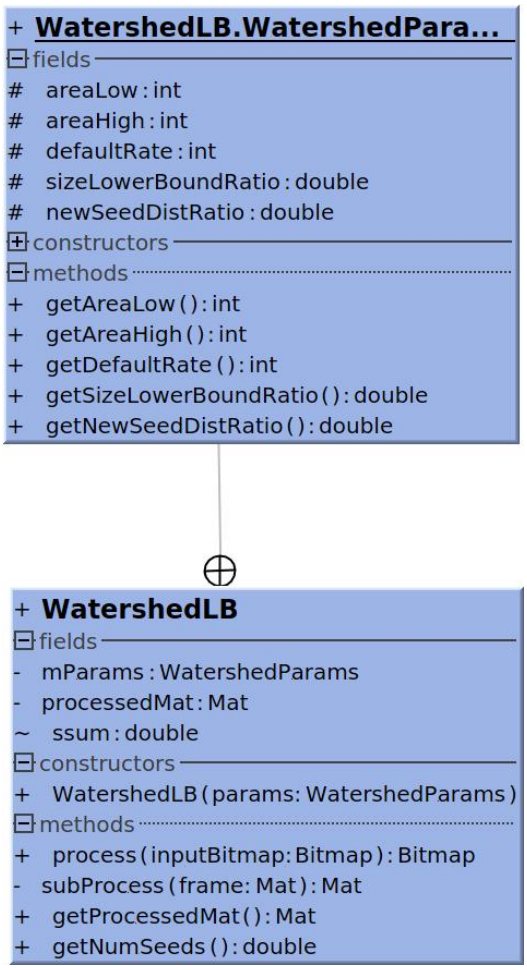

Figure 4.14: *Image Processing Watershed Light Box Class Diagram*

- We then calculate an opening using the threshold mat. An opening is, erosion followed by dilation to reduce the noise in the image. To perform this operation, we use the function **morphologyEx** with the morphology type *MORPH_OPEN* and a kernel size of *3*.

- Sure background is calculated by dilating the opening mat. This is done using the function **dilate** with a kernel size of *3*.

- Distance transform is performed on the opening mat, which is later used to calculate sure foreground. It is calculated using the function **distanceTransform** with a distance value of *2* and a mask size of *3*.

- Sure foreground is calculated by performing a threshold on the distance transform mat calculated above. The **threshold** function is used, with the minimum threshold value set to *8* and the maximum set to *255*.

- The region apart from sure foreground and sure background is called unknown and is calculated by subtracting sure foreground from sure background using a **Core** package function **subtract**

- We then determine all the components that are connected to each other. This is done using the function **connectedComponentsWithStats** and passing sure foreground mat as the input. The function also returns labels of the components, statistics, centroids and also the connectivity value between each of the components.

- The labels are used as markers and passed as a parameter to perform Watershed segmentation using the function **Watershed**. This image is segmented and returns the new segmented labels. We then calculate the unique labels and create a color mask.

- By using the color mask, we find the contours. This is done using the function **findContours**, which takes the parameters, colored mask, contour retrieval method and contour approximation method. In our case we used the contour retrieval method *RETR_TREE* and contour approximation method *CHAIN_APPROX_SIMPLE*.

- We then loop through each of the contour and determine the contour area using the function **contourArea**. We skip the contours with an area less than the minimum threshold and greater than the maximum threshold. The other contours perimeter is also calculated using the function **arcLength**. We store the area and perimeter values in separate array lists and increment the seed counter.

- Using the contour we also determine the moments which gives us the centroid of the contour. This is done using the function **moments**. The centroids are marked on the mat for reference.

- We then estimate the number of seeds again by reiterating and computing the average seed size by checking the contour area within different bounds. We get a better accuracy by iterating for a couple of times. We then adjust the estimated seed area and count accordingly.

- We now implement our proposed extension to watershed. We check if the size is greater than the estimated average size and the perimeter is 1.5 times the estimated average perimeter. In that case we segment the seeds again without re-running watershed.

- We segment the seeds by computing the dot product and getting the absolute value of the dot product and the sum. We check to see if all the dot products are greater than zero and compute the dot sum and update it. We also double check to identify small seeds with relatively large dot products and update the seed counts accordingly.

- The seed count obtained from the above segmentation is validated with the estimate seed count based on the earlier iterative average size calculation. If both the counts match, then we conclude the process else we adjust the seed count based on the latest segmentation count.

3 Seed Characterization

The different characteristics of the seeds are stored and calculated in another class called **Seed**. The Seed class is used to create seed type object for each of the seed that is detected, its class diagram is shown in Figure - 4.15. The parameters extracted from image processing like the area, perimeter and centroid values are in pixels. The coin pixel values extracted from Coin Detection process are co-related to their physical dimensions available in coin data table. We then use this co-relation factor to convert all the seed parameter values from pixels to physical dimensions. These values are then stored in the database using the seed objects.

The class also provides a lot of setter and getter method to set and get different seed parameter values, along with methods to check the shapes of different kinds of seeds.

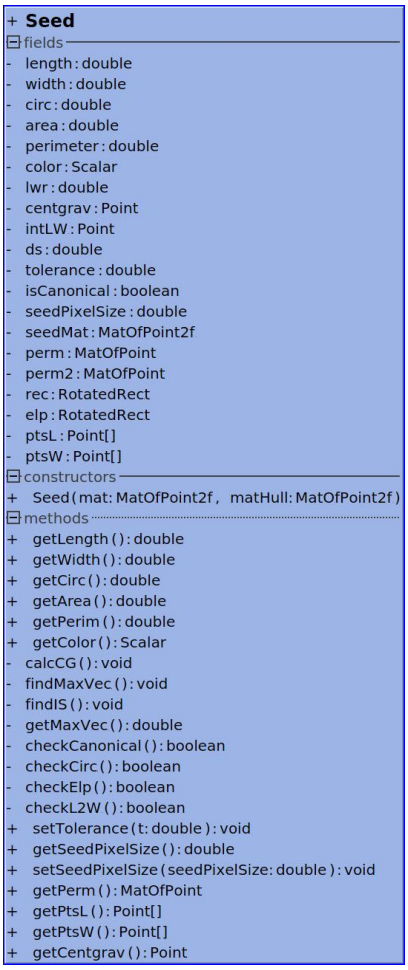

Figure 4.15: *Seed Class Diagram*

4.3.3 Database

MySQLiteHelper

All the database activity is implemented using a database helper class MySQLiteHelper. The class extends the Android's abstract class SQLiteOpenHelper and overrides the function **onCreate**.When the app is installed this method is invoked and the necessary database and tables in it are created. The class diagram of the class is shown in Figure- 4.16.

The following are the detailed description of the methods available in the class,

- **addSampleRecord, addSeedRecord** - these methods are used to store Sample and Seed type records into the database.

Figure 4.16: *MySQLiteHelper Class Diagram*

- **importCoinData** - is triggered to load the coin data into the Coins table after the installation of the app. This method is also used to reset the data to original, if the user has made any changes.

- **coinData** - is a method used to add or update a coin record into coins table. It accepts three parameters,

  record - this is a String array, the value is **null** in case of a new record. In case of an update it consists of the existing coin details

  updates - this is a String array consisting of the new coin details or update coin details whether its a new or update record

newRecord - this is a boolean value indicating whether its a new record or update record

- **exportSummaryData, exportRawData, exportCoinsData** - these methods are used to export the data from the database to a CSV file.

  Summary data consists of the overall sample details along with the user information.

  Raw Data consists of the entire dump of both Seed and Sample tables.

  Coins Data consists of the entire dump of the coin table.

- **getLastSample, getAllSamples, getFromCoins** - these methods are used to get data from Sample and Coins table. The first two methods are used to get the last processed sample from Sample table and all the sample data from the same table. The last method is used to get all the coin data from the Coins table.

- **deleteSample, deleteCoin, deleteAll** - these methods are used to delete data from the tables. The first two methods are used to delete specific record from Sample or Coin table. The deleteAll method is used to erase the entire data from the respective tables.

4.4 Processing phases

The entire process goes through the following phases

1. **Settings**

   (a) set the coin size, this is a mandatory setting. The user selects country and then the coin name.

   (b) set if additional preprocessing is necessary, if capturing the sample image using a normal background and not a light box. The user has to set the background color to be used for thresholding using the color picker

(c) select the type of processing mode, the default mode is UI blocking

2. **Capture**

   (a) align with the grid box shown on the camera preview and make sure all the coins are within the grid space

   (b) the sample image has to be captured with the device exactly above the sample space. In case of any inclination a maximum threshold circularity of 2.5% is allowed

   (c) the user then clicks the capture button

3. **Coin detection**

   (a) coin detection is performed using Hough circles

   (b) the coins are masked using background color picking

   (c) estimate the corners based on centroids

   (d) create coin objects, store the centroids, corners and coin objects

   (e) limit the search space, by cropping the mat using the corners.

4. **Processing**

   (a) in single mode, single task processing is done in the background and the progress is shown on the UI.

   (b) in queued background mode, single task processing is done in the background and the progress is shown in Notification drawer, new tasks are queued.

   (c) in multi queued background mode, multiple tasks processing is done in the background and the progress is shown in Notification drawer, new tasks are queued.

5. **Results**

   (a) user can see each sample processing results immediately after the processing

   (b) the processing results are also available in the View Data table

(c) all the processing results stored in the app's database can be exported to CSV files.

Chapter 5

Dynamic Hopper for real-time algorithm

5.1 Introduction

The static algorithm works accurately for both counting the seeds and phenotyping. The only limitation is that, at a time, it can only process a relatively small sample. This has led us to develop a new approach to capture the seeds as they flow across a simple back lit background. The actual algorithm is described in a recent paper by Neilsen, Courtney, et al., 2017[16] . The summary of the algorithm's functionality is described in Section 5.2

5.2 Process

1. The paper empirically defines that the lowest capture rate is 60 fps (frames/second) for practical use and can also be used at a higher capture rate of 120 fps.

2. Each frame undergoes some preprocessing before the contours are determined using binary thresholding.

3. The current location of the seed is predicted using the previous location and the average

flow rate.

4. As the seeds flow from top to bottom the seeds position is calculated and compared to that of their predicted position.

   (a) If no seed is within the maximum seed distance, which is computed as 4* average flow rate mark it as a new seed and add it to the list.

   (b) Otherwise, the new coordinates are updated, the seed is marked and increment the number of times the seed is updated

5. Each of the seed that is not marked, we age the seed. If the seed is aged more than half the screen height and the the age of the seed is greater than the number of updates, then remove the seed from the count.

6. The number of times a seed should appear in the frame is given by the equation

$$nsfr = height\ of\ frame/ave\ frame\ rate$$

7. The maximum age of the seed which is computed dynamically based on the average flow rate is given by the equation

$$maximum\ age = int(nsfr)/2 + 1$$

8. Compute the average flow rate for all active seeds and the average seed size. The perceived size of the seed may vary as it travels from top to bottom, so we use all samples in our average.

5.3 Implementation

The above algorithm is implemented in a novel and user friendly android app. The app provides the feature to use its own cameras video input stream to get the frames in real-time, provided the hardware supports 60 fps. Otherwise the app can also process video which is recorded with 60 fps, the flowing stream of seeds on a backlit background.

The flow of seeds is controlled using a mechanical hopper built using modularly designed and 3D printed components. The designing is done using open source tools like Blender, OpenSCAD etc and also commercially licensed tools like Cubit which is a modelling software developed by Sandia Laboratories. The models are then exported as STL files and then transformed to gcode using an open source tool called Slic3r. The gcode is then fed to the 3D printer.

5.4 Mechanical Hopper

The printed models are effectively designed using 3D printing fill technique called the Honeycomb to consume less material, yet produce rigid and reliable components. The entire setup is as follows, it consists of

1. Seed bin which is used to feed the bulk amount of seeds
2. Seed track which consists of two gears over which a rubber conveyor belt is mounted and is fixed to the track using two axles. The track also has an extension to hold a NEMA 17 stepper motor fixed to the extension using a 3D printed mount. The motor is used to drive the conveyor belt.
3. A converter is designed and printed to connect the axle and the motor.
4. The motor is driven using a stepper motor driver A4988 which is connected to an Arduino
5. A pot is also connected to the arduino to control the speed.
6. The driver specific code is written and loaded on the Arduino to receive the input from the pot and drive the stepper at controlled speeds with high precision.
7. A trough designed based on the idea of a Plinko effectively distributes the seeds over an area onto the back lit background

8. A 3D printed mobile holder frame is placed over the back lit background to keep the mobile phone steady and ease of use for the user to see the entire processing.

5.4.1 3D printed components

The designed and 3D printed components are Seed Track, Seed Bin, Mobile holder, Trough, Stand and Connector

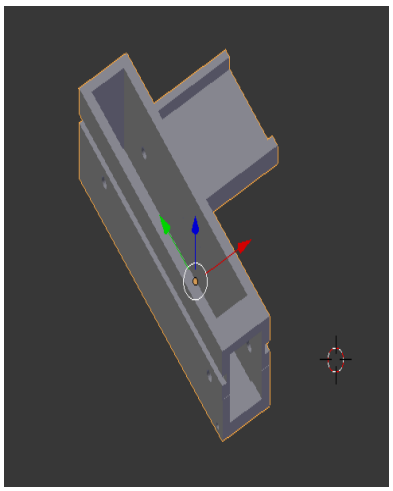

Figure 5.1: *Seed Track*

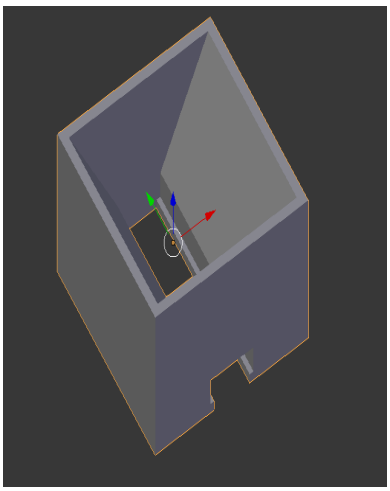

Figure 5.2: *Seed Bin*

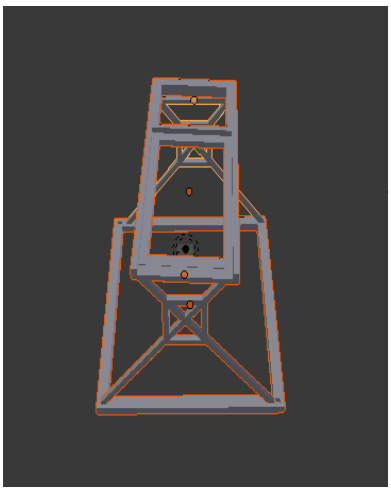

Figure 5.3: *Mobile Holder*

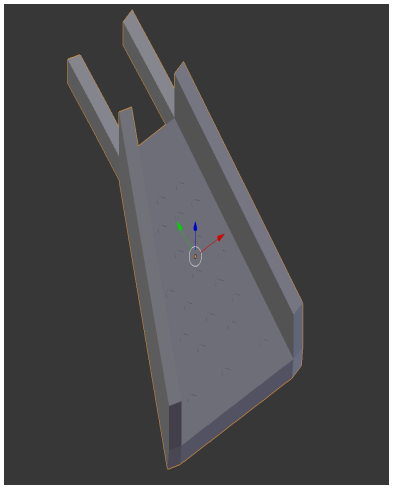

Figure 5.4: *Trough*

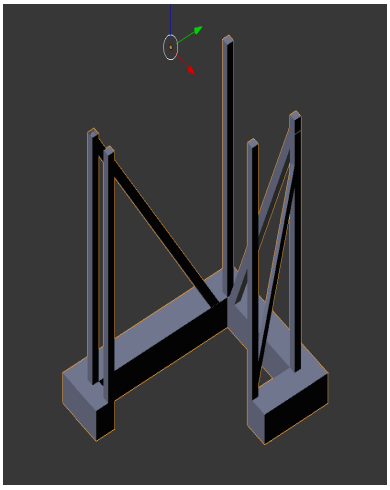

Figure 5.5: *Stand*

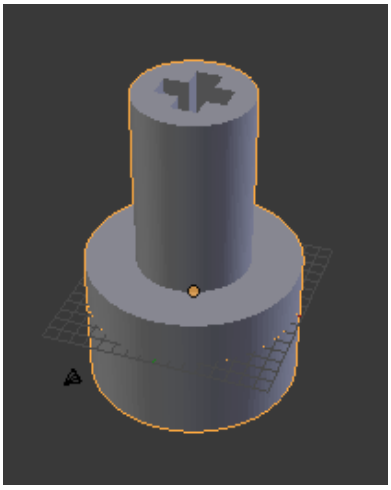

Figure 5.6: *Connector*

5.4.2 Electrical components

The electrical components used in the build are NEMA 17 stepper motor, Arduino 328p, A4988 stepper driver, Power adapter, Pot

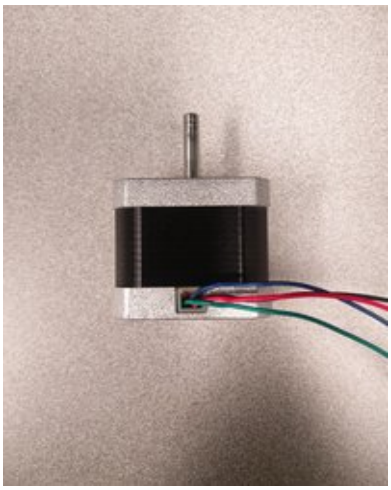

Figure 5.7: *NEMA 17 stepper motor*

Figure 5.8: *Arduino 328p*

Figure 5.9: *Stepper driver*

Figure 5.10: *Power Adapter*

Figure 5.11: *Pot*

5.4.3 Other components

Some other miscellaneous components used are Belt mounts, Back lit board, Axles, Conveyor belt, Gears.

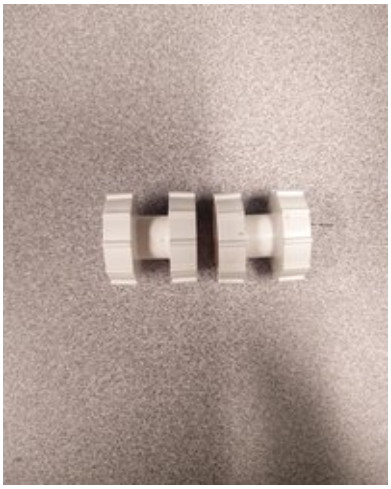

Figure 5.12: *Belt mounts*

Figure 5.13: *Back lit board*

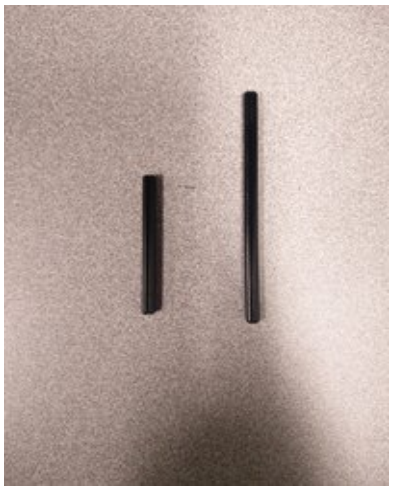

Figure 5.14: *Axles*

Figure 5.15: *Conveyor belt*

Figure 5.16: *Gears*

5.4.4 Assembly

The assembly process is pretty simple due to the modular design principle used in developing the components. The below steps provides the walk through of it,

1. The conveyor belt is mounted on to the mounts, with one of the mounts fitted with the gears as shown in Figure 5.17

2. The setup is then transferred into the seed track, making sure that the mounts with gears go towards the closed end of the track and the axles are connected as shown in Figure 5.18

Figure 5.17: *Conveyor belt over mounts*

Figure 5.18: *Seed track fit with belt, gears and axles*

3. The motor is fixed with the connector and is mounted onto the extension of the track

Figure 5.19: *Stepper with connector*

Figure 5.20: *Seed track setup with stepper motor*

4. The motor is connected to the driver. The pot and the driver are connected to the Arduino as per schematic shown in Figure 5.21

5. The trough is attached to the open end of the seed track and the entire setup is mounted on the stand as shown in Figure 5.23 and Figure 5.24

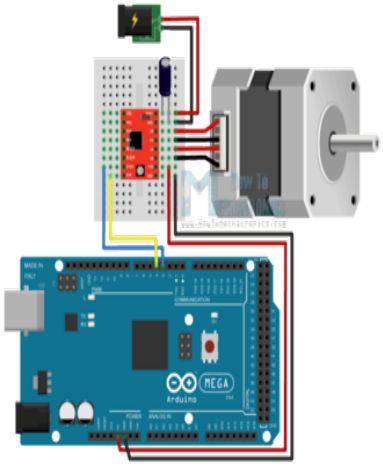

Figure 5.21: *Schematic diagram*[17]

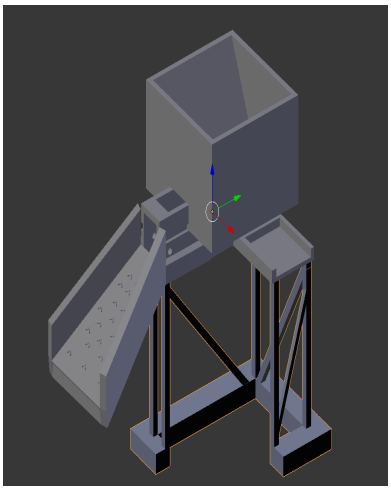

Figure 5.22: *Complete Setup*

Figure 5.23: *Mounted on Stand*

Figure 5.24: *Setup with the Trough*

Chapter 6

Analysis

6.1 OneKK

6.1.1 Testing

The app has been extensively tested using different varieties of seeds varying in color and size. The various features implemented in the application are tested using Android Unit Test cases. These tests are run on the Android device, to make sure all the features work well when installed on a real device. These are implemented using **Android JUnit Test** class. The algorithm in the application has also been extensively tested with a variety of parameters and same seed image samples with varying number of seeds, using **Android JUnit Parameterized** class. The results in analyzing soybeans, poppy, Silphium and Wheat are shown in Table 6.1

6.1.2 Performance

Performance is a crucial factor when building applications for mobile devices. The limited memory and processing capabilities pose even more of a challenge in developing algorithms and implementing them in mobile applications. As one of the principles, this application is targeted at mobile phones starting from a very low price and had to support older devices

Seed Type	EC	AC	RT	AW	AH	AA
Soybeans	31	31	18.43	12.44	11.61	114.4
Soybeans	50	54	30.91	12.54	11.82	117.68
Soybeans	222	217	41.04	12.62	11.52	115.37
Maize	80	79	24.56	21.8	14.08	235.83
Maize	100	100	31.3	21.9	14.3	241.45
Maize	120	119	39.5	21.41	14.1	230.27
Silphium	100	100	13.94	7.24	3	15.54
Silphium	800	807	68	7.3	2.96	15.49
Silphium	2000	2005	157	7.34	3.04	15.79
Wheat	200	202	22.43	10.47	4.41	35.34
Wheat	300	304	29.78	10.08	4.1	31.86
Wheat	900	892	68	10.35	4.12	32.94

Table 6.1: *OneKK Seed Characteristics*
(EC - Expected Count, AC - Actual Count, RT - Run Time (sec) , AW - Average Width (mm), AL - Average Length (mm), AA - Average Area (mm))

running previous versions of Android. In such devices the physical memory can be as low as 512 MB.

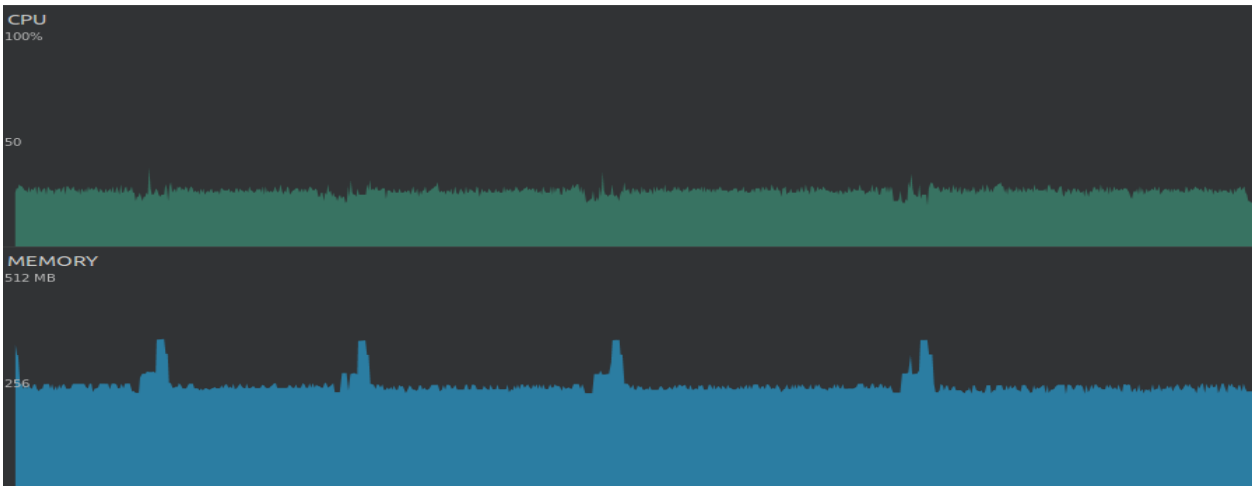

Figure 6.1: *Wheat Performance CPU and Memory*

OneKK has lots of heavy computations involved, with various image processing techniques implemented. It has been implemented very effectively to use less than 512 MB for each of its individual processing. In most cases, it uses at most 256 MB of memory.

This level of performance is achieved even with high resolution sample images of up to

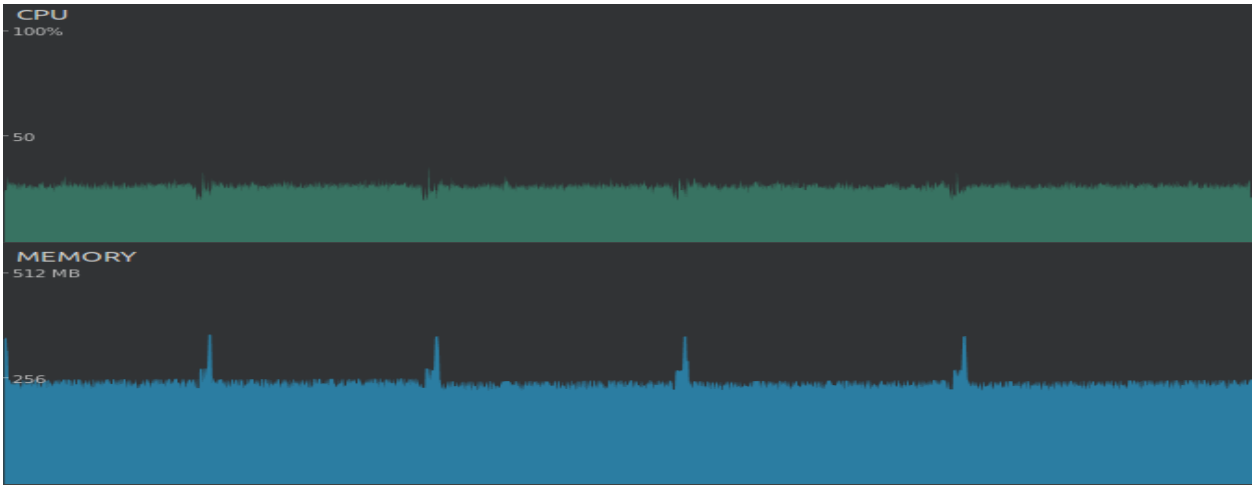

Figure 6.2: *Poppy Performance CPU and Memory*

4K, consisting seeds in excess of 1500. The following are the CPU and Memory performance statistics of various image samples processed using OneKK. The app also gives quick results

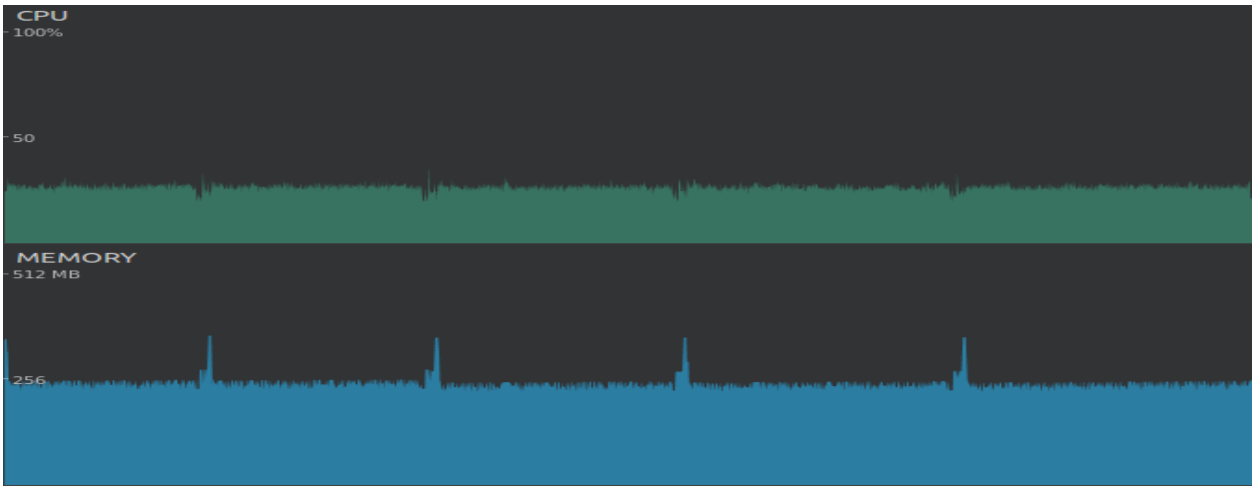

Figure 6.3: *Poppy Performance CPU and Memory*

whether used in single or multi sample processing modes. The stats are taken from five different processing runs of different seed samples consisting of varying number of seeds. The turnaround time for single sample processing for different seeds is shown in Table 6.2.

	Single Task				Multiple Tasks			
Seed Type	ST	ET	RT	TAT	ST	ET	RT	TAT
Wheat	06:05:41	06:06:04	22.410	23	06:01:56	06:02:20	23.608	24
Soybeans	06:05:49	06:06:15	10.925	26	06:02:02	06:02:12	9.134	10
Maize	06:05:56	06:06:34	18.708	38	06:02:09	06:02:26	17.650	17

Table 6.2: *Single & Multiple Tasks Turn-around Times*
(ST - Start Time, ET - End Time, RT - Run Time (sec) , TAT - Turn around time (sec))

6.2 Abacus

The cost of the industrial pneumatic vibrator ranges somewhere between $800 to $1000, whereas the test bench setup we built cost at the most $100 but comparatively has similar flow rates and control mechanisms

The results of the app using different varieties of seeds at different flow rates using both the vibrator and our setup is shown in Table 6.3 (all the values are in seconds)

		Conveyor belt			Pneumatic Vibrator		
Seed Type	Count	Slow	Medium	Fast	Slow	Medium	Fast
Maize	100	6.15	3.43	1.9	18.05	9.39	5.16
Maize	150	8.25	4.92	2.71	19.48	9.8	6.5
Maize	200	10.52	6.21	3.19	23.61	13.45	9.52
Wheat	100	2.55	1.71	1.12	9.81	5.39	2.84
Wheat	200	2.81	1.88	1.46	12.39	6.61	5.27
Wheat	300	3.46	1.97	1.57	13.63	7.35	5.89
Soybeans	100	4.02	2.52	1.45	10.49	5.89	4.23
Soybeans	200	6.2	3.72	1.95	14.45	7.91	5.56
Soybeans	300	9.3	4.81	2.41	18.75	9.71	6.14

Table 6.3: *Mechanical Hopper Seed Rate*

The algorithm has been extensively tested using a variety of seeds and different background setups resulting in accurate results. Even with the video capture resolutions as low as 480p, we were able to bound the error rate under 1%.

Chapter 7

Conclusions

The demand for high yielding crops has, based on the previous results and success of developing novel image analysis algorithms and Android applications for field-based high-throughput phenotyping(HTP), new and better ones are being developed to model and extract plant phenotypes. The thesis describes a new static algorithm and its implementation in an Android application. It also describes the process to build a cost-efficient mechanical hopper used in counting seeds in real time.

The static algorithm is a novel extension of the marker based Watershed segmentation used to segment seeds from a sample image and characterize them. The extension accurately splits clusters of seeds in the image sample. The algorithm is developed to offer a generic processing approach, with the ability to process image samples captured either on a back-lit background or any colored paper. It can efficiently process a wide variety of seeds with different shapes and sizes.

This algorithm is implemented in a Android application called OneKK which is available on Google Play Store, and all open source for the community to use. The application installed on a smartphone, can be used to collect different phenotype data, making it robust and flexible for field use. It also adopts a general approach to let the user use different country coins used as references to characterize the seeds. It provides user flexibility with lots of configurable options, especially for breeders, yet is simple enough to cater a wide

variety of users. The application is thoroughly tested for both accuracy and performance, with seed count accuracy of more than 95% and memory consumption of just 256MB. This makes the application usable even on low-cost devices with lesser memory and computing power.

The Hopper described in the thesis is built using a modular design approach, where each component is designed and printed as separate modules. These modules can be assembled and adjusted to fit different user needs. The electrical components used are of high quality with low cost and power consumption. It also lets you to set the seed rate at which the seeds flow and count them using the application which implements the real-time, dynamic seed counting algorithm. The Hopper's rigidity and robustness is exhaustively tested for usage under different conditions. The seed rates using the Hopper are consistently better when compared to the seed rates using an industrial pneumatic vibrator.

The current work can be extended on various ends. The static algorithm can be extended to characterize more phenotypes and be optimized for better performance. Most modern smartphones have an embedded GPU and have better performance than a CPU. We can exploit this capability to drastically improve the application's performance. The latest API's from Google for Android and Computer Vision provide more advanced ways to implement the application's functionality and extend the dynamic, real-time seed counting algorithm's implementation.

Bibliography

[1] Anthony King et al. The future of agriculture.

[2] Trevor W. Rife and Jesse A. Poland. Field book: An open-source application for field data collection on android. *Crop Science*, 54(4):1624, 2014. doi: 10.2135/cropsci2013.08.0579.

[3] Field book - apps on google play. URL https://play.google.com/store/apps/details?id=com.fieldbook.tracker&hl=en.

[4] Chaney Courtney, Mitchell Neilsen, and Trevor Rife. Mobile applications for high-throughput phenotyping, 10 2017.

[5] Paul Tanger, Stephen Klassen, Julius P. Mojica, John T. Lovell, Brook T. Moyers, Marietta Baraoidan, Maria Elizabeth B. Naredo, Kenneth L. Mcnally, Jesse Poland, Daniel R. Bush, and et al. Field-based high throughput phenotyping rapidly identifies genomic regions controlling yield components in rice. *Scientific Reports*, 7:42839, 2017. doi: 10.1038/srep42839.

[6] Alina Bradford and Stephanie Pappas. Effects of global warming, Aug 2017. URL https://www.livescience.com/37057-global-warming-effects.html.

[7] Smartphone penetration to reach 66 URL https://www.zenithmedia.com/smartphone-penetration-reach-66-2018/.

[8] Jacob Poushter. Smartphone ownership and internet usage continues to climb in emerging economies, Feb 2016. URL http://www.pewglobal.org/2016/02/22/smartphone-ownership-and-internet-usage-continues-to-climb-in-emerging-economies.

[9] Noah Fahlgren, Malia A Gehan, and Ivan Baxter. Lights, camera, action: high-throughput plant phenotyping is ready for a close-up. *Current Opinion in Plant Biology*, 24:9399, 2015. doi: 10.1016/j.pbi.2015.02.006.

[10] Robert T Furbank and Mark Tester. Phenomics–technologies to relieve the phenotyping bottleneck. *Trends in plant science*, 16(12):635–644, 2011.

[11] Gary Bradski. The opencv library. URL http://www.drdobbs.com/open-source/the-opencv-library/184404319.

[12] opencv/opencv. URL https://github.com/opencv/opencv.git.

[13] Mitchell Neilsen, Shravan D Gangadhara, and Siddharth Amaravadi. Extending watershed segmentation algorithms for high-throughput phenotyping on mobile devices, 10 2017.

[14] Circle hough transform, Dec 2017. URL https://en.wikipedia.org/wiki/Circle_Hough_Transform.

[15] Asynctask.java. URL https://android.googlesource.com/platform/frameworks/base.git//master/core/java/android/os/AsyncTask.java.

[16] Mitchell Neilsen, Chaney Courtney, Siddharth Amaravadi, Zhiqiang Xiong, Jesse Poland, and Trevor Rife. A dynamic, real-time algorithm for seed counting, 10 2017.

[17] Controlling stepper motor circuit schematics. URL https://howtomechatronics.com/wp-content/uploads/2015/08/Controlling-Stepper-Motor-Circuit-Schematics.png.

Appendix A

OneKK Samples and Results

A.1 Coin Recognition

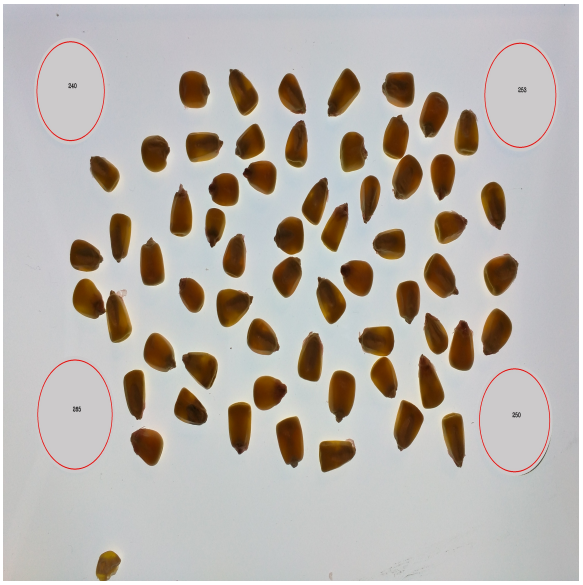

Figure A.1: *Color detection and masking*

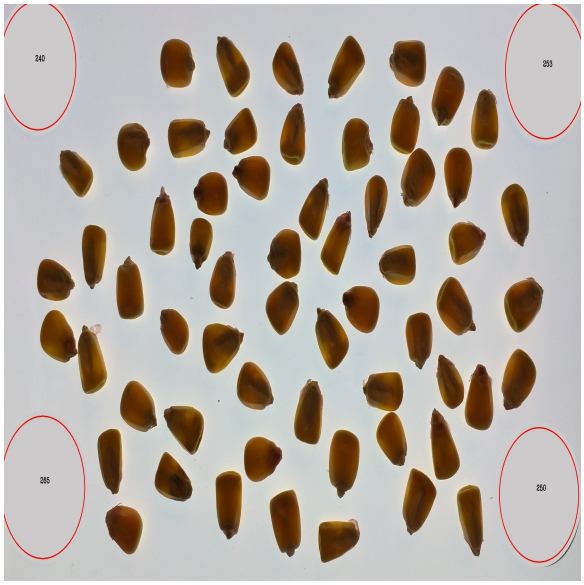

Figure A.2: *Cropping sample image*

A.2 Image Processing

Figure A.3: *Maize sample image*

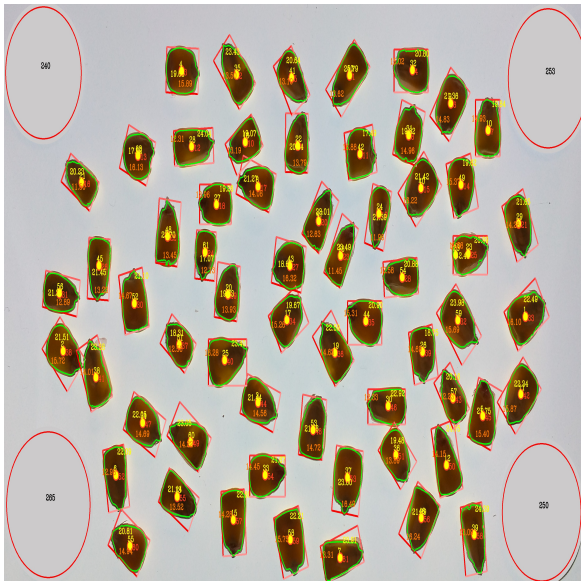

Figure A.4: *Maize analyzed image*

Figure A.5: *Wheat sample image*

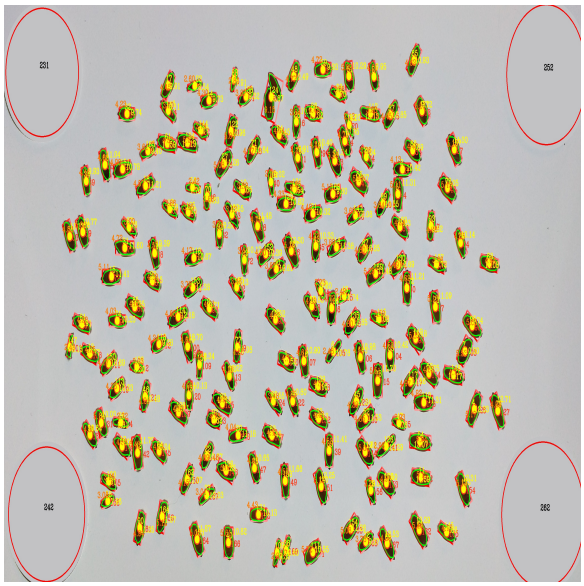

Figure A.6: *Wheat analyzed image*

Figure A.7: *Silphium sample image*

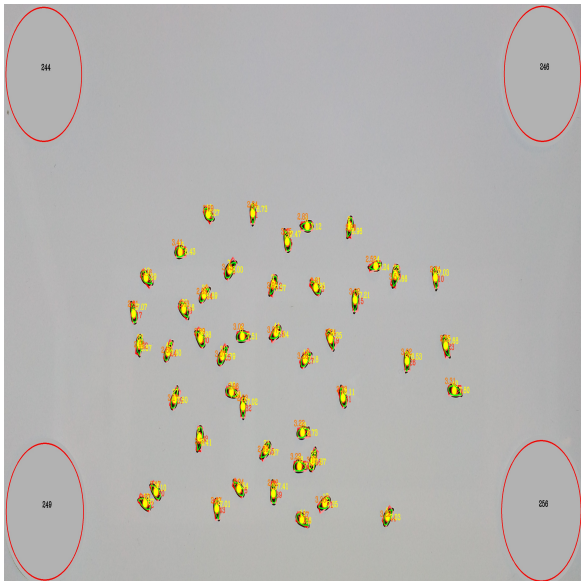

Figure A.8: *Silphium analyzed image*

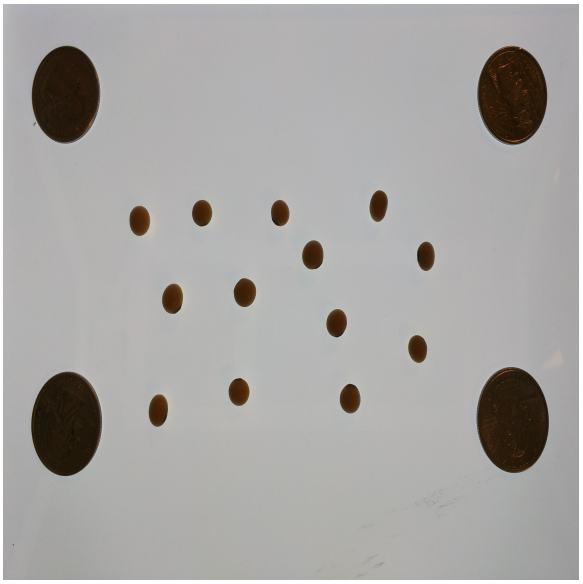

Figure A.9: *Soybeans sample image*

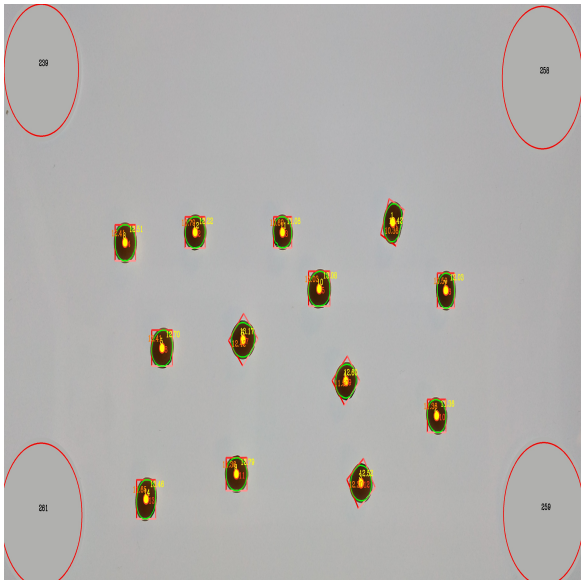

Figure A.10: *Soybeans analyzed image*

Appendix B

Code Snippets

B.1 Core Process Initialization

```
/* MainActivity.java */

public class MainActivity extends AppCompatActivity implements OnInitListener {
.........
@Override
public void onCreate(Bundle savedInstanceState) {
   .........
   .........
      private void imageAnalysisLB(final Uri photo) {
         /* get user settings from shared preferences */

         /* set Watershed parameters to be passed to the actual algorithm */
         final WatershedLB.WatershedParams params = new
             WatershedLB.WatershedParams(areaLow, areaHigh, defaultRate,
             sizeLowerBoundRatio, newSeedDistRatio, 0);
         mSeedCounter = new WatershedLB(params);
```

```
        final CoreProcessingTask coreProcessingTask = new
            CoreProcessingTask(MainActivity.this, mSeedCounter, photoName,
            showAnalysis, sampleName, firstName, lastName, weight,
            r.nextInt(20000), backgroundProcessing, coinSize);

        if (multiProcessing)
        coreProcessingTask.executeOnExecutor(AsyncTask.THREAD_POOL_EXECUTOR,
            inputBitmap);
        else
        coreProcessingTask.execute(inputBitmap);
        data.getLastData();
      }
   }
}
```

B.2 Coin Recognition

```
/* CoinRecognitionTask.java */

public class CoinRecognitionTask extends
    AsyncTask<byte[],AsyncTask.Status,ArrayList<Point>> {

   private double COIN_CIRCULARITY = 0.95;
   private double COIN_SIZE_THRESHOLD = 2.5;
   /* default constructor */
   public CoinRecognitionTask(double coinSize){
      this.coinSize = coinSize;
```

```
    }
    public boolean process(Mat initialMat){
        /* variable initialization and pre-processing */
        Imgproc.HoughCircles(gray, circles, Imgproc.HOUGH_GRADIENT, 1.0,
        (double)gray.rows()/8, //8 change this value to detect circles with
            different distances to each other
        100.0, 30.0, 150, 300); // change the last two parameters
        // (min_radius & max_radius) to detect larger circles
        ........
        // circular mask
        radius = (int) Math.round(c[2]);
        Imgproc.circle(initialMat, center, radius + 20, maskColor(initialMat), -1,
            8, 0 );
        if(checkCirucularity()) {
            if (checkPixelSize()) {
                processedMat = cropImage(initialMat, cornerArrayList);
                return true;
            }
        }
    ......
    }
}
```

B.3 Image Processing

```
/* WatershedLB.java */

public class WatershedLB {
```

```
public static class WatershedParams {
   /* Constructor to initialize the Watershed parameters before processing */
   public WatershedParams(int areaLow, int areaHigh, int defaultRate,
   double sizeLowerBoundRatio, double newSeedDistRatio, double pixelMetric) {
      this.areaLow = areaLow;
      this.areaHigh = areaHigh;
      this.defaultRate = defaultRate;
      this.sizeLowerBoundRatio = sizeLowerBoundRatio;
      this.newSeedDistRatio = newSeedDistRatio;
      this.pixelMetric = pixelMetric;
   }
.......
}
public Bitmap process(Bitmap inputBitmap) {
   .......
   Mat ret = subProcess(frame);
   .......
}
private Mat subProcess(Mat frame) {
   .......
   Mat binMat = new Mat();
   Imgproc.threshold(gray, binMat, thresh, 255, Imgproc.THRESH_BINARY_INV);

   Mat opening = new Mat();
   Imgproc.morphologyEx(binMat, opening, Imgproc.MORPH_OPEN, Mat.ones(new
       Size(3,3), CvType.CV_8UC1));

   Mat sure_bg = new Mat();
   Imgproc.dilate(opening, sure_bg, Mat.ones(new Size(3,3), CvType.CV_8UC1));
```

```
Mat dt = new Mat();
Imgproc.distanceTransform(opening, dt, 2, 3);

Mat sure_fg = new Mat();
Imgproc.threshold(dt, sure_fg, 8, 255, 0);

Mat unknown = new Mat();
Core.subtract(sure_bg, sure_fg, unknown, new Mat(), CvType.CV_8UC1);
.......
int numObjects = Imgproc.connectedComponentsWithStats(sure_fg, labels,
    stats, centroids, connectivity, CvType.CV_32S);
.......
Imgproc.cvtColor(gray, gray, Imgproc.COLOR_GRAY2BGR);
labels.convertTo(labels, CvType.CV_32S);
Imgproc.watershed(gray, labels);

for (Double label : unique) {

Mat mask = Mat.zeros(labels.size(), CvType.CV_8U);
int l = (int) label.doubleValue() + 1;
Scalar colorLabel = new Scalar(l,l,l);
Core.inRange(labels, colorLabel, colorLabel, mask);
List<MatOfPoint> contours = new ArrayList<>();
Mat hierarchy = new Mat();
Imgproc.findContours(mask, contours, hierarchy, Imgproc.RETR_TREE,
    Imgproc.CHAIN_APPROX_SIMPLE, new Point(0, 0));
i = i + 1;
final int countContour = contours.size();
```

```
if (countContour > 0) {
      for (MatOfPoint matOfPoint : contours) {
      double area = Imgproc.contourArea(matOfPoint);
      if (area < minAreaThreshold || area > 32000)
         i = i - 1;
      else {
         final Point[] contourPoints = matOfPoint.toArray();
         final MatOfPoint2f contour = new MatOfPoint2f(contourPoints);
         double perimeter = Imgproc.arcLength(contour, true);
         areas.add(area);
         perimeters.add(perimeter);
         seedCount.add(1);
         cpoints.add(contourPoints);
         ssum = ssum + area;

         final Moments M = Imgproc.moments(contour);
         final double cx = (int) (M.get_m10() / M.get_m00());
         cxCoord.add(cx);
         final double cy = (int) (M.get_m01() / M.get_m00());
         cyCoord.add(cy);

         Rect boundingRect = Imgproc.boundingRect(matOfPoint);
         RotatedRect rotatedRect = Imgproc.minAreaRect(contour);
         String strWidth = String.format("%.2f",boundingRect.width *
            mParams.pixelMetric);
         String strHeight = String.format("%.2f",boundingRect.height *
            mParams.pixelMetric);
         ......
         Mat boxPoints = new Mat();
```

```
Imgproc.boxPoints(rotatedRect,boxPoints);
......
Point midpoints[] = new Point[4];
Point points[] = new Point[4];
rotatedRect.points(points);
for(int v=0; v<4; v++){
Imgproc.line(frame, points[v], points[(v+1)%4], new Scalar(v * 50,v
    * 50,255),5);
midpoints[v] = new Point((points[v].x + points[(v+1)%4].x)/2,
    (points[v].y + points[(v+1)%4].y)/2);
}
......
double midx = Math.pow((midpoints[0].x - midpoints[2].x),2);
double midy = Math.pow((midpoints[0].y - midpoints[2].y),2);
double midDistance1 = Math.sqrt(midx + midy);
......
midx = Math.pow((midpoints[1].x - midpoints[3].x),2);
midy = Math.pow((midpoints[1].y - midpoints[3].y),2);
double midDistance2 = Math.sqrt(midx + midy);
......
strWidth = String.format("%.2f",Math.min(midDistance1,midDistance2)
    * mParams.pixelMetric );
......
strHeight =
    String.format("%.2f",Math.max(midDistance1,midDistance2) *
    mParams.pixelMetric);
......
Rect customRect = new
    Rect((int)points[0].x,(int)points[0].y,(int)Math.min(midDistance1,midDistance
```

```
            ......
            /* draw the contours on the mat */
            Imgproc.drawContours(frame,contours,-1, new Scalar(0,255,0),3);
            ......
            /* draw a circle to mark the center of the seed */
            Imgproc.circle(frame, new Point(cx, cy), 15, new Scalar(0,255,255),
                -1);
            ......
            /* get the text size to write values on the mat */
            Imgproc.getTextSize(String.valueOf(i),
                Core.FONT_HERSHEY_COMPLEX,0.5,1,textSize);
            ......
            /* put a number on each seed, along with the width and height */
            ......
            /* create a seed object for each seed and store them in an
                ArrayList */
        }
    }
}
/* iterate a couple of times to estimate average area and perimeter */
int count = i;
int est_size = (int) (ssum / count);
final List<Integer> bound = new ArrayList<>();
bound.add(0);
bound.add(200);
bound.add((int)(1.5 * est_size));
bound.add((int)(2.5 * est_size));
bound.add((int)(3.5 * est_size));
bound.add(20000);
```

```
ssum = 0;
double psum = 0;
count = 0;
int n = areas.size();

for (i = 1; i < n; i++) {
int seeds = 0;
double area = areas.get(i);
double perimeter = perimeters.get(i);
for (int j = 0; j < 4; j++) {
if (area >= bound.get(j)) {
seeds = j;
seedCount.set(i, j);
}
}
count = count + seeds;
ssum = ssum + area;
psum = psum + perimeter;
}
......
est_size = (int) (ssum / count);
int est_perimeter = (int) (psum / count);
/* detect clusters and split them using dot products */
for (int j = 0; j < nPoints; j++) {
dotProds.set(j, ((xCoords.get(j) - xCoords.get((j+nPoints-delta) %
    nPoints)) * (yCoords.get((j+delta) % nPoints)
- ((yCoords.get(j) - yCoords.get((j+nPoints-delta) % nPoints)) *
    (xCoords.get((j+delta) % nPoints) - xCoords.get(j))))));
```

```
if (dotProds.get(j) < 0) {
dotSum = dotSum - dotProds.get(j);
} else {
if (dotSum >= 300) {

numBig = numBig + 1;
if (dotSum > maxDotSum) {
maxDotSum = dotSum;
}
}
if (dotSum >= 140 && dotSum < 300) {
if (dotSum > nextLargest)
nextLargest = dotSum;
}
dotSum = 0;
}
}
int numSeeds = (int) (1 + numBig / 2);
if (numSeeds == 1 && ((maxDotSum + nextLargest) >= 550)) {
numSeeds = 2;
}
if (numSeeds == seedCount.get(i)) {
//print
} else {
if (seedCount.get(i) < numSeeds) {
seeds = seedCount.get(i);
seedCount.set(i, numSeeds);
......
ssum = 0;
```

```
      for (int k = 1; k < n; k++) {
      ssum = ssum + seedCount.get(k);
   }
   ......
}
```

B.4 Data Operations

```
/* MySQLiteHelper.java */

public class MySQLiteHelper extends SQLiteOpenHelper {
   private static final int DATABASE_VERSION = 1;
   private static final String DATABASE_NAME = "onekkdb";

   public MySQLiteHelper(Context context) {
      super(context, DATABASE_NAME, null, DATABASE_VERSION);
   }
   .......
   @Override
   public void onCreate(SQLiteDatabase db) {
      db.execSQL("CREATE TABLE sample (id INTEGER PRIMARY KEY AUTOINCREMENT,
          sample_id TEXT, photo TEXT, person TEXT, date TEXT, seed_count TEXT,
          weight TEXT, " +
      "length_avg TEXT, length_var TEXT, length_cv TEXT, width_avg TEXT,
          width_var TEXT, width_cv TEXT, area_avg TEXT, area_var TEXT, area_cv
          TEXT)");
      db.execSQL("CREATE TABLE seed (id INTEGER PRIMARY KEY AUTOINCREMENT,
          sample_id TEXT, length TEXT, width TEXT, circularity TEXT, area TEXT,
```

```
        color TEXT, weight TEXT )");
}
.......
@Override
public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
   // Drop older table if existed
   // Create fresh tables
   this.onCreate(db);
}
// Table names
private static final String TABLE_SAMPLE = "sample";
private static final String TABLE_SEED = "seed";
private static final String TABLE_COINS = "coins";
public void addSampleRecord(SampleRecord sample) {
   // 1. get reference to writable DB
   SQLiteDatabase db = this.getWritableDatabase();
   // 2. create ContentValues to add key "column"/value
   ContentValues values = new ContentValues();
   values.put(SAMPLE_SID, sample.getSampleId());
   values.put(SAMPLE_PHOTO, sample.getPhoto());
   values.put(SAMPLE_PERSON, sample.getPersonId());
   values.put(SAMPLE_TIME, sample.getDate());
   values.put(SAMPLE_NUMSEEDS, sample.getSeedCount());
   values.put(SAMPLE_WT, sample.getWeight());
   values.put(SAMPLE_LENGTHAVG, sample.getLengthAvg());
   values.put(SAMPLE_LENGTHVAR, sample.getLengthVar());
   values.put(SAMPLE_LENGTHCV, sample.getLengthCV());
   values.put(SAMPLE_WIDTHAVG, sample.getWidthAvg());
   values.put(SAMPLE_WIDTHVAR, sample.getWidthVar());
```

```
        values.put(SAMPLE_WIDTHCV, sample.getWidthCV());
        values.put(SAMPLE_AREAAVG, sample.getAreaAvg());
        values.put(SAMPLE_AREAVAR, sample.getAreaVar());
        values.put(SAMPLE_AREACV, sample.getAreaCV());
        // 3. insert
        db.insert(TABLE_SAMPLE, null, values);
        // 4. close
        db.close();
    }
    public void addSeedRecord(SeedRecord seed) {
        // 1. get reference to writable DB
        SQLiteDatabase db = this.getWritableDatabase();
        // 2. create ContentValues to add key "column"/value
        ContentValues values = new ContentValues();
        values.put(SEED_SID, seed.getSampleId());
        values.put(SEED_LEN, seed.getLength());
        values.put(SEED_WID, seed.getWidth());
        values.put(SEED_CIRC, seed.getCircularity());
        values.put(SEED_AREA, seed.getArea());
        values.put(SEED_COL, seed.getColor());
        values.put(SEED_WT, seed.getWeight());
        // 3. insert
        db.insert(TABLE_SEED, null, values);
        // 4. close
        db.close();
    }
    .......
}
```