Open source application development for phenotypical data acquisition

by

Chaney Lee Courtney

B.S., Kansas State University, 2014.

A REPORT

submitted in partial fulfillment of the requirements for the degree

MASTER OF SCIENCE

Department of Computing and Information Sciences
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2017

Approved by:

Major Professor
Mitchell Neilsen

Abstract

The Poland Lab at Kansas State University studies the genetics of wheat ‘to develop a climate-resilient wheat variety that can combat rising heat and drought.’ With populations and food demand rising the need for accelerated food growth is imminent. A previous group of Android applications, Field book has shown that the use of open source app development could create a segue to increasing food development through modernized plant breeding across the world. This is especially useful to various countries that may have a limited budget and have a rising market for Android mobile devices.

The applications described in this report include: Verify, a barcode scanning application that can quickly confirm if various seed identifiers are found within a database, Field Mapping, an application that identifies individual plot segments throughout farmland and finally, Survey, an application for manual input of latitude and longitude data. These three applications and services open a new open-source domain for acquisition of data on farmlands to accompany the research of plant breeders.

Acknowledgments

This report could not have been completed without the help of the Plant Pathology Department at Kansas State University. Thank you for the information and help throughout the research of this paper. The research done for this paper was funded by the Bill and Melinda Gates Foundation with the help of my advisor Professor Mitch Neilsen. Also, I would like to give thanks to my committee members, Doina Caragea and Torben Amtoft and all the CIS professors who showed me there are more questions than answers in the world.

Table of Contents

List of Figures

List of Abbreviations

A-GPS = Assisted GPS

API = application program interface

CSV = comma separated values

EOF = end of file

IPC = interprocess communication

s.a = such as

UI = user interface

URL = Uniform Resource Locator

URI = Uniform Resource Identifier

UUID = Universally Unique Identifier

List of Supplemental Files

Repositories:

Verify: https://github.com/chaneylc/Verify

FieldMapping: https://github.com/chaneylc/FieldMapping

Java heap log files:

edu.ksu.wheatgenetics.fieldmapping.hprof

edu.ksu.wheatgenetics.verify.hprof

Memory and CPU Utilization Logs:

fieldmapping_log.png

verify_log.png

Introduction

The project described in this paper is funded by the Basic Research to Enable Agricultural Development (BREAD) program along with the Bill & Melinda Gates Foundation. The purpose of this research is to support smallholder agriculture in developing countries by providing open source software for field-based phenotype acquisition. The project has two focus areas, Developing High Throughput Low Cost Phenotyping Tools and Devices (PHENO) and Advancing Basic Research in Crop Plants Relevant to Smallholder Agriculture (ABRDC). The following paper will describe the development and use-cases of open source applications that focus on these principal areas.

The use of modern mobile services allows for a low-cost distribution of applications. Google Play Store serves as an open portal for app developers to distribute their apps to anyone with an Android device with no cost for hosting. Similarly, the website, GitHub, acts as a host for application source code; the open-source climate of GitHub allows for continuous application growth and code availability. The software described in this report have been developed and stored on GitHub, and will be available on the Play Store market for worldwide availability. The applications coincide with the 1KK application Field Book, a wheat genetics project at Poland Lab planning to reduce development time for plant breeders. Distributing software as open source enables its basic usage to any developing country, which is necessary to satisfy PHENO and ABRDC.

Historically, phenotypical data acquisition has been completed by hand or accompanied by expensive software, which may be unavailable to most developing countries [1]. With the explosion of genomic data and need for increased speed of data collection, applications such as

Field Book have been useful [2]. The purpose of this project is to develop helpful applications for plant scientists while keeping in mind the plethora of different Android devices. Each application targets 100% of Android devices, because of the significant differences between devices the following implementations try to parameterize certain features and offload the most process-intensive tasks to conserve battery life. The following sections will describe the three applications developed for this research, Verify, Field Mapping, and Survey.

Chapter 1: FieldMapping

Field Books APPS-150 Field Mapping App

1.1: Introduction

Android applications typically have a user interface, the buttons, lists, and images that are rendered on the device's screen are handled by what is referred to as the UI thread. This main loop handles button events and view renderings which create the application's layout. FieldMapping is a class developed for the BREAD project that acts as an Android Service rather than an application; therefore, FieldMapping has no UI thread and all processing happens in the background. This data pattern allows other Android applications, s.a Field Book, to use the functionality of the FieldMapping Service. The functionality of FieldMapping is to identify within a set of coordinates which entry is the closest reference from where the user is pointing. This service requires the use of the magnetometer, accelerometer, and GPS unit embedded in Android devices. The magnetometer and accelerometer are used to find the user's bearing in degrees from true North. The GPS uses the default android location service to maintain an accurate location parameterized by the user.

1.2: Implementation

The FieldMapping Service is a subclass of Service and implements SensorEventListener. Service classes mainly require two methods to be overridden: onStartCommand, and onCreate. The onCreate method is called at the beginning of an Android applications lifecycle.

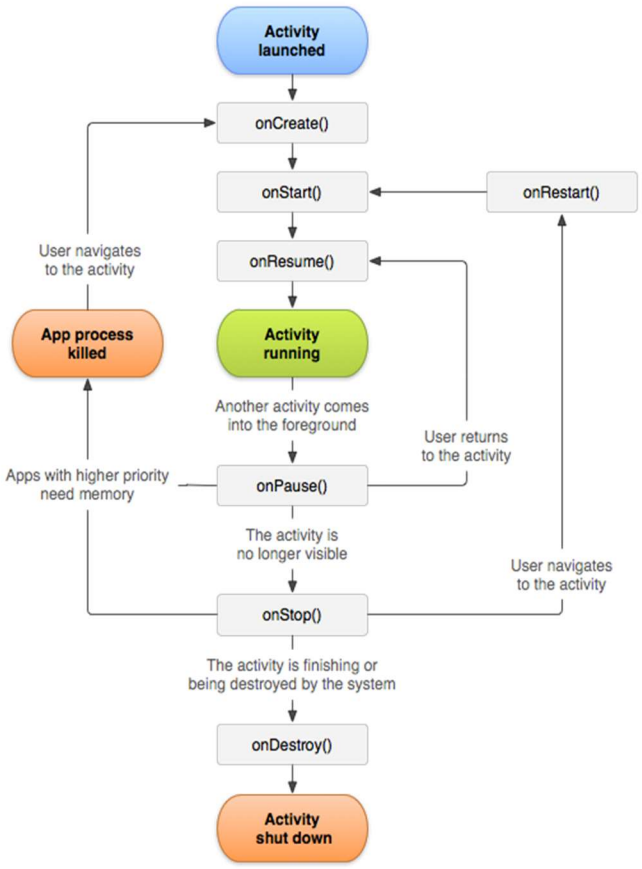

Figure 1.1. An image of android activity lifecycle. [7]

During the call to onCreate, variables are initialized. Some of these variables are sensors that are retrieved from a system service, these are registered with this class to handle events such as when accelerometer values change. FieldMapping implements SensorEventListener to listen for sensor changes and update variables accordingly. Finally, onCreate begins to request location

updates. After finding two accurate locations the application can generate a bearing, which is measured in degrees from true North (0 = North, 90 = East). At least two locations are needed to find the direction of motion of the user, the magnetometer can also be used to acquire the direction a user is facing but this often requires calibration. When onStartCommand is called, it is passed an Intent, which is Android's way of IPC. The user can supply an array, serialized map, or csv file Uri to the given Intent. The Intent may also hold various constraints to tune the listeners including: max accuracy, minimum distance, and minimum time. This type of parameterized implementation allows developers to monitor and adjust usage for battery consumption. These constraints pertain to FieldMapping's AccurateLocationListener which is a subclass of Android's LocationListener. AccurateLocationListener provides the user with a parameterized LocationListener by only returning values that are equally or more accurate than the user requested. Modern GPS units embedded in most mobile phones keep record of signal quality as a coordinate is transferred to the phone [3]. This signal quality, GSA, is available at runtime and is used dynamically in this class. The following method shows this implementation which is called when the minimum distance is traveled and the minimum time has passed.

```
@Override
public void onLocationChanged(Location location) {

    //check if accuracy is below the maximum requested accuracy
    if (location != null)
        if (location.hasAccuracy() && location.getAccuracy() <= _max_accuracy) {
            _prevLocation = location;
            broadcastLocation(location);
            broadcastAccuracy(location.getAccuracy())
```

Figure 1.2. Overriding the location changed event listener.

The Android Location object supplies developers with its accuracy in meters. Mobile devices use A-GPS units which combine network and satellite data for a corrected location. A-GPS units typically have an accuracy of 2-15m [3], which is variable depending on the satellite

constellation and surrounding environment. When an accurate location is found, the Service will broadcast a return Intent that contains the user's location and accuracy, applications that use this Service are required to implement a private class of BroadcastReceiver to accept this message. A complete example application that utilizes this Service is available on the FieldMapping GitHub as FieldMappingServiceTest. Figure 1.3 shows an example of implementing a receiver for this Service.

```
private class ResponseReceiver extends BroadcastReceiver {

    @Override
    public void onReceive(Context context, Intent intent) {

        if (intent.hasExtra(FieldMappingConstants.LOCATION)) {
            final String plot_id = ((Location) intent.getExtras()
                    .get(FieldMappingConstants.LOCATION))
                    .toString();
            if (plot_id != null)
                ((TextView) findViewById(R.id.locationText)).setText(plot_id);
        }

        if (intent.hasExtra(FieldMappingConstants.PLOT_ID)) {
            ((TextView) findViewById(R.id.idText)).setText(
                    (String) intent.getExtras()
                            .get(FieldMappingConstants.PLOT_ID)
            );
        }

        if (intent.hasExtra(FieldMappingConstants.ACCURACY)) {
            ((TextView) findViewById(R.id.accuracyText)).setText(
                    String.valueOf(intent.getExtras()
                        .get(FieldMappingConstants.ACCURACY))
            );
        }
    }
}
```

Figure 1.3. A java subclass of Android's BroadcastReceiver.

After the Receiver is implemented the developer must create an Intent filter to only supply the receiver with the given requests.

```
final IntentFilter filter = new IntentFilter();
filter.addAction(FieldMappingConstants.BROADCAST_LOCATION);
filter.addAction(FieldMappingConstants.BROADCAST_ACCURACY);
filter.addAction(FieldMappingConstants.BROADCAST_PLOT_ID);
LocalBroadcastManager.getInstance(this).registerReceiver(
        new ResponseReceiver(),
        filter
);
```

Figure 1.4. Java code for passing parameters to FieldMapping's location listener.

When the user decides to begin the FieldMapping Service they must supply a set of coordinates for the Service to choose from. FieldMapping has three different ways for the user to do this: supply a csv URI, a serialized map, or a vector of flattened coordinate values. If a CSV URI or an array is given, it must have only a list of comma separated doubles. The order of the doubles correlate to latitude and longitude values given sequentially; e.g. if you have two locations A(lat=-103.4, lng=-96.5), B(lat=-105.4, lng=-96.3) then you would declare an array s.a in Figure 1.4 to create the flattened list of the above values A and B.

```
new double[] {-103.4, -96.5, -105.4, -96.3}
```

Figure 1.5. An example of a flattened array of latitude/longitude values.

To pass an array to the FieldMapping Service, create an intent, pass the double array as an extra, and start the service.

```
final Intent fieldMappingIntent = new Intent(this, FieldMappingService.class);
fieldMappingIntent.putExtra("array", new double[] {1.0, 2.0, 3.0, 4.0});
startService(fieldMappingIntent);
```

Figure 1.6. Passing a flattened array of coordinate values to FieldMapping.

FieldMapping also supports the passing of a map implementation s.a a HashMap. This can be helpful when the user wants to return an identifier for a given coordinate within a set of

references rather than a singular location. An example of this HashMap is given in Figure 1.6, the user passes a sequence of string identifiers and flattened arrays of coordinate values to the map. Utilizing this, the user can query the Service for the closest reference point, e.g: “N” in Figure 1.6 if the user is oriented towards North.

```
final HashMap<String, double[]> idMap = new HashMap<>();
idMap.put("N", new double[] {39.187959, -96.584348});
idMap.put("NW", new double[] {39.187988, -96.584680});
idMap.put("W", new double[] {39.187500, -96.584705});
idMap.put("SW", new double[] {39.187006, -96.584697});
idMap.put("S", new double[] {39.187006, -96.584297});
idMap.put("SE", new double[] {39.187024, -96.583954});
idMap.put("E", new double[] {39.187492, -96.583929});
idMap.put("NE", new double[] {39.187968, -96.583946});
idMap.put("Q1", new double[] {39.187791, -96.584093});
idMap.put("Q2", new double[] {39.187706, -96.584554});
idMap.put("Q3", new double[] {39.187268, -96.584509});
idMap.put("Q4", new double[] {39.187268, -96.584093});
fieldMappingIntent.putExtra("map", idMap);
```

Figure 1.7. Java code for declaring and passing a map to the FieldMapping Service.

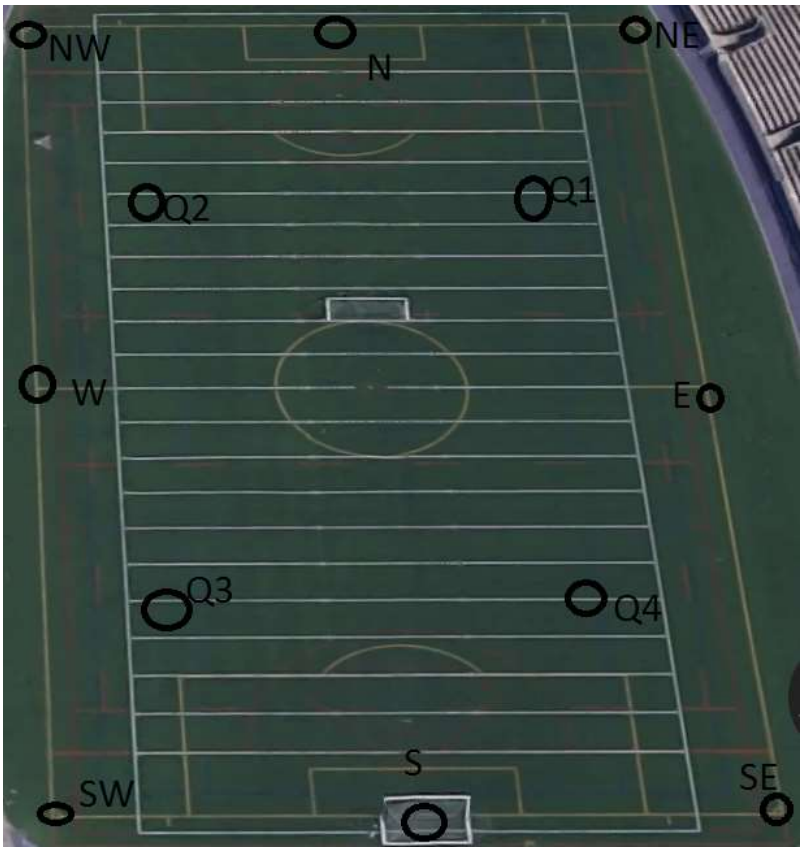

Figure 1.8. Image of locations from Figure 1.7.

1.3: Algorithm Analysis

We propose the following algorithm, impact zone (IZ), for geo-locating reference points within a set of coordinates C. The user's location is represented as U, it contains the user's heading which is measured in radians from true North, and a function bearingTo which calculates the radians between the user and a location. Theta, max and min_distance act as thresholds for distance and radians from the user's heading. This is a greedy algorithm that filters coordinates which do not satisfy the given threshold parameters and returns the closest point which the user is oriented towards. The Haversine method is used to calculate distance on a surface like the Earth's using great-circle distances [4]. Figure 1.3.1 visualizes the IZ the green trapezoidal area shows the actual zone that a reference point is searched for. Figure 1.3.1 shows a user (Southern most marker), and a located reference point (Northern most marker).

Figure 1.3.1. An image visualizing the impact zone algorithm

1.3.1 Impact Zone Algorithm

```
ImpactZone(C, U, theta, max_distance, min_distance)
closest_point = null
closest_distance = INF
for(c=coordinate in C)
        userToC = U.bearingTo(c)
        if (userToC >= azimuth – theta and userToC <= azimuth + theta)
                distance = distanceHaversine(U, c)
                if (distance <= max_distance and distance >= min_distance)
                        if (closest_distance > distance)
                                closest_distance = distance
                                closest_point = c
return closest_point
```

The above algorithm runs in Θ(n) linear for each coordinate in the set C.

1.4: Memory Usage

The FieldMapping Service only uses map objects when the user requests coordinates in that format, otherwise sparse arrays are always used to avoid autoboxing [5].

Android Studio monitors the memory allocations of the given application. The supplemental files section includes hprof files for visualizing heap dumps of the running application. Also included is ‘fieldmapping_log.png’ which is a generated graph from log output of the memory and CPU utilization of FieldMappingTest and FieldMapping Service running. The freeing of memory at 39s and 1m10s may be due to IPC. Maximum memory used in this test application is about 7.6MB.

1.5: Results

An experimental application was built to use and test the FieldMapping Service. The application is available on GitHub along with the FieldMapping Service. This activity is called FieldMappingTest, and is shown on the right. It is a simple utilization of the FieldMapping Service, showing the plot id calculated from the impact zone algorithm. For this test four ids were programmatically created using the LatLngUtil class provided. The method used to project points using a heading and distance and a combination of other trigonometric functions are found in this class [4]. The screenshot on the right shows the user oriented towards the NE plot while the GPS maintains an accuracy of 22m. Future developments may use a sparse array in replacement for the user inputted map, hashing the plot id and using a String array may reduce memory usage.

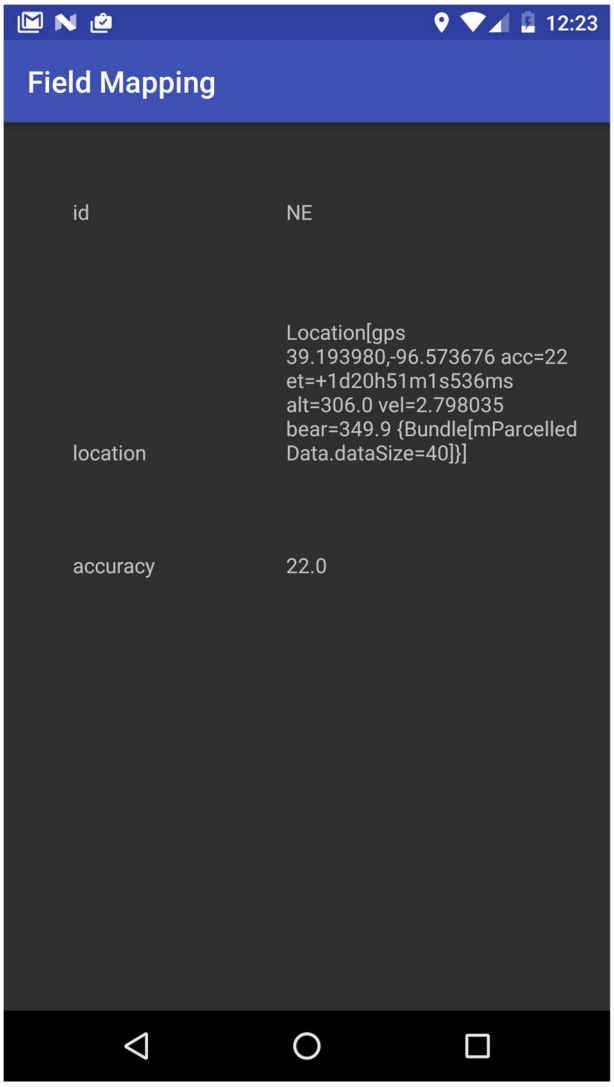

Figure 1.5.1. An image of FieldMappingTest application.

Chapter 2: Verify

Field Books APPS-169 Verify

2.1: Introduction

With the use of modern machine vision APIs, Android devices can function as an image detector, text reader, or even barcode scanner. Verify is an Android application that uses Google's new MobileVision library to detect scanned barcodes and check whether it exists in a set of uploaded codes. Google's barcode API can scan any of the following types of barcodes: EAN-13, EAN-8, UPC-A, UPC-E, Code-39, Code-93, Code-128, ITF, Codabar and also supports the following 2D barcodes: QR Code, Data Matrix, PDF-417, AZTEC (Google Barcode API). Verify specifically has direct engagement with international partners in India. Plant pathologists in India are currently limited to checking barcodes by hand, with a large set of plant identifiers spanning 9-digits this can be a daunting task. The objective of Verify is to simplify this by utilizing modern APIs with handheld devices.

2.2: Implementation

Verify has two main activities, CameraActivity, and MainActivity. The main activity allows the user to input data via a CSV file and check whether a given UUID is within the uploaded set. The purpose of this activity is for the user to give text input (essentially a UUID) and gain references to its data or check if the input is within the given set. The user may directly connect a barcode scanner using the USB port, or input data manually using the device's keyboard. Barcode scanners that connect directly to Android devices act as keyboards and the

laser-scanned barcode UUID is passed as text to the focused text view. Our algorithm accepts all types of barcode values and uses the Google API to interpret the raw value of the code which is typically an integer. Verify's MainActivity uses Android's TextWatcher class to activate a lookup function whenever text is changed on the UUID user input. Using TextWatcher and a HashMap to store columnar data allows for immediate response times when checking barcodes.

The implementation for MainActivity utilizes an asynchronous task to offload processing from the UI thread. The task is a subclass of the AsyncTask interface. AsyncCSVParse is used when the user provides a CSV Uri, the asynchronous thread creates a BufferedReader and processes the given Uri.

```
new AsyncCSVParse().execute(_csvUri);
```

Figure 2.2.1. Showing invocation and execution of AsyncCSVParse.

The file is processed by first finding the header line, which it assumes to be the first line of the file. Secondly the file is read until the EOF is found, the algorithm expects comma separated values where the UUID is the first column. All other columns found in the file are saved in a map, implemented as a HashMap for fast retrieval later. Future developments may include more sophisticated techniques for finding plot id headers or additional UI for handling header parsing customization. As text is input into the scanner user input field, the value is queried for inside the map to find stored columns. AsyncTasks have the option of overriding a method which posts information to the UI thread. AsyncCSVParse overrides the method in Figure 2.2.2 and fills the main ListView of MainActivity with the parsed UUIDs.

```
@Override
protected void onPostExecute(String[] headers) {

    final ListView lv = ((ListView) findViewById(R.id.idTable));
    if (headers != null && headers.length > 0 && headers[0] != null) {
        final ArrayAdapter<String> idAdapter =
                new ArrayAdapter<String>(_ctx, R.layout.row);
        for (String id : _idMap.keySet()) {
            idAdapter.add(id);
        }
        lv.setAdapter(idAdapter);
    } else lv.setAdapter(new ArrayAdapter<String>(_ctx, R.layout.row));

}
```

Figure 2.2.2. Code for overriding onPostExecute which will post data to the UI thread.

```
<?xml version="1.0" encoding="utf-8"?>
<TextView xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/row"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:padding="10dp"
    android:textSize="16sp" >
</TextView>
```

Figure 2.2.3. Xml definition of R.layout.row.

The Camera activity is a simply designed layout that utilizes the device's camera to render a preview to the user as if they were taking a picture. CameraActivity is a subclass of AppCompatActivity and implements SurfaceHolder.Callback to create this preview functionality. SurfaceHolder.Callback is an interface that defines functions for the lifecycle of a given Surface, in this case a camera preview. Simply, the camera object begins its preview when the surface is created, and stops when the surface is destroyed. The camera object uses Google's CameraSource.Builder to create an auto-focusable camera viewing at 30 fps. Barcodes are read by implementing an inner class that uses Google's Detector.Processor, the algorithm used within the detector defers barcodes that were previously scanned. Specifically, the same barcode can be scanned ten times before another notification is sent. The CameraActivity defines another

asynchronous task, AsyncIDCheck to offload UUID checking to another thread. AsyncIDCheck linearly checks the UUIDs passed via an intent.

Rather than taking a picture, the user must simply hover their camera over a given barcode until it is detected. When the code is detected a temporary message icon displays on the bottom of the screen, if the message is found within the given set of UUIDs a notification is sent to the device. The notification has a visual and audible representation, the sound is the Android's default notification sound, and the visual representation is a text notification on the device's task bar. A code that is detected but not found in the given UUID list still gives a notification, but notes that it was not found in the list and shows the found UUID.

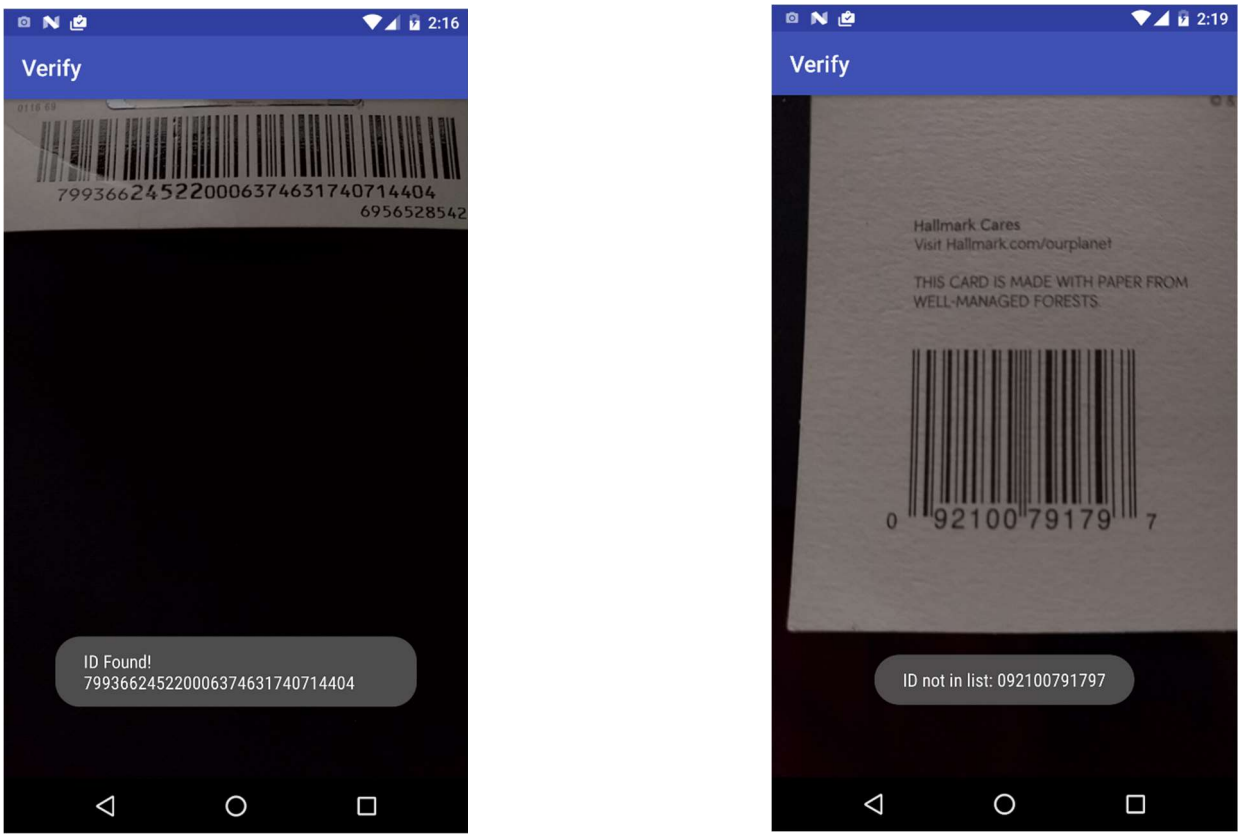

Figure 2.2.4. Two images showing successful and unsuccessful scan attempts with Verify.

2.3: Layout

Figure 2.3.1 shows a Nexus 5X device running the Verify application. The first table represents the ListView used, each entry is an uploaded row found in the CSV file. The top right two buttons show the upload button, and the camera activity switch button. When the upload button is pressed a context switch occurs to allow the user to upload a file via their device storage or Google drive. After the ListView is a EditText which allows user input via the keyboard or barcode scanning device, this view is focused by default. The bottom segment is a TextView which is used to render columnar data. The camera activity layout is a full-screen camera that renders notifications as UUID's are scanned. By default, all views are empty, the user must first upload a CSV file and continue with text input to verify their UUIDs.

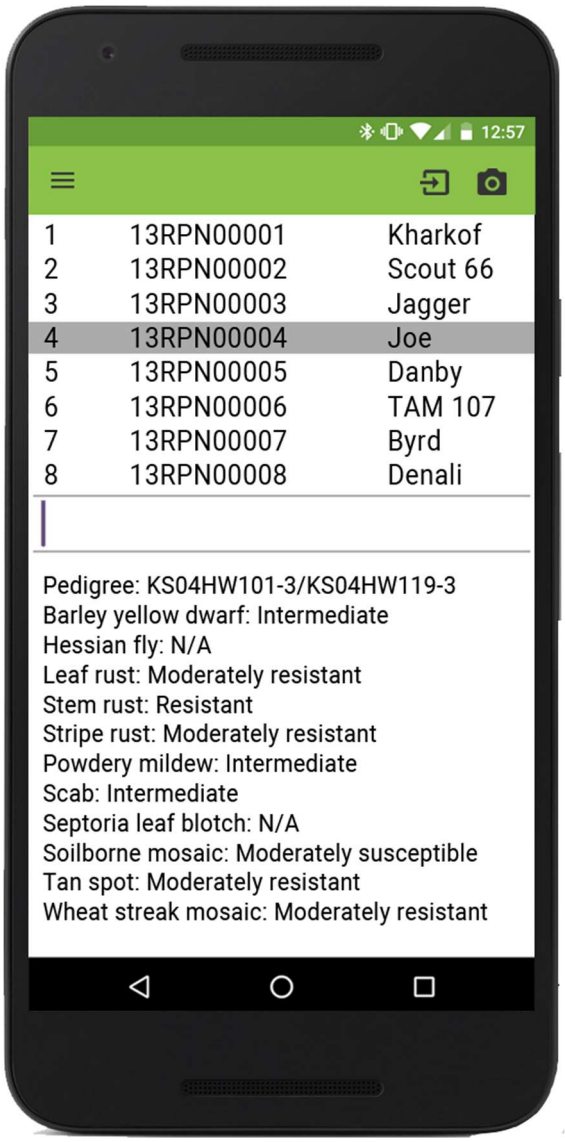

Figure 2.3.1. A mockup of the Verify application.

2.4: Algorithms

When CameraActivity begins and its onCreate method is called an array of identifiers are passed from the parent activity, MainActivity. These identifiers are used within AsyncIDCheck by comparing the scanned value with the given entries. The UUIDs are checked linearly, if an identifier matches a notification is sent to the device's screen. To suppress multiple-similarly detected code notifications, the following algorithm is used within the detector's processor:

2.4.1: Detect Algorithm

```
Detect(B, prevValue)
        sameCount = 0
        if (B.size != 0)
                code = B.valueAt(0)
                if (!code.rawValue.equals(prevValue))
                        prevValue = code.rawValue
                        sameCount = 0
                else sameCount++
                if (sameCount == 0 || sameCount > 10)
                        sameCount = 0
                        new AsyncIDCheck().execute(prevValue)
                        notifyUser()
```

2.4.2: Detect Algorithm Analysis

The algorithm has an upper bound of O(n), because AsyncIDCheck.execute will check each identifier linearly within a separate thread. The algorithm has a lower bound of Ω(1) when the check is differed. B refers to a sparse array of barcodes, the first element is always chosen. B is an array because Google's MobileVision can detect multiple barcodes at the same time, the algorithm assumes the user situates the camera over a singular barcode. The code object contains references to scanned values, e.g: QR codes may contain a URL and canned food barcodes may contain a unique identifier. However, only Barcode.rawValue is used for data acquisition. As the algorithm describes, a count is kept of how many times the same barcode is scanned, this helps when the user leaves the camera over a code the notifications may be deferred, while at the same time giving immediate response for newly detected codes. Finally, the notifyUser function builds an object that displays an operating-system default notification, this analysis assumes notifyUser operates in Θ(1) constant time.

2.5: Memory Usage

The supplemental file 'verify_memory.png' contains a graph representing the logged output of Android Studio while running Verify on an actual Nexus 5X device. Logging begins at 1m50s as the MainActivity is opened; as the user inputs data, there are small spikes in CPU usage. CPU usage gradually increases as the CameraActivity begins at 2m4s, the camera is utilizing the CPU. CPU usage declines after the back button is pressed and the CameraActivity is destroyed, calling the MainActivity. At 2m41s the GC frees data from the CameraActivity.

2.6: Results

Further improvements may include the usage of Zebra's library to listen to wireless barcode readers. Prototypical implementations were slightly altered for the result. First implementation used a ListView to render columnar data after a given UUID was verified. ListViews require TextView objects to be instantiated for each entry in the list using a ListAdapter<String>, swapping the ListView for a single TextView with concatenated string data using StringBuilder showed drastically smaller memory usage. Initially there was a third asynchronous task to render data inside the ListView (which held columnar data values read in the CSV file), this was replaced by a map implementation to store columnar data. The idea behind the third asynchronous task, AsyncTableUpdate was to conserve memory by not always holding table values and only render

data as the user requested it. The first resulting application was a bit slow for each lookup, it was replaced by the map implementation because of value lookups are constant in time complexity, $\Theta(1)$. Future features include entry deletion on verification, and aesthetic changes to the UI design.

Figure 2.6.1. Verify rendering column data.

Chapter 3: Survey

A Field Books prototype

3.1: Introduction

Survey is an Android application using Google Map's API to mark and save latitude and longitude values. Modern data acquisition for precise agricultural equipment use expensive surveying equipment to retrieve precise data. This technology is not available around the world and can be moderately replaced using this application. Therefore, this software will enable smallholder farmers with the ability to label precise locations on a map. This type of basic functionality is a stepping-stone for later uses s.a for phenotypical analyses and other plant based research.

3.2: Implementation

Survey is an Android application subclass of FragmentActivity. Survey implements the following interfaces: GoogleMap.OnMapClickListener, OnMapReadyCallback, GoogleApiClient.OnConnectionFailedListener, GoogleApiClient.ConnectionCallbacks, LocationListener. To retrieve the user's location there is a typical GPS location button in the top right of the device's screen, this will orient a view over the user, show a bearing, and will overlay a radius of the accuracy in meters.

The application utilizes the FusedLocationProviderApi from the GoogleApiClient which automatically chooses between providers to lower battery consumption and maintain an accuracy. The choice in accuracy is limited, Android has permissions to access fine location or

coarse locations. The application uses both FusedLocationProviderApi and Android's default LocationListener to provide coordinates if either are unavailable. The GoogleMap.OnMapClickListener interface is implemented by creating an entry into a sparse array and rendering the location as an 'x' on the given map. The OnMapReadCallback is implemented by creating a function that is called when the MapFragment is attached to the activity. When the map is ready the Google Map is instantiated and orients over Kansas State University's campus. The map begins to listen for user input and renders 'x's on the screen where the user has clicked. These reference points can then be saved in a table.

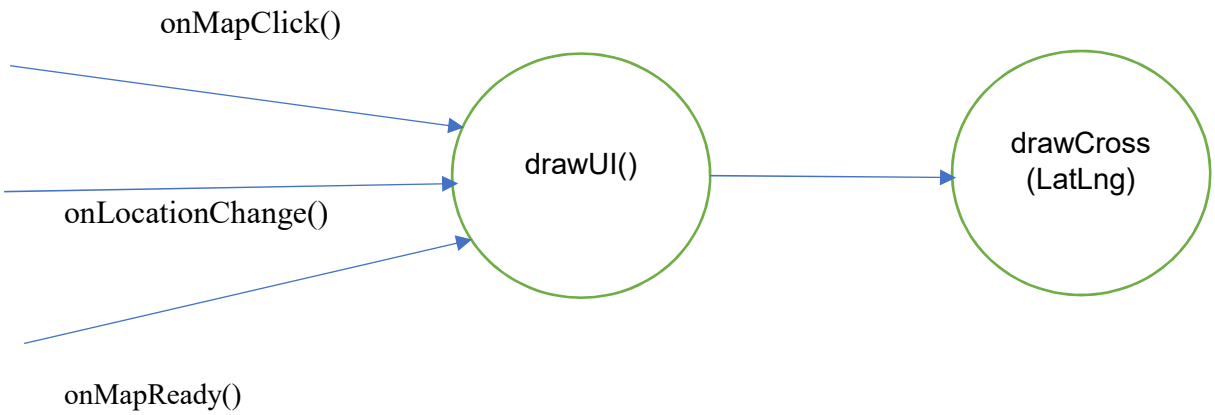

Figure 3.2.1. Figure showing the different calls to render locations.

The drawUI function first checks whether the location array is empty, and if it has an even number of entries. This check verifies that there are two values for each coordinate given in the array. Otherwise, we may have a coordinate with only latitude or only longitude.

```
private void drawUI() {

    mMap.clear();

    if (_locArray != null) {
        final int locSize = _locArray.size();
        //check if location array has at least two values for each coordinate
        if (locSize > 0 && locSize % 2 == 0) {
            for (int i = 0; i < locSize; i = i + 2) {
                drawCross(new LatLng(_locArray.get(_locArray.keyAt(i)),
                                     _locArray.get(_locArray.keyAt(i+1)))
                );
            }
        …
```

Figure 3.2.2. A java method for checking array parity mathematically.

```
private void startLocUpdates() {

    final int UPDATE_INTERVAL = 10000;
    final int FASTEST_INTERVAL = 5000;
    final float SMALLEST_DISPLACEMENT = 10f;
    final LocationRequest locRequest = LocationRequest.create()
            .setPriority(LocationRequest.PRIORITY_LOW_POWER)
            .setInterval(UPDATE_INTERVAL)
            .setSmallestDisplacement(SMALLEST_DISPLACEMENT)
            .setFastestInterval(FASTEST_INTERVAL);
    if(ContextCompat.checkSelfPermission(this,
            Manifest.permission.ACCESS_FINE_LOCATION) ==
PackageManager.PERMISSION_GRANTED) {
        LocationServices.FusedLocationApi.requestLocationUpdates(mGoogleApiClient,
locRequest, (com.google.android.gms.location.LocationListener) this);
    }
}
```

Figure 3.2.3. A java method for requesting location updates via GoogleClientApi.

3.3: Results

Replacing FusedLocationProviderApi with the defined AccurateLocationListener from Section 1.2 might give similar results and use less Google dependencies. Future developments may scrap Google dependencies and use a custom overlay for map viewings. Figure 3.2.3 shows how the GoogleClientApi is used to retrieve location updates. This shows that the location provider can be easily parameterized by minimum distance traveled, and update intervals, but there is no parameter for accuracy. A more sophisticated approach for saving files is necessary for this project to continue. Future developments will allow the user to choose an output file or directory from internal or external storage.

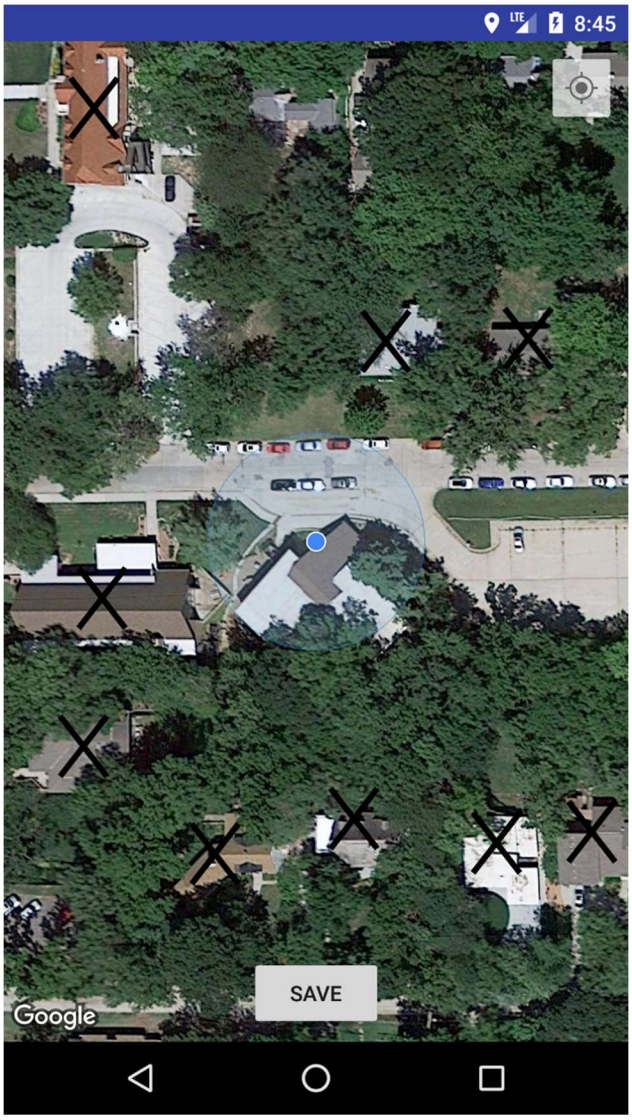

Figure 3.3.1. A screenshot of Survey running on a virtual Android device.

Conclusion

This report has explained the layout, implementation, various algorithms, and resource utilization used by three Android applications created to help plant scientists. As this research continues, features and changes will be made to this software and kept uploaded on respective GitHub repositories. All algorithms provided are linear or constant in time and have minimal effect on the Android UI thread. This project implements various asynchronous tasks to leverage the multi-processors present in most Android devices to offload work from the UI thread. Combining such asynchronous tasks into a library might be beneficial for developers to not reuse common code e.g. parsers. This paper also describes some functionality of the GoogleClientApi which provides most modern phones with geographical information, and how it lacks the parameter for accuracy. This project has also pointed out uses of Android objects and Views that optimize memory usage, to create minimal applications. The applications and services described in this paper can potentially be the start to a useful toolkit for plant scientists to acquire data and manage farmland plots.

References

1. Rife, T. W., and J. A. Poland. 2014. “Field Book: An Open-Source Application for Field Data Collection on Android.” Crop Sci. 54:1624-1627. doi:10.2135/cropsci2013.08.0579

2. M.L. Neilsen, S.D. Gangadhara, T. Rife, “Extending watershed segmentation algorithms for high throughput phenotyping”, in *Proceedings of the 29th International Conference on Computer Applications in Industry and Engineering*, Denver, CO, Sept. 26-28, 2016.

3. Marko Modsching, Ronny Kramer, and Klaus ten Hagen. “Field trial on GPS Accuracy in a medium size city: The influence of built-up” in *Proceedings of the 3rd workshop on positioning, navigation, and communication.* 2006.

4. Calculate distance, bearing and more between Latitude/Longitude points, http://www.movable-type.co.uk/scripts/latlong.html 2017.

5. Android Development Patterns:Best Practices for Professional Developers, Phil Dutson, 2016.

6. Mobile Vision, https://developers.google.com/vision 2017.

7. Android API Reference, https://developer.android.com/reference/android/ 2017.