# Test Plan Document

**Team Android Optimizers**

Jose Franco Baquera

Atiya Kailany

Jared Peterson

CS 448

December 7, 2019

# Table of Contents

# 1. Introduction

This test plan document will describe in detail any strategies and processes that will be used by team Android Optimizers to verify and validate the *Autonomous Robotic Vehicle (ARV)* application. Furthermore, the document will inform our clients on which processes will be used to gather qualitative and quantitative data about the implemented system. Relevant sections discussed include the following: major testing issues, major testing areas, testing strategies, testing environments, performance and stress testing, and project results.

## 1.1   Major Testing Issues

In this subsection, we discuss the most important and problematic issues that team Android Optimizers needs to deal with when evaluating the implemented system. More specifically, we will describe what is needed to test and evaluate our system. Since the entire application is highly interactive and is heavily intertwined with components outside our scope of work, it will be hard to test it fully as endless possibilities exist. The following subsections describe in more detail any issues that we will have to deal with while testing our system.

### 1.1.1   *Spitfire* Server Correctness

The application is heavily dependent and intertwined with NMSU's *Spitfire* server since users can manipulate any chosen virtual drone. However, reprogramming, modifying, testing, or reconfiguring this server is outside the scope of work for this project. This can lead to some problems while testing the application. For example, when sending a command to *Spitfire* to send a drone to a particular location, the server can return erroneous coordinates. This leads to the following dilemma: Is the implemented system sending the correct arguments or is *Spitfire* doing erroneous calculations? For testing purposes, we will assume that *Spitfire* will always do accurate calculations and will only publish erroneous coordinates when erroneous arguments are passed to it.

### 1.1.2   Google Maps Correctness

The application is also heavily dependent and intertwined with Google Maps' servers since a user can choose coordinates on the map to send a particular drone to. Reprogramming, modifying, testing, or reconfiguring these servers and the API responsible for establishing communication between them and the Android application is, likewise, outside the scope of work for this project. For testing purposes, we will assume that Google Maps' API will always return accurate/precise latitude and longitude coordinates.

### 1.1.3   Variation in User Opinions

The user interface was designed with well-established user interface design principles and practices, such as Fit's Law (see Design Specification Document for more information). However, when users are testing our system, many of them

might have different opinions or suggestions that might not necessarily reflect good design principles. Therefore, satisfactory thresholds will be established in order to account for this variability while doing user testing. A perfect example of one satisfactory threshold will be that 85% or more of users must be satisfied on how the interface allows them to send a drone to a location. These thresholds are needed while testing our system since it will prevent us from forming erroneous conclusions.

### 1.1.4   Floating-Point Number Accuracy

The coordinates stored in *Spitfire* (x and y) are different than the typical coordinates used by Google Maps (latitude and longitude). Because of this, the Android application must use mathematical formulas to convert *Spitfire* coordinates to latitude and longitude coordinates, and vice versa. *Spitfire* will also do mathematical manipulations on any x and y arguments passed to it before it can return any relevant game information. With all this data manipulation on floating point numbers, we expect to lose some precision and accuracy. For example, if a user chose the *actual* coordinates 32.2777 and -106.747, *Spitfire* might return the *calculated* coordinates 32.183 and -105.991. Therefore, calculating percent errors between expected and actual coordinates will be needed in order to test the accuracy and precision of our system.

### 1.1.5   Game Engine Correctness

Lastly, the application is heavily dependent and intertwined with the game engine that is responsible for communicating with *Spitfire* about simulated game scores, drone battery life, and any potential hidden objects and danger zones. Reprogramming, modifying, testing, or reconfiguring this game engine is also outside the scope of work for this project. This can lead to some problems while testing the application, especially since hidden objects and danger zones may vary from gameplay to gameplay. These variations may occur because the locations of danger zones and hidden objects can be modified at any time by Ahmed and Dr. Toups. In addition, we have no way of knowing if the communication between the game engine and *Spitfire* is corrupted or erroneous. For example, if a drone does not find a hidden object that we know exists, does that mean that our system is not displaying it or that the game engine is not simulating it correctly? For testing purposes, we will assume that the game engine will always simulate and publish to *Spitfire* the following Topics correctly: game scores, drone battery life, hidden objects, and danger zones.

## 1.2   Major Testing Areas

In this subsection, we discuss some of the major testing areas of our system. Because of the issues and restrictions mentioned in section 1.1, we will focus on testing only certain

parts of our system that are feasibly possible to test. The most important testable aspects of our system include the following:

- Accuracy between *actual* coordinates and *calculated* coordinates
- System performance such as startup time and "janky" frames proportion
- User interface design
- Mobility of Android application between tablet to tablet
- Communication with *Spitfire* (i.e. argument passing)

These are, in essence, the areas that must be fully evaluated and tested for the application to be fully functioning given our scope of work. Each of these testing areas will be discussed in more detail in sections 2, 3, and 4.

## 1.3   Links to Relevant Documents

- *Design Specification* – The following link can be used to access the project's *Design Specification* document*:*
  https://eltnmsu-my.sharepoint.com/:b:/g/personal/jose5913_nmsu_edu/EQbkIc6P1K9CsQx2hrEtrv8BMS_H2WcuIi5bZ-Tplt_r4w?e=hXVUUr

- *Requirements Specification* – The following link can be used to access the project's *Requirements Specification* document*:*
  https://eltnmsu-my.sharepoint.com/:b:/g/personal/jose5913_nmsu_edu/EQbkIc6P1K9CsQx2hrEtrv8BMS_H2WcuIi5bZ-Tplt_r4w?e=Zq1vTi

- *Statement of Work* – The following link can be used to access the project's *Statement of Work* document:
  https://eltnmsu-my.sharepoint.com/:b:/g/personal/jose5913_nmsu_edu/EZ-cpMLMQypNppmWFABqVagBhKiNphEzzEbQVM9Ruo0Vcg?e=W4KQwa

## 1.4   Acronyms, Abbreviations, and Definitions Used Throughout the Document

| Term | Definition |
|------|------------|
| **Android Device** | Physical Android device |
| **Android Studio** | Official integrated development environment for Google's Android operating system |
| **ARV** | Acronym for *Autonomous Robotic Vehicle* |
| **Danger Zone** | Drone and human danger zones that can be encountered by a drone while in-flight |
| **F.L.O.P.S** | Acronym for *Floating Point Operations Per Second* |
| **Hidden Object** | Drone and human hidden objects that can be found by a drone while in-flight |

| | |
|---|---|
| **NMSU** | Acronym for *New Mexico State University* |
| **Publish** | Occurs when *Spitfire* returns messages to ROSBridge |
| **ROS** | Acronym for *Robot Operating System*; An open-source operating system for robots that provides services such as hardware abstraction and message-passing between processes |
| **ROSBridge** | Protocol used to communicate to a ROS-implemented machine or server |
| ***Spitfire*** | The NMSU server that will be responsible for providing relevant game information |
| **Virtual Drone / Drone** | A drone object on the Android application user interface; Does not imply an actual physical drone |

**Table 1**. A list of the most widely used acronyms, abbreviations, and definitions used in this document.

# 2. Testing Strategy

In this section of the document, we will discuss the overall approach of evaluating the implemented system. This section also describes how we will integrate individual components and test them together in order to perform a fully integrated system test. More specifically, we will use a combination of unit, usability, mobility, and performance testing.

## 2.1 Unit Testing

For this particular project, we will focus specifically on testing the following three feasibly testable units using a bottom-up approach:

- *Coordinate Conversion and Argument Dispatcher Unit* – This unit is responsible for converting *Spitfire's* published x and y drone coordinates into real-life latitude and longitude coordinates, and vice versa. That is, this unit also converts real-life latitude and longitude coordinates into drone x and y coordinates that are recognizable by *Spitfire*. Furthermore, the unit is responsible for sending user-chosen coordinates to *Spitfire* whenever a user wishes to send a drone to a particular location (one coordinate needed) or to make a drone search an area (two coordinates needed). Both scenarios use mathematical formulas that convert one type of coordinate to another. These mathematical formulas were provided to us by our client and are assumed to be correct. Because of this, we must only test that we are sending the correct arguments to *Spitfire* and that our code does not do so many unnecessary F.L.O.P calculations that it degrades accuracy during coordinate conversion.

  To accomplish this test, we will send a drone to multiple random NMSU coordinates as arguments to *Spitfire*. ROSBridge's debug flag will then be turned on and will be used to display the final coordinate values returned from *Spitfire*. Lastly, we will calculate the percent error between the *actual* coordinates and *calculated* coordinates using the following formulas:

$$Percentage\ Error\ (Latitude) = \left|\frac{Calculated\ Latitude - Actual\ Latitude}{Actual\ Latitude}\right| \times 100\%.$$

$$Percentage\ Error\ (Longitude) = \left|\frac{Calculated\ Longitude - Actual\ Longitude}{Actual\ Longitude}\right| \times 100\%$$

It is crucial to note that we will use both formulas to test the send and search commands. Nevertheless, regardless of which coordinate component or command is tested, no percentage error shall be higher than 5%. Percentage errors that are higher than 5% provide evidence that we are either sending wrong arguments to *Spitfire,* or that our code has too many unnecessary F.L.O.P.S while converting coordinates.

One way we can improve the accuracy of our coordinate conversion and argument dispatcher unit is to re-write the provided mathematical formulas. Generally speaking, subtracting two nearly equal numbers should be avoided. Furthermore, nested multiplication should be performed whenever a polynomial is evaluated. These two guidelines minimize the number of F.L.O.P.S needed to evaluate a mathematical equation, hence minimizing the error. More information pertaining to this subject can be found online.

- *Landing and Altitude Argument Dispatcher Unit* – This unit is responsible for sending any arguments that are related to landing a drone or modifying a drone's altitude to *Spitfire*. To test this component, we will manually land and modify the altitude of all four drones. Furthermore, ROSBridge's debug flag will be turned on and will be used to see if a chosen drone actually lands and if its altitude changes accordingly. We assume that drones will not land or change altitude correctly only when this unit does not dispatch the correct arguments to *Spitfire.* The following table illustrates what is expected when a chosen drone is landed or modified (in terms of altitude):

| Command | Altitude Expectation (meters) |
|:---:|:---:|
| *High Altitude* | 25 |
| *Low Altitude* | 15 |
| *Land* | 0 |

**Table 2**. A chosen drone's altitude expectation after a user chooses the high altitude, low altitude, or land command.

- *User Interface Unit* – This unit is responsible for displaying drones, danger zones, hidden objects, altitude, battery life, game score, NMSU's map, and other relevant game information. It is important to mention that the unit fails only when it is not displaying the elements correctly and may occur even if other components are working properly. To test this component we will, in essence, check if there is a correct mapping between what the user interface is displaying and *Spitfire's* responses. For example, if a user chooses to send a drone to a specific coordinate, then the drone must move to that coordinate. Another example is that if a user chooses to land a drone, then the drone's altitude displayer should drop to 0. The user interface unit will be checked during regression and integrated testing. Please refer to sections 2.2, 2.3, and 3.1 (i.e. Record Log of Visual User Interface Tests) for more information.

## 2.2 Regression Testing

Because of the complexity and nature of this project, regression testing cannot be automated and must be done manually. Team members will test the functionality of each component individually and integrated whenever a modification is made and/or a new feature is added. In other words, our team will do regression testing by doing "mini" component integration and integrated testing before each demo. Another way our team will do regression testing will be by asking for acceptance from our clients when we demo the system to them every week. Please refer to section 2.3 for more information about component integration and integrated testing.

## 2.3 Component Integration and Integrated Testing

We will integrate user interface and server communication components only when they are fully tested. That is, each individual and testable component mentioned in this document will be tested accordingly. It is worth mentioning that a user interface component will always map to a specific server communication component. For example, the interface allows users to choose a drone and send it to a specific location on the map. This afforded user task maps to the application requesting services from *Spitfire.* To implement the connection between these components, we will get user input, manipulate it as needed, and communicate with *Spitfire* using ROSBridge. The application will wait until *Spitfire* publishes relevant events and will handle them appropriately.

Because of the previously mentioned information, our team will do integrated testing by checking if there are any discrepancies between what is being displayed on the interface, user input, and what the server is actually publishing. More specifically, we will check if there is a correct mapping between what the user interface is displaying and *Spitfire's* responses. An example of such a discrepancy would be a drone searching a different area on the map that was not chosen originally by the user. We will conclude that the application is handling *Spitfire's* published events incorrectly if such discrepancies occur.

We note that the application's individual components are heavily intertwined with one another, which implies that we cannot test a single component alone without other dependencies. Therefore, when we test a single component, we will also be, in essence, doing integrating testing. Lastly, integrating testing will also be done along the way when our team conducts regression testing before each demo. It is important to note that integrated testing will be accomplished visually and manually only since it would be too complex to write a program that tests the user interface's display as commands are entered.

## 2.4 Usability Testing

Usability testing is extremely important since it will allow us to determine whether or not our interface is usable by the intended user population. Furthermore, such testing will also enable us to see if the intended user population can carry out the tasks that the interface was designed to afford. Testing of our user interface will be done through a controlled setting involving users. Potential users of interest that will test our interface include classmates, professors, and perhaps our clients. Direct observations will be made while we instruct a user to accomplish a specific task, such as sending a drone to a specific location, sending a drone to search, landing a drone, or modifying a drone's altitude. After each user finishes testing our application, a mix exit-interview (i.e. an interview with both open and close questions) will take place in order to gather feedback on the interface's ease of use, content organization, simplicity, and overall design. A reasonable satisfactory threshold is that 85% or more of all participants must be satisfied with the overall design and implementation of our user interface. The following list of interview questions will be asked to participants after they finish using our application:

1. Is sending a drone to a location easy or difficult? Why?
2. Is sending a drone to search easy or difficult? Why?
3. Was it easy or difficult to land a drone? Why?
4. Was it easy or difficult to modify a drone's altitude? Why?
5. Does the user interface look cluttered or uncluttered? Why?
6. Was the application easy or hard to use? Why?
7. Is there anything in the application that confused you?
8. With 5 being extremely hard and 1 being extremely easy, how would you rate the application's ease of use? Explain.

## 2.5 Mobility and Portability Testing

The application's portability from hardware to hardware is another important aspect that needs to be tested. To test this aspect, we plan to download and run the application on several different emulated Android tablets using Android Studio. By doing this, we will, in essence, test the application's portability and its ability to be run by multiple Android tablets that differ in operating system version, hardware, and screen size. In order to pass this test, the application should not have performance or user interface issues while being run by different Android tablets.

## 2.6 Performance Testing

We will test several performance aspects of the application using a variety of tools. Each of these aspects are discussed in more detail in the following subsections. More information about performance and stress testing can be found in sections 3 and 4 of the document.

- *Janky Frames and Frames Per Second* – Most Android devices refresh their screen 60 times a second, meaning that it is crucial for the rendering of each frame to be done in less than 16 ms (i.e. 1s/60fps = 1 ms per frame). If a frame is not rendered in this allotted time, it is skipped. These "janky" frames negatively affect user experience since they cause a jump or skip in animation. To test this performance aspect, we will measure the number of "janky" frames that are skipped while using the application. It is crucial to note that a high percentage of "janky" frames is not desirable.

- *Startup Time* – The time it takes for the application to startup in any Android tablet should be reasonable. More specifically, our Requirements Specification document states that the Android application should display the main user interface 5 to 8 seconds after it is opened.

- *Tablet's CPU, Energy, and Memory Usage* – The Android application should not overuse the tablet's resources, such as its CPU, battery, and main memory. We note that network speeds will not be tested since this aspect is more reliant on internet speeds rather than the actual Android application.

## 2.7 Final Release Requirements

We will know that our system is ready for release once it passes all the testable components mentioned in this document and is able to accomplish successfully the following five requirements:

1. Send a drone to a location (will be tested)
2. Make a drone search an area (will be tested)
3. Land a drone (will be tested)
4. Modify a drone's altitude (will be tested)
5. Display hidden objects and danger zones (cannot be tested since simulations may vary from gameplay to gameplay)

## 2.8 Logging Failures and Resolutions

Every group member will keep track of any bugs or failures they encounter while implementing the application in individual project journals. These journals will also specify any attempts made to solve these bugs, as well as any successful resolutions.

These entries will be submitted through Canvas and will be available to any team member that requests them.

## 2.9 GitHub Prototypes and Group Meetings

Each working prototype will be pushed to a private GitHub repository, which our clients have access to. Furthermore, each prototype will be tested during group meetings with our clients in order to get feedback on both the correctness of the system and the design of the user interface.

## 2.10   Final Testing Case

As a final testing case, the completed system will be presented to Ahmed Khalaf and Dr. Toups. Both clients will use and test the system's entire functionality. Feedback will be gathered and acted upon only if time permits.

# 3. Test Environment

In this section of the document, we will describe any environments, methodologies, and testing tools that will be used by our group to test our system. It is worth mentioning that because of the highly interactive nature of our system, our test environments and chosen tools will differ from the ones used on more traditional systems.

## 3.1 Environments and Methodologies

The following list describes any environments and methodologies that will be used to test our system:

- *ROSBridge's Debug Flag* –  To do unit testing, we will turn ROSBridge's debug flag on, which will print information relating to all events published by *Spitfire.* We will then use the formulas mentioned in section 2.1 to manually calculate the percent errors that occur during coordinate conversion. It should be stressed that we will not use unit test tools (e.g. Junit, Expresso, etc.) to accomplish these unit tests since setting up such an environment would be difficult, time consuming, and unnecessary. Furthermore, we are more interested in testing the correctness of the arguments being passed to *Spitfire* than the application's logic.

- *Controlled Setting Involving Users* – To do usability testing, we will set up a controlled environment in which we will ask participants to use our system. Feedback will be gathered by using interviews, questionnaires, and/or direct observations. We note that a controlled setting is needed in order to avoid any potential bias from creeping in into our results.

- *Record Log of Visual User Interface Tests* – When doing integrating and regression testing, each group member will keep a record of any discrepancies found between what is being displayed on the interface, user input, and what the

server is actually publishing. It is important to highlight that setting up an automated test environment that simulates user input would be too tedious, difficult, and impractical since there is an infinite amount of possible user inputs.

## 3.2 Primary Tools

The following list describes the primary tools that will be used to test our system:

- *Android Studio Profiler Tool* – The Android Profiler is a tool provided by Android Studio that can be used to gather real-time data on how the application uses CPU, memory, and battery resources. This will be the primary tool used to conduct performance and stress testing.

- *dumpsys* – Dumpsys is a tool that runs on Android devices and is used to gather information about system services. More specifically, we will use the following command to test the number of "janky" frames that were skipped by the Android device: *adb shell dumpsys gfxinfo.* Before we run this command, however, we must run the *adb shell dumpsys gfxinfo reset* command first in order to get more accurate measurements. As previously mentioned in section 2.4, a high percentage of "janky" frames is not desirable.

- *Android Emulator* – The Android Emulator is a tool provided by Android Studio that simulates Android devices. This allows developers to test their applications on a variety of different devices and operating system versions. It is critical to note that this tool only provides a limited number of available Android tablets that can be simulated (e.g. we cannot simulate a Samsung tablet). Nevertheless, the Android Emulator will be the primary tool we use to conduct mobility and portability testing since it would be too expensive and impractical to test our application on multiple physical tablets.

# 4. Performance and Stress Testing

In this section, we describe how we will test our system to ensure that the performance of it is satisfactory. Ideally, we would also describe how our system would operate under high usage load. However, we do not anticipate a need to stress test our system since only one user at a time can use the Android application and the user can only modify one drone at a time. This implies that the load on the system will stay somewhat consistent whenever it is running on an Android tablet. Additionally, testing *Spitfire's* efficiency on how it would operate under high usage load is outside our scope of work. We will, however, test if the system does have performance degradation when a user manipulates multiple drones in a short period of time. The following list of performance standards will be used to ensure that the performance of our system is satisfactory:

- *Janky Frames Standard* – 75% or more of the total frames rendered by the application should not be classified as "janky."

- *Startup Time Standard* – The application should display the main user interface at most 8 seconds after it is opened.
- *Tablet's CPU, Energy, and Memory Usage Standards* – The application must adhere to the following standards regarding resource consumption:

| Resource | Standard |
|---|---|
| *CPU* | The application can only use at most 70% of the tablet's CPU power while running. |
| *Battery* | The application can use the tablet's battery only in a moderate manner. That is, an application that drains a tablet's battery life quickly is undesirable. |
| *Memory* | The application can only use at most 65% of the tablet's RAM memory. |

**Table 3**. A list of resource consumption standards that our system should meet.

More information on performance testing and any relevant tools that will be used can be found in sections 2.4 and 3.2 of the document.

# 5. Project Results

In this section, we summarize the actual results of developing our project and utilizing the methods previously mentioned. We note that a number of single tests and their outcomes were purposely omitted from this document, since we focused more on providing a history of our team's experience in using our test plan ideas.

## 5.1 Unit Testing Results

In this subsection, we will summarize the results that were gathered while testing all three system units.

- *Coordinate Conversion and Argument Dispatcher Unit Test Results* – Table 4 summarizes the latitude and longitude percent errors that were manually computed through multiple trial runs using the following two formulas:

$$Percentage\ Error\ (Latitude) = \left| \frac{Calculated\ Latitude\ -\ Actual\ Latitude}{Actual\ Latitude} \right| \times 100\%.$$

$$Percentage\ Error\ (Longitude) = \left| \frac{Calculated\ Longitude\ -\ Actual\ Longitude}{Actual\ Longitude} \right| \times 100\%$$

It is worth noting that the purple portion of the table corresponds to testing the *send* command while the green portion corresponds to testing the *search* command. Furthermore, it is important to reiterate that the *send* command only requires one coordinate point, while the *search* command requires two.

| Trial Run | Latitude Percent Error | Longitude Percent Error |
|---|---|---|
| **1** (*Point 1*) | $\left\|\frac{32.34567 - 32.279595}{32.279595}\right\| = \left\|\frac{0.066075}{32.279595}\right\| \times 100\% = 0.2046\%$ | $\left\|\frac{-106.12034 - (-106.748067)}{-106.748067}\right\| = \left\|\frac{0.627727}{-106.748067}\right\| \times 100\% = 0.5880\%$ |
| **2** (*Point 1*) | $\left\|\frac{31.92392 - 32.279890}{32.279890}\right\| = \left\|\frac{-0.35597}{32.279890}\right\| \times 100\% = 1.10278\%$ | $\left\|\frac{-106.85234 - (-106.754018)}{-106.754018}\right\| = \left\|\frac{-0.098322}{-106.754018}\right\| \times 100\% = 0.09210\%$ |
| **3** (*Point 1*) | $\left\|\frac{32.292302 - 32.282238}{32.282238}\right\| = \left\|\frac{0.010064}{32.282238}\right\| \times 100\% = 0.0312\%$ | $\left\|\frac{-106.75672 - (-106.754312)}{-106.754312}\right\| = \left\|\frac{-0.002408}{-106.754312}\right\| \times 100\% = 0.00226\%$ |
| **1** (*Point 1*) | $\left\|\frac{32.29239 - 32.280679}{32.280679}\right\| = \left\|\frac{0.011711}{32.280679}\right\| \times 100\% = 0.03628\%$ | $\left\|\frac{-106.74791 - (-106.747811)}{-106.747811}\right\| = \left\|\frac{-0.000099}{-106.747811}\right\| \times 100\% = 0.000092\%$ |
| **1** (*Point 2*) | $\left\|\frac{32.29234 - 32.282216}{32.282216}\right\| = \left\|\frac{0.010124}{32.282216}\right\| \times 100\% = 0.03136\%$ | $\left\|\frac{-106.76001 - (-106.750800)}{-106.750800}\right\| = \left\|\frac{-0.00921}{-106.750800}\right\| \times 100\% = 0.008628\%$ |
| **2** (*Point 1*) | $\left\|\frac{32.269394 - 32.279863}{32.279863}\right\| = \left\|\frac{-0.010469}{32.279863}\right\| \times 100\% = 0.0324\%$ | $\left\|\frac{-106.76234 - (-106.754214)}{-106.754214}\right\| = \left\|\frac{-0.008126}{-106.754214}\right\| \times 100\% = 0.007612\%$ |
| **2** (*Point 2*) | $\left\|\frac{32.29234 - 32.282338}{32.282338}\right\| = \left\|\frac{0.010002}{32.282338}\right\| \times 100\% = 0.03098\%$ | $\left\|\frac{-106.74394 - (-106.752927)}{-106.752927}\right\| = \left\|\frac{0.008987}{-106.752927}\right\| \times 100\% = 0.008418\%$ |

**Table 4**. Summary of latitude and longitude percent errors that were manually computed through multiple trial runs. We note that the purple portion of the table corresponds to testing the *send* command, while the green portion corresponds to testing the *search* command.

As the previous table demonstrates, no percent error that was calculated exceeded the 5% threshold that was originally established in section 2.1. In fact, all errors are relatively low, with all but one being less than 1%. We can, therefore, safely conclude that both the *send* and *search* commands do send correct arguments to *Spitfire,* and that the system does not carry out too many unnecessary F.L.O.P calculations.

- *Landing and Altitude Argument Dispatcher Unit Test Results* – The following table summarizes the results that were gathered while manually landing and modifying the altitude of all four drones:

| Drone ID | Altitude Before Command Is Pressed (meters) | Command | Altitude After Command Is Pressed (meters) | Test Passed? |
|---|---|---|---|---|
| 0 | 16.23 | *land* | 0.01 | ✔ |
| 0 | 0.01 | *high altitude* | 25.01 | ✔ |
| 0 | 25.01 | *low altitude* | 15.00 | ✔ |
| 1 | 18.02 | *land* | 0.02 | ✔ |
| 1 | 0.02 | *high altitude* | 25.00 | ✔ |

| | | | | |
|---|---|---|---|---|
| 1 | 25.00 | *low altitude* | 15.01 | ✔ |
| 2 | 25.01 | *land* | 0.00 | ✔ |
| 2 | 0.00 | *high altitude* | 25.02 | ✔ |
| 2 | 25.02 | *low altitude* | 15.01 | ✔ |
| 3 | 0.01 | *land* | 0.00 | ✔ |
| 3 | 0.00 | *high altitude* | 25.03 | ✔ |
| 3 | 25.03 | *low altitude* | 15.00 | ✔ |

**Table 6**. Summary of results that were gathered while manually landing and modifying the altitude of all four drones.

The previous table illustrates that there is some variability between the *land, high altitude,* and *low altitude* commands among all four drones. However, this variability is small enough to be statistically insignificant, and thus implies that our system did pass all the tests in this section. For instance, a drone is technically "landed" if its altitude is 0.01 meters (i.e. the altitude of the drone does not necessarily have to be exactly 0.00 meters). Variations in outputs that appear to be statistically insignificant allows us to safely conclude that the *land, high altitude,* and *low altitude* commands do send correct arguments to *Spitfire*.

▪ *User Interface Unit Test Results* – As previously mentioned in section 2.1, this unit fails only when it is not displaying game elements correctly, and may occur even if the other two units are working properly. To test this unit, we simply checked if there was a correct mapping between what the user interface displayed and *Spitfire's* responses. It is important to mention that, in order to see *Spitfire's* responses, our team turned on ROSBridge's debug flag. Turning this debug flag on made it much easier for our team to find and fix any bugs in this unit. Most noticeably, while our team was checking for the mapping between the user interface and *Spitfire's* responses to the *send* command, we noticed that, visually, the drone was moving correctly on the user interface. However, according to *Spitfire's* responses*,* the drone was being sent to a location that was opposite of what was originally chosen. We soon found out that we had accidently switched the x and y coordinates that were being published by *Spitfire* (i.e. the x coordinate corresponds to longitude, not latitude). Without ROSBridge's debug flag, we would have never realized that such mistake was made, since the user interface would always display correct drone movements. Other similar bugs were found, thanks to the debug flag, and were quickly fixed. We note that this unit was

vigorously checked during regression and integrated testing, which worked well for us since we were able to find bugs early on before they became a problem.

## 5.2 Regression Testing Results

In this project, regression testing was conducted by doing "mini" component integration and integrated testing before each demo. For example, while implementing the *land* command for demo 4, we first checked that the *Landing and Altitude Argument Dispatcher Unit* was working correctly by manually landing a drone. After this unit passed this test, we then tested the *User Interface Unit* by displaying random "dummy" values onto the user interface. This was done in order to make sure that the *User Interface Unit* was able to display double values correctly. After both units passed their corresponding tests, we then slowly integrated them together by checking if there was a correct mapping between what the user interface displayed and *Spitfire's* responses (see sections 2.1 and 5.1).

It was extremely beneficial to the development of the project to do regression testing before each demo, since it allowed us to find bugs early on. Furthermore, regression testing by doing "mini" component integration and integrated testing made it much easier to test our final system, since the testing workload was spread throughout the semester instead of a couple of weeks. In this project, regression testing was also conducted each week by asking for acceptance from our clients. This, too, was a brilliant idea, since bugs were pointed out by our clients every time they evaluated our Android application. Regression testing was, however, a little tedious, since we did not write or implement any automated tests. Therefore, we wish we could have done more research on how to implement automated testing for a highly interactive Android application.

## 5.3 Component Integration and Integrated Testing Results

Component integration and integrated testing for this project was done by focusing on the user interface components and its connection to other system components. More specifically, our team did integrated testing by checking if there were any discrepancies between what was being displayed on the user interface, user input, and what the server was actually publishing. This methodology was chosen and implemented since a user interface component will always map to a specific server communication component. For example, the first integrated test that our group conducted was to make sure that the *send* command took user input correctly and that it sent the correct arguments to *Spitfire*. The drone user interface component was then monitored to check whether or not it correctly used the coordinates being published by *Spitfire* to display the virtual drone.

The previous methodology for doing component integration and integrated testing worked very well for us since it made it easy for each team member to test the five different important interaction sequences (i.e. *send, land, search, high altitude,* and

*low altitude*). In addition, the methodology aligned itself very nicely with the structures of these important interaction sequences. For more information on the system's important interaction sequences, please refer to Figure 5 in our Design Specification document (i.e. this diagram illustrates how vital information is passed between different classes and servers).

As previously mentioned in section 2.3, the application's individual components are heavily intertwined with one another, meaning that we could not test a single component alone without other dependencies. Nevertheless, this possible dilemma proved to not be a problem since our project does not have much individual components to begin with. In fact, a heavily dependent system was more beneficial for us to test since we were able to catch bugs and other errors more quickly. A perfect example of this benefit occurred when we were testing the *Coordinate Conversion and Argument Dispatcher Unit,* which is heavily dependent on *Spitfire.* In a more conventional and different implementation of this project, it would be difficult to track which component has a bug if wrong results were being calculated or published. However, because we assumed that *Spitfire* would only publish erroneous events if we sent it erroneous arguments, it made it much easier to conclude that problems between it and the *Coordinate Conversion and Argument Dispatcher Unit* would always be caused by the latter.

## 5.4 Usability Testing Results

To conduct usability testing on our Android application, our group recruited six participants and instructed them to accomplish a series of specific tasks in a controlled user setting. After each participant finished the study, a short, unstructured, and open exit interview was conducted. The following table summarizes the most relevant responses gathered from our participants and highlights potential future changes that can be made to our system:

| Interview Question | Summary of Data Collected | Lessons Learned From Participants |
|---|---|---|
| Is sending a drone to a location easy or difficult? Why? | All six participants agreed that sending a drone to a specific location was easy. Most participants cited that the user interface was straightforward and intuitive for this command. | The user interface implemented the *send* command well. |
| Is sending a drone to search easy or difficult? Why? | Four participants stated that it was easy to make a drone search while the other two expressed that it was difficult. The most common reason cited as to why it was difficult was a lack of feedback on how many markers are needed to make a drone search. | There are some modifications that can be made to make the *search* command have more feedback and thus be easier to use. |

| | | |
|---|---|---|
| Was it easy or difficult to land a drone? Why? | All six participants agreed that landing a drone was easy. Most participants cited that the user interface was straightforward for this command. | The user interface implemented the *land* command well. |
| Was it easy or difficult to modify a drone's altitude? Why? | All six participants agreed that modifying a drone's altitude was easy. Most participants cited that the interface was straightforward for this command. | The user interface implemented the *high altitude* and *low altitude* commands well. |
| Does the user interface look cluttered or uncluttered? Why? | All six participants agreed that the user interface looked uncluttered since only the essential game components are being displayed at all times. Two participants did recommend, however, that we should "erase" the imaginary search squares after a certain amount of time. | Implementing a trigger button that "erases" all imaginary search squares might help reduce some of the interface's clutter. |
| Was the application easy or hard to use? Why? | All six participants agreed that the application was easy to use. Most participants cited that the user interface was simple and straightforward, and that the buttons had a correct mapping between its visual representation and functionality. | Users who have never used the Android application can quickly learn how to use it. |
| Is there anything in the application that confused you? | Two participants stated that the icon used to represent the *land* command was a little confusing since it maps to "stopping" rather than "landing." | Changing the icon used to represent the *land* command to something that maps better to the *land* functionality would help reduce ambiguity. |
| With 5 being extremely hard and 1 being extremely easy, how would you rate the application's ease of use? Explain. | Four participants rated our Android application as 1 while the other two rated it as 2. | 100% of our participants were reasonably satisfied with the overall design and implementation of our user interface. |

**Table 7**. Summary of most relevant responses and suggestions gathered from our six participants.

Conducting usability testing was extremely beneficial for our project since it allowed us to determine whether or not our intended user population can carry out the tasks that the interface was designed to afford. According to our participants, the application's user interface was well designed and implemented. Nevertheless, if time permits, we plan to still make the modifications that were suggested to us since they are valid and would make our user interface less error-prone.

## 5.5 Mobility and Portability Testing Results

In this section, we will discuss the mobility and portability testing results of our Android application. To test these aspects, the application was downloaded and run on several different emulated Android tablets using Android Studio. By doing this, we, in essence, tested the application's portability and its ability to be run by multiple Android tablets that differ in operating system version, hardware, and screen size. The results are summarized in the following table:

| Android Tablet Brand | Android API Level | Screenshot of Application Running | User Interface Test Passed? | Performance Test Passed? | Functionality Test Passed? |
|---|---|---|---|---|---|
| Lenovo Tab 10 | 27 |  | ✔ | ✔ | ✔ |
| Google Nexus 9 | 28 |  | ✔ | ✔ | ✔ |

| Google Pixel C | 29 |  | ✓ | ✓ | ✓ |

**Table 8**. Summary of the mobility and portability testing results of our Android application.

It is important to note that the application was still tested thoroughly on other different Android tablets and API levels not mentioned here. Ultimately, we concluded that our Android application is portable since it can run on multiple different Android tablets that differ in operating system version, hardware, and screen size.

## 5.6 Performance Testing Results

In this subsection, we will summarize the data that was gathered while testing the performance aspect of our Android application.

- *Janky Frames Standard Results* – As Figure 1 illustrates, 62.02% of all frames rendered at the time of the screenshot were classified as "janky." Nevertheless, after running this command several times during different states of the application, a 59.45% average was computed. Because our original threshold stated that we must have at most 35% of the total frames rendered classified as "janky," our Android application failed the *Janky Frames Standard* performance aspect. We note, however, that the total number of "janky" frames was heavily dependent on which computer was running the emulator. In other words, a computer with more CPUs and computational power had less "janky" frames on average. Because of this, we wish we could have found a more robust way (that did not depend on computer hardware) of finding the total percent of dropped frames.

```
josefranco@jose59134de3 ARVWearable % adb shell dumpsys gfxinfo | grep –A 6 Window
Window: com.example.arvwearable/com.example.arvwearable.MapsActivity
Stats since: 44224459762903ns
Total frames rendered: 3391
Janky frames: 2103 (62.02%)
50th percentile: 20ms
90th percentile: 27ms
95th percentile: 30ms
```

Figure 1 – Screenshot of output after running the command once. We note that this is only one example of many trial runs.

- *Startup Time Standard Results* – On average, the application displayed the main user interface in 2.445 seconds after it was opened, which is well below the 8 second threshold that was formulated in section 4. It is worth mentioning that we measured the time it took to open the application only *after* the emulated Android tablet finished starting up.

- *Tablet's CPU, Energy, and Memory Usage Standards Results* – The Android profiler tool was used to test the application's resource consumption over time. Figure 2 is a screenshot of the profiler tool after several commands were run (e.g. sending a drone to search, landing a drone, etc.). These commands are represented as pink events on top of the profiler. From Figure 2, we note that using the implemented functionalities did not impact the application's resource consumption.
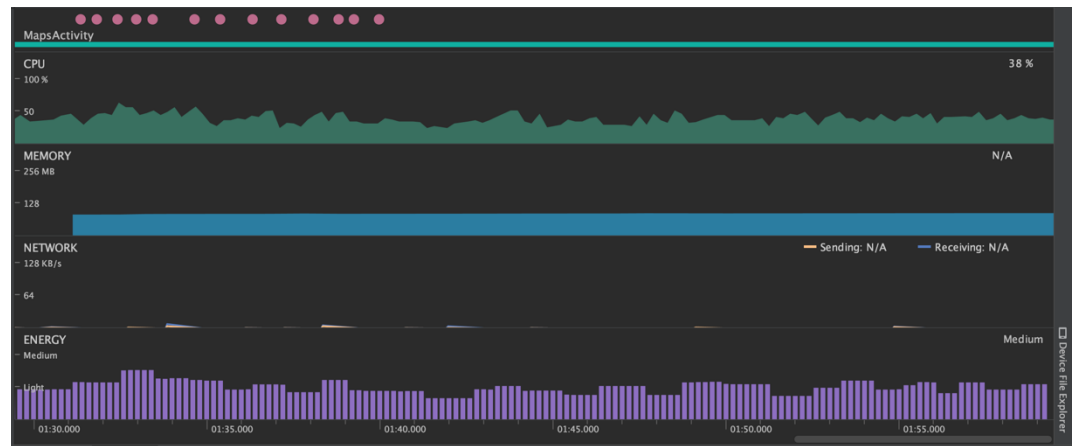


Figure 1 – Screenshot of Android Studio's profiler tool that was used to measure the application's resource consumption over time.

The following table summarizes more specific data that was gathered and uses it as evidence to conclude whether specific performance tests were passed:

| Resource | Standard | Test Result | Test Passed? |
|---|---|---|---|
| *CPU* | The application can only use at most 70% of the tablet's CPU power while running. | The application used at most 45% of the tablet's CPU power while running. | ✔ |
| *Battery* | The application can use the tablet's battery only in a moderate manner. That is, an application that drains a tablet's battery life quickly is undesirable. | The application used the tablet's battery only in a "light" to "medium" manner. | ✔ |

| | The application can only use at most 65% of the tablet's RAM memory. | The application used at most 36% of the tablet's RAM memory while running. | ✔ |
|---|---|---|---|
| *Memory* | | | |

**Table 9**. Specific data gathered from Android Studio's profiling tool. A comparison is made between the gathered data and the thresholds established in section 4.

## 5.7 Final Release Requirements Results

We note that our Android application passed all testable components and is able to accomplish successfully the following five requirements:

1. Send a drone to a location (was tested and passed)
2. Make a drone search an area (was tested and passed)
3. Land a drone (was tested and passed)
4. Modify a drone's altitude (was tested and passed)
5. Display hidden objects and danger zones (was not tested but drones can find hidden objects and danger zones)

We conclude that our system is ready for deployment.

## 5.8 Additional Methods That Worked Well

In this section, we describe methods that worked well for the development of this project.

### 5.8.1 Logging Failures and Resolutions

Requiring team members to log any failures and resolutions they encountered while implementing the application was very beneficial. For example, logging attempted resolutions that did not fix a bug or problem prevented team members from retrying failed resolutions subconsciously since a record of their failure existed.

### 5.8.2 GitHub Prototypes and Group Meetings

Allowing our clients to have direct access to our source code was extremely beneficial since they were able to run and test our system constantly and thus provide feedback. Furthermore, group meetings were also beneficial for the development of this project since feedback and suggestions were frequently being provided by our clients.

### 5.8.3 Final Testing Case

Passing the final testing case (i.e. allowing Ahmed Khalaf and Dr. Toups to use and test the system's entire functionality) made our group more confident in the system's correctness and usability.

### 5.8.4 Controlled Setting Involving Users

The controlled setting that was setup for our usability testing allowed us to avoid any potential bias from creeping in into our results. That is, the data and suggestions gathered from our participants is valid, relevant, and accurately reflect real and unbiased opinions.

## 5.9 Methods That Did Not Work Well

In this section, we describe methods that did not work well during the development of this project.

### 5.9.1 Record Logs of Visual User Interface Tests

Writing record logs of visual user interface tests while doing integrating and regression testing soon became tedious and inefficient since too much unnecessary writing to describe the user interface was needed. Therefore, it would have made more sense to have taken screenshots of the user interface whenever a discrepancy would occur (i.e. this directly relates to the saying "one picture is worth a thousand words").

### 5.9.2 Coding Standards

Too many coding standards were formulated as a requirement. More specifically, the excess of comments required made the source code, at times, harder to read. Nevertheless, we wish we had commented our code along the way instead of waiting until the last minute.

## 5.10 Game Engine Knowledge

A concrete methodology for testing hidden objects and danger zones was not formulated because our group had little to no knowledge (at that time) on how the game engine worked. After learning about the internal workings of the engine, we soon found out that *thoroughly* testing the communication between it and the Android application is not needed. The reason why this is the case is because *Spitfire* acts as an intermediary between the game engine and Android application. Therefore, by testing the communication between *Spitfire* and the Android application, we are also, indirectly, testing the application's communication with the game engine.