Imperial College London

MSc Individual Project

Imperial College London

Department of Computing

xDrone Programming Language: Object Recognition and DSL Development

Author:
Gafar, Muhamad

Supervisor:
Dr Madasar Shah

Submitted in partial fulfillment of the requirements for the MSc degree in Computing Science of Imperial College London

January 6, 2019

Abstract

Programming empowers people in the modern world, by allowing them to use their problem-solving skills with a combination of the computational power available, to maximise efficiency. Programming can be made simpler through the use of high-level programming languages, integrated development environments (IDE), or domain-specific languages (DSL), among others. In an effort to attract new programmers a DSL was developed which targets the domain of drone programming to capitalise on its growing popularity. A prototype was created comprising of a Raspberry Pi server and a web application with an embedded custom IDE for the DSL; where the Raspberry Pi is used for interfacing with a *Parrot AR Drone 2.0* and a front-end user device. Furthermore, the DSL and web application are equipped with support for media content - this includes using the drone's camera feed for taking mid-flight snapshots or recording autonomous flights. Lastly, the DSL makes use of computer vision techniques for real-time face detection, employing cascade classifiers, and real-time object recognition through feature matching. The end result is a unique application that could be used by both hobbyists or novice programmers.

Acknowledgments

- **Parents:** First and foremost I would like to thank my parents for the endless support they gave throughout my academics.
- **Supervisor:** I am grateful to Dr Shah Madasar for his patience and guidance throughout the project. He was able to propose a project that was fun and challenging and this project wouldn't have been possible without him.
- **Brother:** I would like to thank my brother for this continuous support. He was also kind enough to generate the *xDrone* logo and banner (shown at the end of the page).
- **Friends:** The friends I've met at Imperial kept me going through the tough course. In no particular order, my gratitude extends to: Quang Nguyen, Ashman Misra, Talha Ahmed, Darrell Adjei, and Shuai Zeng.
- **Imperial:** I am thankful to the institution and all the staff members; without the teaching and encouragement at Imperial the project wouldn't have been the same.
- **User Testers:** I am grateful for the user testers that agreed to take part in the user study. They dictated some aspects of the project and provided useful insight.

Contents

1 Introduction **1**
1.1 Aims and Motivation 1
1.2 Contributions 2
1.3 Report Outline 2

2 Background **3**
2.1 Domain Specific Languages 3
2.1.1 Parsers 3
2.1.2 Abstract Syntax Tree 4
2.1.3 Integrated Development Environment 5
2.1.4 Xtext and Xtend 5
2.2 Build Automation Tools 5
2.2.1 Gradle vs Apache Maven 6
2.3 Web Application Deployment 7
2.4 Existing Drone Programming Software 7
2.5 Parrot AR Drone 2.0 9
2.6 OpenCV 10
2.6.1 Object Detection 11
2.6.2 Object Recognition 12
2.6.3 Image Resolution and Computer Vision 14
2.6.4 Drones With Face Detection 15

3 Specifications **16**
3.1 Core Specifications 16
3.2 Possible Extensions 16

4 Implementation **18**
4.1 Development Methodology 18
4.1.1 Continuous Integration 19
4.1.2 Work Flow 20
4.2 Walking Skeleton 21
4.2.1 Architecture 1 21
4.2.2 Architecture 2 21
4.2.3 Initial Web Application and DSL 22
4.2.4 Server Configuration 23

4.2.5 Continuous Integration 23
4.3 Face Detection 24
4.3.1 Dependencies and Implementation 24
4.3.2 Optimization 25
4.3.3 Integrating With DSL 25
4.3.4 Full Architecture Testing and Further Optimisation 26
4.3.5 Regression Testing 26
4.4 Feature Matching 27
4.4.1 Dependencies and Implementation 27
4.4.2 Changes to DSL and Regression Testing 30
4.5 Viewing Generated Code 31
4.6 Image Gallery 32
4.6.1 Changes to DSL 32
4.6.2 Changes to Server 32
4.6.3 Additions to Web Application 32
4.7 Video Recording 34
4.7.1 Dependencies 34
4.7.2 Implementation and Changes to DSL 34
4.7.3 Changes to Server 35
4.7.4 Changes to Web Application 35
4.8 Connection Feedback 35
4.8.1 Changes to Server 35
4.8.2 Changes to Web Application 36
4.9 Landing Site Recognition 36
4.9.1 Changes to DSL 36
4.9.2 Landing Site Experiment 37
4.10 User-Functions 38
4.10.1 Changes to DSL and Generator 38
4.10.2 IDE Validation 39
4.10.3 Main Program Syntax Graph 39
4.10.4 Parser Rules Breakdown 41
4.11 Request Blocking 42
4.12 Final Architecture Revision 44

5 Evaluation **45**
5.1 User Testing in Person 45
5.2 User Validation Using a Survey 47
5.3 Implementation of The Components 50
5.3.1 The DSL 50
5.3.2 Image Gallery and Video Player 51
5.3.3 Server Updates 52
5.3.4 Displaying Generated Code 52
5.3.5 Computer Vision 52
5.3.6 The Architecture 53
5.4 Existing Software Comparison 53

6 Conclusion and Future Work **55**
6.1 Contributions 55
6.2 Future Work 56
6.2.1 The DSL 56
6.2.2 Use of Generated Code 56
6.2.3 Image Gallery and Video Player 57
6.2.4 Security Implications 57
6.3 Conclusion 57

A Ethics Checklist **I**

B Ethical and Professional Considerations **IV**

C User Guide **V**
C.1 Commands V
C.2 Configurations VI
C.3 Structure of the Program VI
C.4 The Web Application X

Chapter 1

Introduction

1.1 Aims and Motivation

Consumer electronics have been highly influential in today's work environment due to their ability to outperform humans in certain tasks. One particular field which has allowed humans to utilise their problem-solving skills is computer programming; however, people without a background in computer science or engineering fields may find programming difficult. In recent years efforts have been made to attract more programmers by making programming easier. Such efforts include development of high-level programming languages, provision of user-friendly tools and resources such as Integrated Development Environments (IDEs) and libraries, and targeting a particular application domain with specialised languages through Domain-Specific Languages (DSLs).

In recent years, flying drones have had a significant rise in popularity from hobbyists programmers. However, a larger audience could be obtained if the programming is made simpler. This project focuses on designing and developing a DSL with the aim of facilitating drone programming for inexperienced coders. This can be achieved by targeting this specific domain and giving the user the ability to control the drone while abstracting from low-level programming (LLP). LLP refers to programming where functions and commands are closely mapped to the computer's instruction set architecture.

The project also aims to explore potential architectures to be used for integrating the DSL with a web application and the drone. The architecture and the user interface play a key role in the user experience and hence must be designed with the final user in mind.

Lastly, the project will to explore Computer Vision (CV) in drones through the use of DSLs. Computer Vision is a field in computing science that focusses on processing visual data for applications such as face detection, object tracking, or setting recognition. By combining CV algorithms with DSLs, the project could allow users with weak programming backgrounds to write autonomous flight programs which are de-

pendent on object recognition. In the context of drones, autonomous flights refer to pre-programmed flight sequences that can be executed without human intervention.

1.2 Contributions

An outline of the project's main contributions to the field of DSLs, for drone programming, can help put this report into context. The project's main contributions are:

- **Computer Vision:** The DSL is able to provide support for real time face detection and object recognition, using a drone's camera feed. The face detection and recognition are used for executing conditional commands. For example, a user can program the drone to capture a photo if the drone detects someone's face. No other DSL-based applications, aimed at facilitating drone programming, were found (at the time of writing) to have this functionality.
- **Web Application:** The DSL is embedded in a web application with a custom IDE and support for easy access of the media content captured by a drone. Existing applications that simplify drone programming, through graphical programming, were used for comparison with this project's web application; only support for taking snapshots was found. The web application developed for this project takes a step further by allowing the user to programatically record an autonomous flight and view it after the flight has completed.

1.3 Report Outline

The report is structured as follows:

Chapter 2 - Discusses the existing work in the field and the necessary background material.

Chapter 3 - Outlines the core requirements of the project and the possible extensions that will be considered.

Chapter 4 - Documents the chronological development of the project. This includes implementation details, problems encountered and solutions considered, testing, and optimisation.

Chapter 5 - Discusses the tests conducted of the developed application. Users were given an opportunity to have first-hand experience and reflect on the application; their thoughts were outlined.

Chapter 6 - Summarises the project's contributions and the potential extensions.

Chapter 2

Background

This section outlines the relevant background required for this project. A discussion on the existing work in this field will help clarify where this project is situated in the field of DSLs. Furthermore the project required multiple dependencies, they will also be discussed in this section along with other dependencies that were considered but not used.

2.1 Domain Specific Languages

DSLs are typically implemented by targeting a specific framework and mapping tasks to general purpose languages (GPLs) such as *C*, *Java* and *Python* [1]. DSLs makes code easier to understand as the grammar and syntax can be defined in a way that is easy for the target users to understand.

2.1.1 Parsers

A fundamental aspect of programming languages is their need to follow a specified syntax. This is generally done using lexical and syntactic analysis. Lexical analysis refers to the process in which a piece of code is broken down into tokens; such as literals, operators, and identifiers. These tokens are stored in sequential format and the structure is analyzed to verify that it satisfies the language's grammar and syntax, through a program called a parser, sometimes referred to as expression evaluators [2]. The role parsers play in DSLs is whether the parser from the host GPL is reused or if a new parser is written specifically for the DSL.

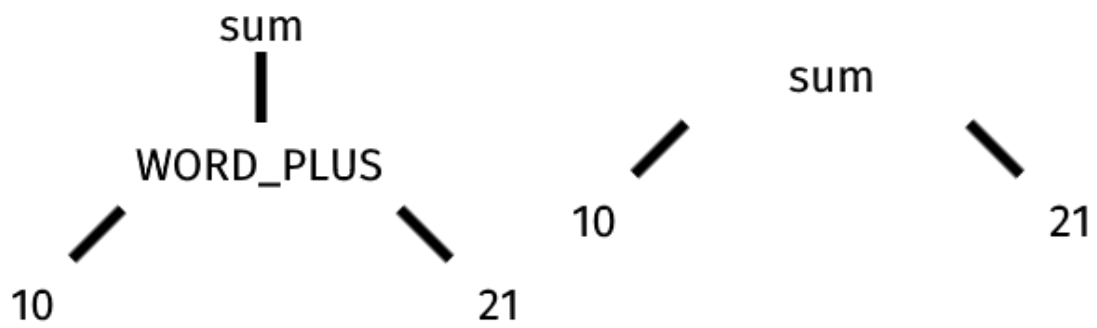
In embedded DSLs the infrastructure of the host-language can be inherited by reusing the parsers. In this case the DSL would share aesthetic similarities and syntax with the host-language since the syntactic analysis doesn't change. Martin Fowler refers to these as internal DSLs and describes them as "a stylized use of that (host) language for a domain specific purpose".

Standalone DSLs, also referred to as external DSLs, do not inherit the infrastructure from the host GPL. It's syntax can be defined by the creator so it requires a custom parser for syntactic analysis. The added flexibility for non-embedded DSLs can make them harder to implement and users that already understand the host-language will have to learn the new syntax.

2.1.2 Abstract Syntax Tree

For a DSL to be usable it needs to be able to generate code. This is usually achieved by having data structures which "represents all of the syntactical elements of a programming language" and then using these elements for compiling and code generation [3]. These data structures are called, Abstract Syntax Trees (AST), and they resemble a hierarchal tree like structure when representing a program. Put simply, the AST differs itself from the data structure used by the parser by ignoring tokens which are required in the original program but not required for code generation, hence why it is called "abstract". These ignored tokens include brackets, whitespace, and expected keywords.

Moreover, the AST is responsible for semantic analysis of the program. Part of semantic analysis is type checking; for example, the parser may not detect any syntactic error when parsing but the AST can detect that an assigned variable is a different type to what the program expects [2].



(a) Parser Tree **(b)** Abstract Syntax Tree

Figure 2.1: Example Parser Tree and AST for a DSL used for adding two integers. Example taken from [4]

.

Figure 2.1 shows the difference between a Parser Tree and AST used in a program with keyword "WORD_PLUS" and two integers to be added. This example shows the artifacts required for constructing a parser tree and how they are reduced to only the required artifacts needed for compilation and code generation.

2.1.3 Integrated Development Environment

IDEs have been used extensively for programming as they make coding easier with visual feedback; such feedback include syntax highlighting, showing errors and warnings, and providing assistance such as automatic code completion or suggestions. However, well-known IDEs are tailored to one, or multiple, GPLs and hence can't be used with DSLs created for a specific purpose. Hence custom IDEs are usually required to be shipped with the DSL, as they will "increase the likelihood of your DSL being adopted and successful" [2].

Custom IDEs should be configured to provide a good user experience. Like normal IDEs they must also provide support for syntax highlighting, highlighting errors appropriately and displaying the error message, and providing assistance for suggested content additions.

2.1.4 Xtext and Xtend

Xtext is the language engineering workbench which was used for development of the prototype (DSL) throughout this project. *Xtext* is an extension of *Eclipse* and it provides the necessary framework for creating the DSL's infrastructure (linker, compiler, parser, semantics definition, etc.). When using *Xtext* the framework is responsible for automatically generating the parser, the AST, and an eclipse editor with some IDE features. This makes the project simpler as several components can be automatically generated.

Xtext itself is shipped with a programming language called *Xtend*. *Xtend* is a GPL for the *Java Virtual Machine (JVM)* and as a result is commonly referred to as an expressive *Java* dialect. The language is designed to make *Java* easier by having a different syntax with better support for "type interface, extension methods, dispatch methods, and lambda expressions" [2]. Given that *Xtext* is a thin layer on top of the *Java Development Kit (JDK)*, it is inter-operable with *Java* and supports all *Java* libraries [5]. Using *Xtend* in the development of *Xtext* is the equivalent of using *Kotlin* for *Android* application development; both languages are inter-operable with *Java* and reduce language verbosity.

2.2 Build Automation Tools

Build automation tools are used in projects which require code to be compiled, packaged, and tested multiple times. An efficient work environment can be setup for automating these tasks so bugs can be identified and each iteration can be evaluated. Moreover, build automation systems address integration problems caused by external dependencies; since software engineering projects are growing increasingly dependent on external libraries, platforms, and frameworks. The build automation systems which were be considered for this project are *Apache Maven*, and *Gradle*.

2.2.1 Gradle vs Apache Maven

At its core, *Gradle* and *Apache Maven* are focused on predominantly supporting development of *Java* based projects; however, they do support other programming languages. Both systems require project configurations files written to the user's specification - *Gradle* uses `groovy` type for it's configuration file whereas *Apache Maven* uses `XML`. The configuration files are responsible for defining the project's external modules, plug-ins, etc. These system's key differenced are outlined below:

1. **Performance:** The differences in working mechanism between *Gradle* and *Apache Maven* results in varying build times for the project. Both systems make use of parallel project building for the project's modules. However, *Gradle* further builds on this mechanism by creating a graph of dependencies which allows for building only modules which have been updated or changed. This feature of incremental compilations for certain classes combined with build caching makes *Gradle* more adequate for large projects in strict need of timely completion.

2. **Dependency management:** Both *Gradle* and *Apache Maven* are capable of performing automated resolution, declaration, and usage of dependencies on a project. The systems are similar in the sense that their respective dependency management implementations support caching external dependencies, and handling dynamic and transitive dependencies (which are downloaded through artifact repositories). However, *Gradle's* has extended functionalities in this field as it allows for easy replacement of dependencies with external dependencies and it also keeps a record of repository metadata and dependency cache which could be used for version conflict resolutions. The metadata is commonly referred to as "information about information".

3. **Plug-in availability:** Both systems have a wide range of available plug-ins. However, *Apache Maven* has been around for a longer period of time and as a result it has accumulated more plug-ins.

4. **Directory structure:** Both systems use a standardized layout for directory structure. However, *Gradle* provides further support for user's that require custom directory structures.

The project definitely has a to use a build system as Eclipse gives the option to setup an *Xtext* project with web integration support only if a build system is specified. The web integration tools can be used for generating a server for deploying the web application and it also automatically generates a sample web application with an IDE associated with the project's DSL; this will make development of the web application simpler.

2.3 Web Application Deployment

Java-based web applications are typically deployed using a Web Application Resource (`WAR`) file. Deployment, in this context, refers to the activities responsible for making an application available for use and the `WAR` file is a collection of all the dependencies needed to run the web application. Included in the `WAR` file are Java Archive (`JAR`) files, Hyper Text Markup Language (`HTML`) files, Cascading Style Sheets (`CSS`), and other dependencies.

The deployment of the `WAR` file is typically done through application servers which operate the applications services for the end user. The most popular web application servers for *Java* applications are *Tomcat* and *Jetty*. Both of which will be considered for the project's implementation.

2.4 Existing Drone Programming Software

There are existing applications which are tailored for drone programming. A popular application is *Tynker* which is designed to work on a tablet and has support for 6 different *Parrot* drones. The application is available on the *Google Play Store* for *Android* devices and *Apple's App Store* for *iOS* devices and has multiple features such as a built in controller and a class room mode that could be used to introduce kids to programming.

Figure 2.2 shows the user interface for *Tynker's* classroom mode. The application has support for autonomous flights where the user can drag and drop blocks of code to the main program. When the start button is pressed the drone will execute the commands. Moreover, there is also features such as: repeating actions using loops, using an "actor" block which the user can use for executing code at any time throughout the flight, connecting to multiple drones and choosing which one executes the code, and more. Some of the commands sent to the drone take in an integer, with some commands being time based (such as "up for **1** second"), some take absolute values (such as "repeat **5** times"), some take degrees (such as "turn right by **180** degrees"), and other take no values at all (such as "flip **flip forward.**"). The user interface seems really straight forward and visually appealing, this is an important factor to be taken into account as it can captivate a kids attention. Furthermore, the use of code blocks makes the sequential execution of commands easier to understand and follow.

Another application that facilitates drone programming is *Scratch*. *Scratch* is an open source application, developed by Massachusetts Institute of Technology, that also works through code blocks which can be dragged and dropped to create a program. *Scratch* has applications compatible with the operating systems *Windows* and *macOS*. While it does not have built-in support for programming drones, like *Tynker*, you can download extensions from the official *Ryze Robotics* web site to makes it compatible with the drone *Ryze Tello*. The extensions have similar functionality to that seen from

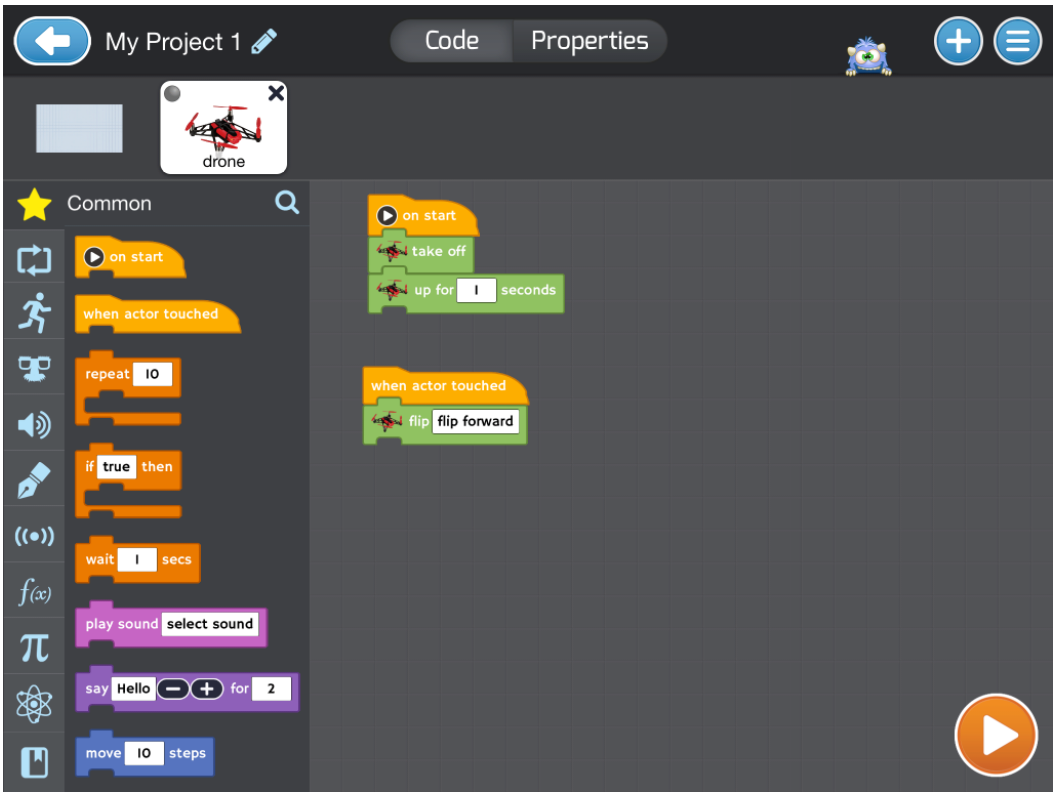


Figure 2.2: Tynker - class room mode. Image taken from [6]

Tynker where drones can be programmed for autonomous flights using time based commands and commands can be run in loops. Furthermore, using *Scratch* with the *Ryze Tello* extensions also gives the user the ability to map keyboard keys to certain actions; essentially the user can program his own controller.

Figure 2.3 shows the user interface for *Scratch*. The drag and drop feature is very similar to that shown in *Tynker* and the application is also designed to introduce kids to programming using a graphical language. The application is not as visually appealing and the extension does not have support for managing multiple connected drones, like in *Tynker*.

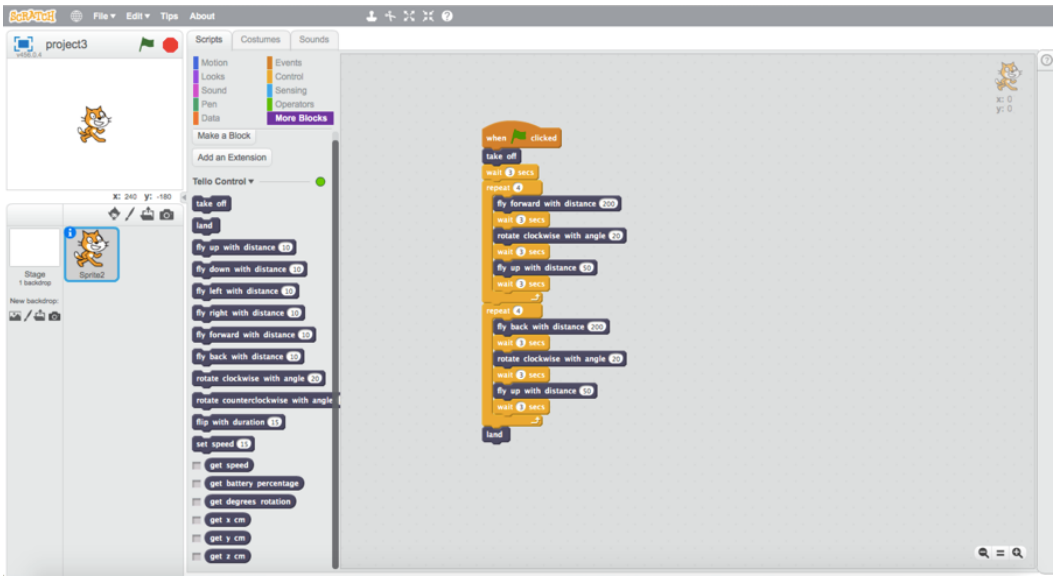


Figure 2.3: Scratch with extensions for Ryze Tello drone. Image taken fron [7]

2.5 Parrot AR Drone 2.0

The drone that was used throughout this project was the *Parrot AR Drone 2.0*. Figure 2.4 shows the drone's dimensions with and without it's protection layer for indoor flights. This drone is relatively inexpensive and at the time of writing costs approximately 99 pounds making it ideal for teaching environments or hobbyists that do not want to spend a large sum.

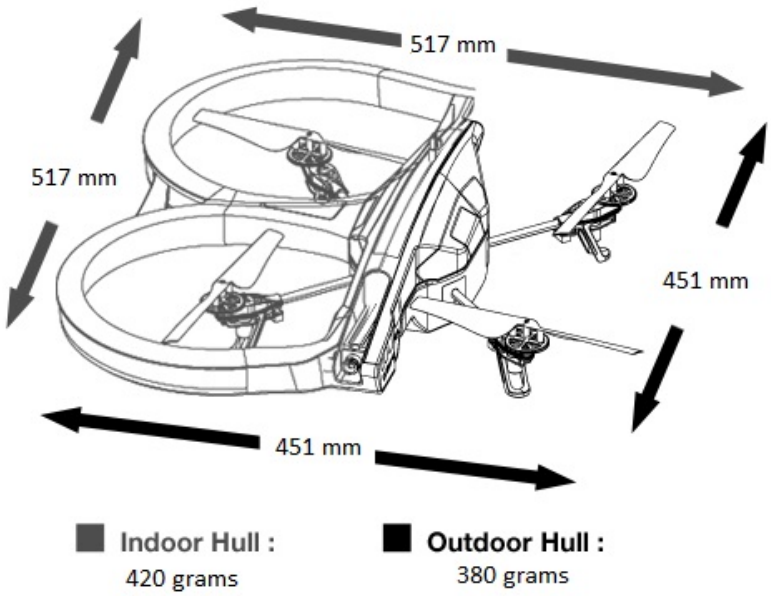


Figure 2.4: Dimensions of the drone that was be used for the project. Image from [8].

The drone interfaces with a PC through a WiFi module and has a maximum range of

50m. The drone's access point is pre-configured with a static IP address (192.168.1.1) and uses the default ports 5554 (UDP), 5555 (TCP), and 5556 (UDP) for communication of navigation data, an `h264` video stream, and AT commands which control the movement of the drone [9]. In networking the IP address is a unique string used to identify a device in a network and the port is an endpoint used for a specific network service, such as streaming data. UDP (User-Datagram Protocol) and TCP (Transmission Control Protocol) are two protocols used for sending data; where TCP is connection-oriented and ensures transmission of data at the cost of latency, and UDP is connectionless and favours lower latency over reliability. The `h264` refers to a compressed video format and the AT commands are commands sent using text strings.

The drone also has the following built-in sensors: a 3-axis gyroscope, a 3-axis accelerometer, ultrasonic and pressure sensors, and 2 cameras. Both cameras can be used to take images and photos but they can't both be used at the same time, this is a consideration that was taken into account in the development of this project. The front camera can record and take pictures or record at 720p while the bottom camera is a 240p camera.

A *NodeJS* library (*node-ar-drone*) was used as it allows for high level programming of the *Parrot AR Drone 2.0*. This library gives access to the drone's features, such as access to the camera feed, and also makes it easier to define an autonomous flight sequence.

2.6 OpenCV

One of the project's aims is to add support for higher level libraries to add functionality to the DSL. *OpenCV* is one of the open source libraries which could provide functionality in the field of Machine Learning and Computer Vision. These fields refer to how a computer can process data (e.g. visual data) and process it to determine it's contents by detecting features and patterns. With reference to drones the *OpenCV* library can be used with the *node-ar-drone* library to classify the content of the image or video stream.

OpenCV is written natively in *C++* and it has official interfaces with *Python*, *Java*, and *MATLAB*. The DSL in this project is mapped to *NodeJS* because of the *node-ar-drone* library hence the official interfaces can't be used. However, unofficial *OpenCV* bindings for *NodeJS* are available and can be used, while they do not provide full support for all available algorithms they do support some object detection and object recognition algorithms. These can be used for defining flight sequences that are dependent on detected objects by doing real time image processing from the drone's images.

2.6.1 Object Detection

Object detection in *OpenCV* uses image processing to identify objects that can be categorized under a specific class. For applications that require real-time image processing a common approach is using Haar Cascades classifiers. These classifiers make use of a list of stages consisting of weak learners [10]. A weak-learner is itself a classifier but it has poor detection accuracy and is prone to false positive classification; where false positives refer to classifying something as being present in the image when in reality it isn't.

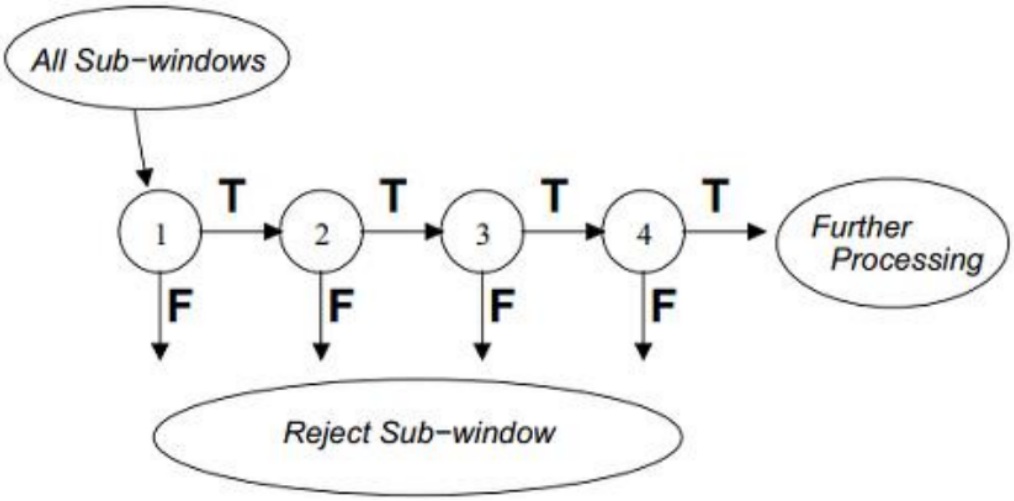


Figure 2.5: Example of a Cascade Classifier. Image taken from [10].

Figure 2.5 shows an example of the stages in a Cascade Classifier. The "sub-window" refers to a segment of the image that is being processed as the algorithm doesn't scan over the whole image, instead it looks at specific regions. These regions are tested for a specific feature throughout each individual stage and the region is discarded if the weak-learner returns a false classification, the sequential testing explains why it's called a "Cascade". An example of the features to be tested is detection of an eye when trying to detect a human face [11, 10]. Lastly, if all the stages return a positive classification then the cascade classifier predicts this region to have the targeted object.

The reason this algorithm is important is because it has been shown that it doesn't require a lot of computational power and hence can be used for real-time detection. Examples of real-time detection using Cascade Classifiers include vehicle detection [10], hand gesture recognition [12], and live human detection on CCTV footage [13]. Furthermore, *OpenCV* gives the user the ability to use pre-trained models as well as train a custom Cascade for detection of specific objects.

The project makes use of *OpenCV's* objection detection using Cascade Classifiers. However, this module was treated as a black box, where the inputs are known and the output type is also known, but the exact theory was not looked into. If interested in the theory and mathematics behind the image processing please refer to [11].

2.6.2 Object Recognition

OpenCV also provides support for object recognition. To differentiate between Object Detection and Object Recognition we can think about the questions they answer. The former answers, “is this object found in this image?”. While the latter answers, “where is this object within this image?”.

Two popular object recognition algorithms supported by *OpenCV* are template matching and feature matching. In template matching a sliding window is used to scan across the image and compare that window to a reference image. *OpenCV* has multiple built in methods to compute a similarity score between the reference image and the tested image. The coordinates of the area with the highest matching probability are returned along with the similarity score.

Figure 2.6: Example of template matching with a sliding reference image. Image taken from [14].

Figure 2.6 shows the sliding window, *OpenCV’s* “matchTemplate” function slides this window one pixel at a time until it iterates through the whole image and returns the area which matches best. While this technique can be used to check if a setting or an object is found within another image, it has several limitations; *OpenCV’s* template matching does not account for distortions or transformations. As a result, this algorithm will likely give poor results if used for object detection in a real-time environment with a moving drone due to image scaling, camera angle, and fish eye distortion.

OpenCV provides more sophisticated ways of Object Detection that are more robust to image distortions, namely Feature Matching. Feature Matching works using keypoints and descriptors. Where keypoints are areas with distinctive invariant features and descriptors are arrays of numbers corresponding to the contents of these keypoints.

Figure 2.7: Example of feature matching. Image taken from [15].

Figure 2.7 shows an example of feature matching where the keypoints from one image are cross checked with the keypoints from another image to test the similarity between the two images. The measure of similarity between keypoints is computed using the distance between the descriptors. A common misconception regarding the "distance" parameter is that it measures the spatial distance between keypoints; the distance between two descriptors is actually a metric for measuring dissimilarity; which is either the Euclidian distance or the Hamming distance depending on the type of descriptor used. Euclidean distance is used to compare descriptors whose data is represented as floating points, Equation 2.1 shows how Euclidean distance is calculated.

$$d_E(a,b) = \sqrt{\sum_{i=1}^{n}(a_i - b_i)^2} \tag{2.1}$$

For example the Euclidean distance for the descriptors a and b with the contents $(12, 36.2, -8, 9)$ and $(11.4, 87, 0, 5)$ respectively is:

$$d_E(a,b) = \sqrt{(12-11.4)^2 + (36.2-87)^2 + (-8-0)^2 + (9-5)^2} \tag{2.2}$$

$$d_E(a,b) = 51.58 \tag{2.3}$$

On the other hand, Hamming distance refers to the number of changes needed to change the contents of a vector to match another vector, it is typically used with descriptors that store data in binary form. For example the Hamming distance for the descriptors a and b with the contents $(1101, 0000, 1111)$ and $(1100, 1111, 1010)$ is 7; since the string in the first index needs 1 changed bit, the string in the second index needs 4 changed bits and the string in the third index needs 2 changed bits.

Put simply, two descriptors are identical if the distance between them is 0 and the distance increases depending on the variance of the descriptors' contents. *OpenCV*

has a framework called “features2d” which allows for the use of Feature Matching and can be integrated with the DSL for object detection.

Similar to the Cascade Classifiers, Feature Matching was also used in this project without looking deeply into the theory; this algorithm was treated as a black box throughout the full development of the DSL and web application.

2.6.3 Image Resolution and Computer Vision

An article on face detection and face recognition provided some useful information on the impact of image size on detection accuracy and recognition accuracy [16].

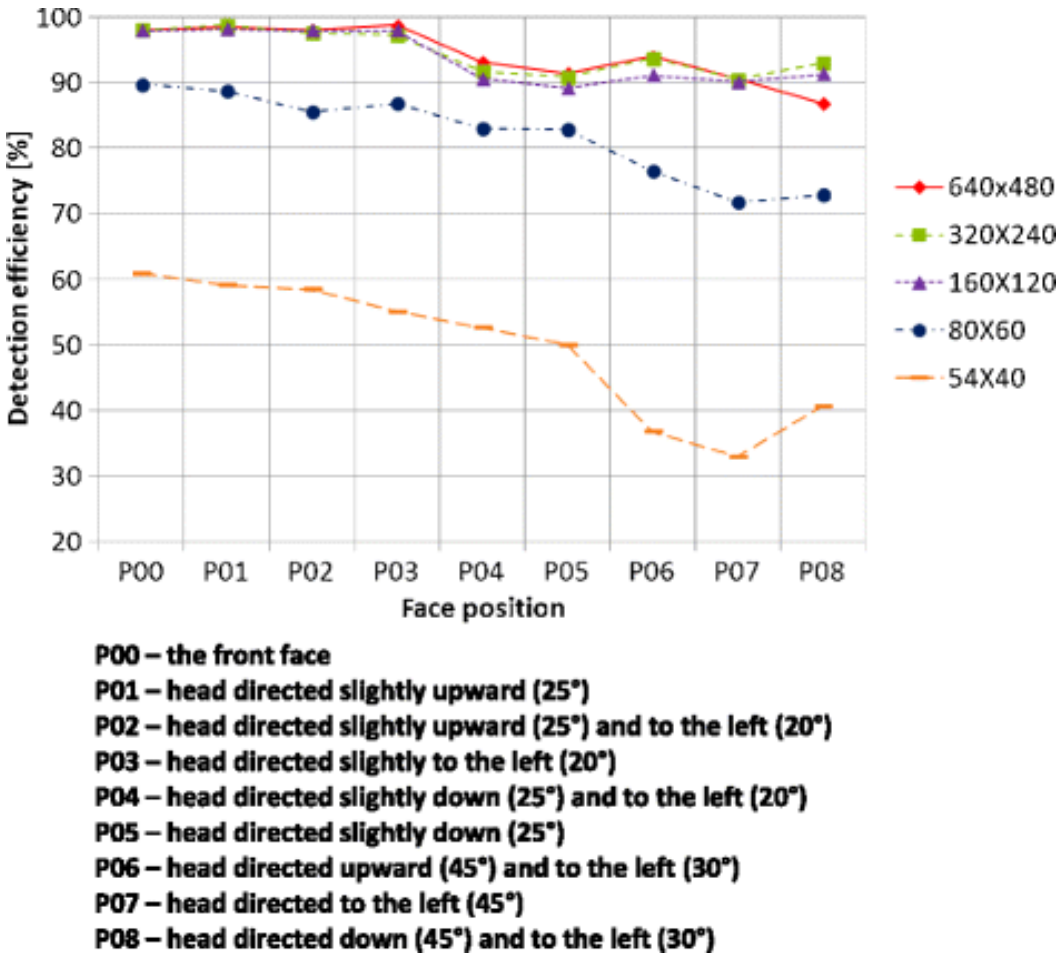


Figure 2.8: Impact of resolution on detection accuracy. Image taken from [16]

Figures 2.8 and 2.9 show their results. There was negligible impact on recognition accuracy with their test data and the resolution they tested. However, their tests for detection accuracy show more variation; there was a 10-20% drop in detection accuracy when reducing the image size, in pixels, from 640x480 to 80x60 and a 30-40% drop when reducing the image size, in pixels, from 640x480 to 54x40.

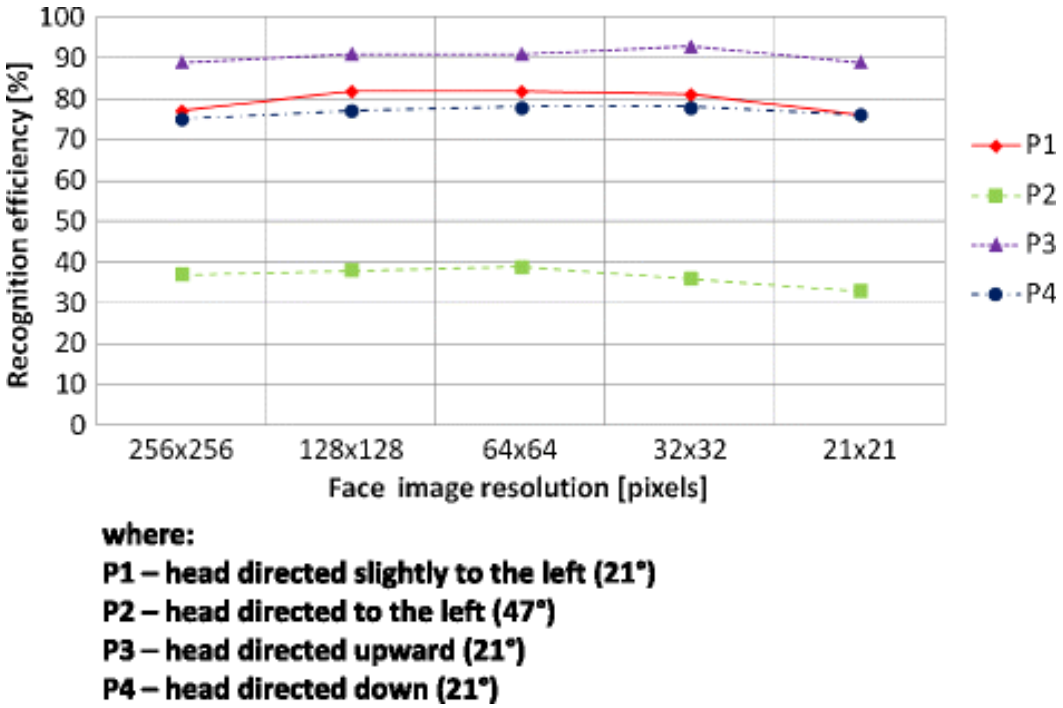


Figure 2.9: Impact of resolution on recognition accuracy. Image taken fron [16]

These results are important because if the project makes use of image processing then images could be resized to reduce computation time without significantly affecting the classification accuracy. The computation time is an important factor for the project because the drone may be required to do image processing in real-time.

2.6.4 Drones With Face Detection

The following uses were found to be interesting applications for drones:

1. **Hover Camera:** In the Consumer Electronics Show 2017 a drone with a built-in facial recognition feature was demoed by a company called *Zero Zero*. The website *Engadget* described this pocket-friendly drone as a selfie-drone that "finds and auto-records its owner" [17]. This is an interesting feature as it can act as catalyst for mass adoption of computer vision powered drones by making social media users a target audience.

2. **Rescue Robot Competition:** There is competition called *RoboCup Rescue* where the usability of robots in post-disaster situations is evaluated. In the 2011 event the *RoboCup Rescue* paper mentions about victim search using face detection and the potential use of the *Parrot AR Drone* [18]. This is a more serious application for drones but it brings potential of saving people's lives.

Chapter 3

Specifications

The intent of following a minimum viable product approach means the project's specification are split between core requirements and extended functionalities. Refer to Chapter 4.1.2 for a justification on this approach. This Chapter lists the identified specifications the project sets out to achieve.

3.1 Core Specifications

These following specifications are listed in descending order of priority.

1. **Development of a DSL:** The language is targeted at facilitating users looking to program their drones to do specific tasks but do not have the technical background for programming using GPLs. The language must have an intuitive syntax and grammar for a tailored user experience and ease of use. Furthermore, the DSL should provide support for the basic autonomous flight commands.

2. **Construction of web application:** A front-end comprised of a web application will be required for the end-user. An embedded *Xtext* module will allow the web application to make use of the DSL for programming the drone. The application will play a key role in the usability of the DSL as well as the user experience, making it a key requirement. A client server architecture must be set-up to fulfil this requirement, for hosting and accessing the application.

3.2 Possible Extensions

The following extensions will be considered if all the core specifications are met and time allows for further development of the DSL. The project is not limited to just these extensions, they are merely possible path for researching.

1. **Support for high level libraries:** The functionality of the DSL could be enhanced through the use of high level libraries. For example, libraries such as *OpenCV*, discussed in Chapter 2.6 could be used for interesting applications such as object detection or recognition.
2. **Error handling:** The project could explore the mechanisms of error handling and recovery; where the user could potentially write unsafe programs but the user can also define the criterion for error handling. For example, the user could have the ability to tell the drone to land in case an error is thrown mid flight.
3. **Support for multiple drones:** The targeted programming device is the *Parrot AR Drone 2.0* but the language could be configured to work on multiple drones by providing support for other done APIs. Having support for more than one device would give the project the potential to serve a larger user base.
4. **Language structure:** The functionality of the DSL could be extended with support for key programming concepts, these include: use of conditionals, user-defined functions, loops, object oriented programming, etc.
5. **Testing and simulation:** The project could provide informative feedback to a user without running the code. For example, in the case of autonomous flights a service could be implemented that steps through the code and outputs a visualisation of the drone's path if it were to be deployed.

Chapter 4

Implementation

This section reflects on the chronological development of the project. The implementation, testing, and optimization is documented in this section from the set-up of the "walking skeleton" until the final submitted product. A user-centric approach influenced some of the "vertical slices" added to the project. 5 dedicated testers were picked to aid in the development, these testers are:

- **1x Managing director at a technology camp**: Runs summer technology camps for kids and has been involved with organising events with drone programming challenges.
- **2x Coding instructors**: They have experience with teaching programming to kids between 5-17 years of age.
- **1x Drone enthusiast**: This user has never programmed before but has owned multiple drones and is familiar with applications and user interfaces typically used for controlling drones.
- **1x Computing science student**: This user has basic experience with multiple programming languages including *Python*, *C++*, and *Q-Sharp*.

4.1 Development Methodology

Software engineering projects often follow a set of principles for delivering a product in an effective manner. The industry standard is to follow the *Agile Manifesto* proclamation where a people-centric approach is used to dictate development and the project is segmented into sub-problems; which are solved iteratively in order of highest priority. A continuous work-flow strategy plays a key role in *Agile* development and it's influence will be considered for this project.

4.1.1 Continuous Integration

Continuous Integration (CI) is a work-flow strategy employed in software engineering projects to allow for smooth integration of distributed work, amongst the project's engineers. The importance of CI is related to the modular nature of modern projects, where complex systems are built through integration of simpler components designed to solve a sub-problem. CI focuses on the integration of these simpler components; a process where bugs and conflicts can be introduced into the system.

Platforms for hosting code, such as *GitHub*, make use of CI by allowing repositories to be linked with CI suites which compile newly added code and tests it with unit tests, these suites are hosted in separate servers. The compilation output dictates whether the code is suitable for merging to the master branch or if it will introduce problems. In the case of failure the developer is immediately informed and is expected to fix the code in timely fashion, so the master branch only has clean code and will compile.

A CI server should only execute code testing if there are changes. This is typically achieved by installing a web-hook for a specific repository which monitors for payloads corresponding to certain events. A web-hook is usually subscribed to `push` events, which on occurrence send an `HTTP POST` payload to the web-hook's URL. The web-hook then triggers the CI testing suite.

Continuous work-flow can also be extended past CI by adding the following extensions:

1. **Continuous Delivery:** This extension focuses on managing the project when newly developed code compiles and passes the integration and unit tests. For the project to be accepted for production it has to be further tested for user acceptance. This is an additional stage to the CI work-flow.

2. **Continuous Deployment:** This is an extension of Continuous Delivery where the code is automatically set for deployment if it passes user acceptance testing. Unlike Continuous Delivery this approach doesn't require a human to deploy the software.

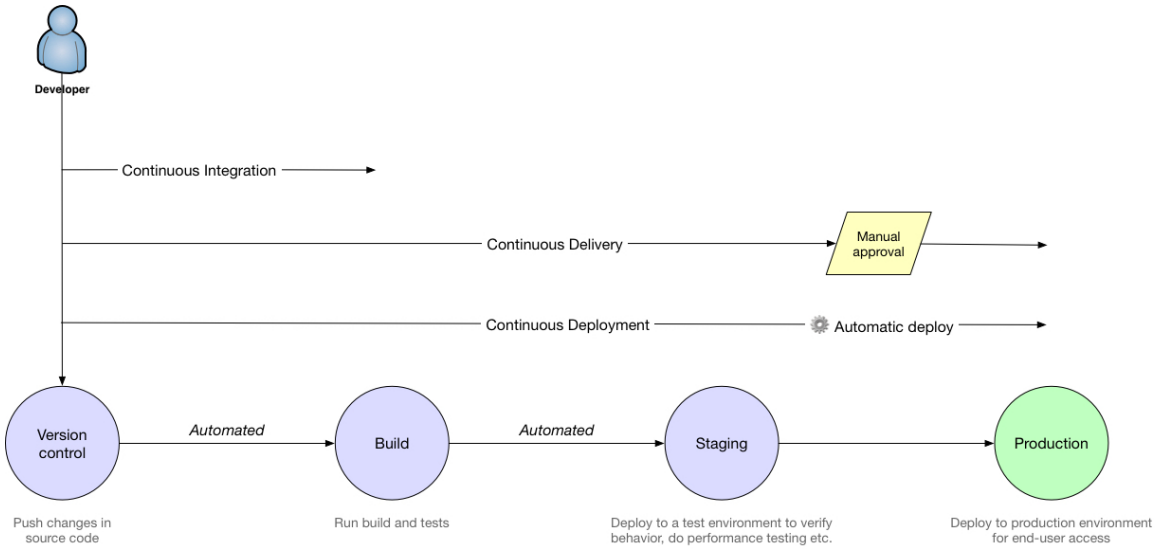


Figure 4.1: Visualization of continuous work-flow strategies. [19]

4.1.2 Work Flow

The tasks outlined below were required for iterative development of the project. The mothodology followed was the "vertical slice approach" where work on a new feature was only started after a previous feature was finished.

1. **Background research:** The project's work must be justified through research. This meant that in the early stages of the project the research was predominantly based on analysis of existing work in the field and familiarisation with tools required for the project. In the latter stages of development primary research was conducted for a user-centric driven development.

2. **Iterative development:** The first "vertical slice" was the first iteration and provided a "walking skeleton" for the project. Once the core had been developed and tested a minimum viable product was the focus of the second iteration; at this stage the product was already showing usefulness. The remaining iterations were focused on feature implementations.

3. **Evaluation:** At the end of every iteration an evaluation of added vertical slices was required to verify that sufficiently valuable incremental changes were made to the different layers in an architecture. This was a recurring task and dedicated testers were used at the end of the project to evaluate the final product.

4. **Documentation:** The implementation details as well as error handling was documented throughout the development stage. This was done to aid in the writing of thesis for this project and provide a context and reference for possible future developers. This was a continuous task throughout the project and was be carried out in parallel with other tasks outlined above.

Throughout project some problem were found when adding new slices. This meant

that backtracking and reworking through previous slices for accommodation of new features was required.

4.2 Walking Skeleton

4.2.1 Architecture 1

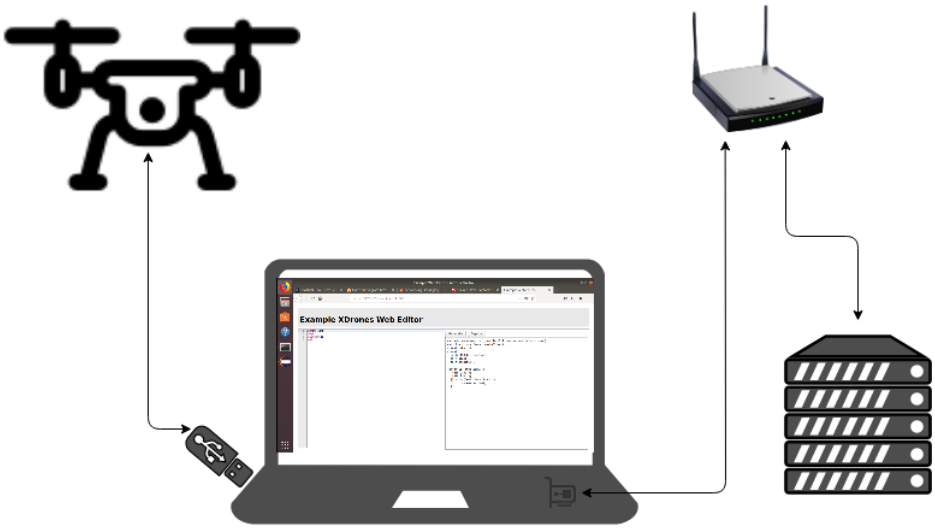


Figure 4.2: First attempted architecture.

Figure 4.2 shows the first architecture that was considered for the project. A server was initially set-up through a virtual machine using the Department of Computing Private Cloud. The intent was to host a web application, with an integrated IDE for the DSL, in the back-end server. The server would then be used to generate the code needed to deploy the drone with the user's commands. Finally, the generated code would be sent to the client's mobile device where it would execute. In this architecture the client's device was required to have two network interface cards (NICs); one for connecting to the drone's access point, and one to connect to the same network as the back-end server so they can communicate.

There was a some flaws with this architecture. Firstly, the only client-side scripts that are normally executed, using the browser, must be written in *JavaScript*. Secondly, even if a solution for sending the *NodeJS* code and executing client-side was found, there would still be compatibility issues as every single mobile device would need to have the required modules installed locally. Since this was a flawed architecture it had to be revised.

4.2.2 Architecture 2

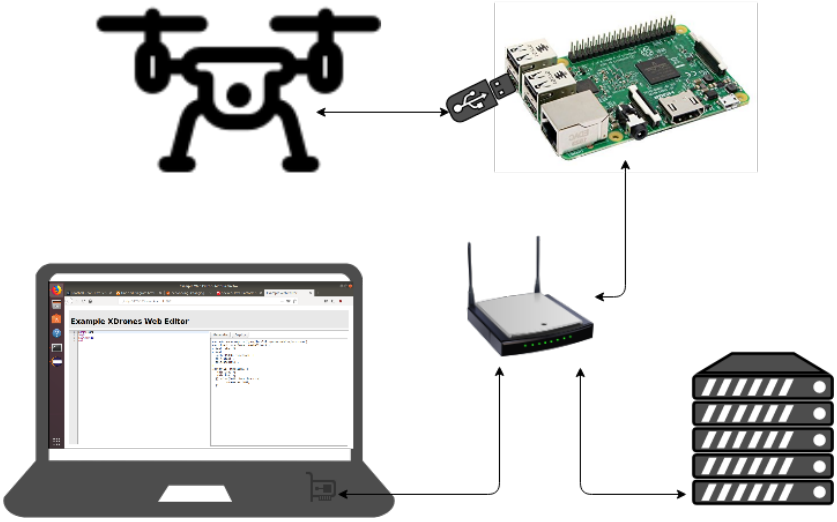


Figure 4.3: Revised architecture.

Figure 4.3 shows the revised architecture where a Raspberry Pi 3B+ was used. This model comes with an integrated Wi-Fi module which was used to connect to the same network as the client's device. An external NIC (a USB Wi-Fi dongle) was used to connect to the drone's access point. In this architecture the Raspberry Pi is used to host the web application through a server listening on the integrated NIC. The mobile device can then access the Raspberry Pi and get access to the web application where the user can use the DSL's IDE. As for the generated code, it can now run locally on the Raspberry Pi where it won't have problems with dependencies and local libraries.

The Raspberry Pi was firstly configured with a Raspbian operating system. It was then checked if the operating system had the project's required dependencies; any missing dependencies were installed. These dependencies include *Java* for the *Xtext* project, *Node*, and the *node-ar-drone* library for controlling the drone. Lastly, the Raspberry Pi was configured with two "wpa_supplicant" files; these are configuration files used for the network interfaces. In one of the files the NIC was associated with the drone's access point, while the other configuration file was used to access the Imperial network since most of the development work was being carried out on campus.

For ease of use, the required libraries were added to an installation script which could be reused to configure another Raspberry Pi, if needed.

4.2.3 Initial Web Application and DSL

The starting DSL and the corresponding generator for *NodeJS* code were taken from the Git repository `ProgrammableDrones/xTextLanguage`. The repository was not forked since it did not work on the development environment. However, their *Xtext*

file, the code generator, and service dispatcher were used as a starting point for this project.

When using the *Gradle* build system on a new *Xtext* project, *Eclipse* allows the user to have web integration support. This option generates an simple web application with an integrated IDE as well as an embedded *Jetty* server which could deploy the application locally. This option was used in order to have a starting point for the server and the corresponding web application. The starting web application does not offer functionality outside of the IDE; to make the web application more useful a deployment button was added which uses the service handlers to run a bash script which deploys the drone with the latest generated code.

4.2.4 Server Configuration

The usual method of deploying a web application is by packaging the application in a `WAR` file and using servlet containers such as *Jetty* or *Tomcat* to host the application (refer to Section 2.3 for further explanation). When this was attempted the IDE was not working and a personal lack of understanding for application deployment meant this problem was not solved. A solution was found in the *Eclipse* documentation on embedded *Jetty* servers which quotes, "Dont deploy your application in Jetty, deploy Jetty in your application!"[20].

Hence this alternative option was considered, in this case the embedded *Jetty* server that was automatically generated could be configured to listen on interfaces other than the loopback network interface; services running on localhost. This was done by writing a function that gets the address of the first NIC in the Raspberry Pi and binds the socket (endpoint for connection) to this address. Notice that in the revised architecture the first NIC (integrated) on the Raspberry Pi was the one used for serving the application through a network which the front-end devices can access.

Because the application was not deployed using a `WAR` file the Raspberry Pi does not need installation of *Jetty* or *Tomcat*. *Eclipse's* web integration support generates a *Gradle* task called "JettyRun" which can be used to launch the server on the Raspberry Pi. Moreover, the Raspberry Pi does not need to have *Gradle* installed in it to run the server as *Eclipse* adds a *Gradle Wrapper* to the project; this is a script which can be used to execute *Gradle* builds or tasks without manually installing *Gradle* to the device [21].

4.2.5 Continuous Integration

Notice on Figure 4.3 the back-end server was kept despite not being explicitly required. The server was still kept to run automatic *Gradle* builds and a parsing test for the DSL whenever new work is pushed on to the git repository `MGafar/xDrone`. This was done to ensure that the repository is kept clean whenever new code is pushed to it.

Because the CI server was configured on a virtual machine running on the Department of Computing Private Cloud it can't be easily accessed without a VPN or being inside the network itself. This was an obstacle with CI builds as they are typically designed to work on push events (to the repository). To get the CI server to work the following components were needed:

1. **Jenkins**: a software for running the CI build.
2. **Ngrok**: a program that can expose a service on a closed network, using introspectable tunnels.

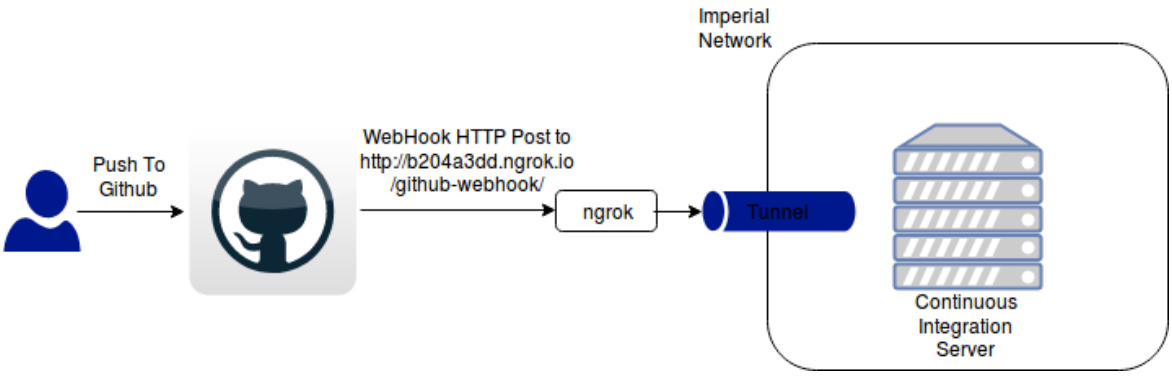


Figure 4.4: CI set-up

Figure 4.4 shows a visualisation of the CI used for this project. The URL in the figure is the actual URL that was available publicly for the Github webhook to post an `HTTP` request on `push` events. The request would then be forwarded to the server in the Imperial network. Allowing public access to the internal network could be a security issue, hence approval from the Imperial Computing Support Group was required for the use of *ngrok* to tunnel to a virtual machine inside the Imperial network. They did not see any issue and allowed the project to go through without having to change this.

4.3 Face Detection

One of the project's aims was to add support for higher level libraries; a library that was explored was *OpenCV* as it provides support for computer vision.

4.3.1 Dependencies and Implementation

Section 2.6 discusses the official bindings for *OpenCV* and how there isn't support for *NodeJS*. Therefore, unofficial bindings found in the repository `peterbraden/node-opencv` were installed in the Raspberry Pi and tested. To do this *OpenCV 2.4.9.1* and it's dependencies were installed, since there is support for this version for the

Raspberry Pi. The unofficial bindings were then added and the examples found on the repository were used for testing, to see if they run as expected.

Section 2.6.1 discusses object recognition using cascade classifiers. This was the first *OpenCV* feature that was explored, using example cascades classifiers for face detection. This feature was not influenced by the user testers; however, potential applications of face detection for drone's has been discussed in Chapter 2.6.4. The use of face detection for drones can be made easier for adoption through DSLs.

The use of face detection for the drone was done by passing images from the `PNG` stream from the drone, using the *node-ar-drone* library, to a pre-trained face detection cascade classifier. In order to have real time image processing for face detection the function that takes the last image from the buffer and tests it with classifier had to run in regular intervals. The interval time was set to 200ms to match the frame rate, 5 frames per second, of the `PNG` encoder from the *node-ar-drone* library.

4.3.2 Optimization

The initial implementation had multiple problems. The first being an error that would sometimes be thrown by the function that continuously tries to detect faces. The solution to this problem was to first apply a 5 second delay to the function since it takes a while for the `PNG` encoder to start running. Secondly, in *JavaScript*, functions that are scheduled using the "setInterval" method do not guarantee that there is a delay between executions. The solution to this was to use scheduling using recursive "setTimeout" functions as explained in the *JavaScript* documentation [22]. This made the program a lot more robust and prone to crashing.

A test program was run where the drone would take off and wait until a face was detected; upon detection it would land. In this test problems with false positive classifications were seen; where the drone would land indicating that a face was detected but in fact there was no faces in the frame. The solution to this was to make the face detection go through more classification stages; since another example cascade that is available with *OpenCV* is for eye detection this was added to the program. Subsequently, a false positive detection would require both the eye detection classifier and the face detection classifier to have an erroneous reading.

4.3.3 Integrating With DSL

In order to use the face detection a keyword "FACEDETECT" was added to the DSL. The user only has the option to declare the keyword outside the main sequence of the flight. Following the "FACEDETECT" a user can declare further commands which will execute on the event of face detection.

The generator was then changed to use chained promises and delays to define a flight sequence. In *JavaScript* chained promises refer to processes which can run in

a defined sequence. The reason this was required was because the flight sequence should be able to break on the event of face detection; *JavaScript* allows you to skip the execution of block in a promise chain using a method called "Promise Resolution". The "Promise Resolution" can hence be executed if a face is detected. Lastly, another set of chained promises was added; it's execution corresponds to the commands the user declares after the "FACEDETECT" keyword; this execution only runs if a face is detected.

4.3.4 Full Architecture Testing and Further Optimisation

When testing on the full architecture it was noticed that the Raspberry Pi was executing the face detection process at a much lower rate compared to the defined scheduling time of 200ms. The likely reason for this was a time-out caused by lack of computational power, since the same code was working correctly on a laptop.

In order to overcome the problems with computational power one of the options was to increase time-out so that the function has more than 200ms to finish execution; the downside of this would be that the image processing wouldn't be as quick. Another option was to try to make the computation less intensive for the face detection. The second option was deemed more reasonable and hence the factors that affect computation time were explored. One of these factors was image resolution which when lowered can reduce the computation time without significantly affecting the detection accuracy, refer to Chapter 2.6.3. The image resolution was not lowered directly from the stream's configuration but instead it was lowed using *OpenCV's* image resize function; the image was reduced from 480x360 to 120x90 pixels. This implementation allowed for a lower resolution image to be processed for face detection without affecting the stream for other potential features that are reliant on the camera feed.

While resizing the image helped increase the execution speed it was still slow. Hence, further optimisation was done was by removing the second cascade classifier which was detecting eyes. Initially the second cascade classifier was added to reduce false positives but this was resolved by changing the detection criterion to a positive classification being required in 3 consequent frames. Rerunning the test program on the Raspberry Pi showed that the face detection was now operating at a significantly faster rate and false positive classification was not having as big an impact.

4.3.5 Regression Testing

Regression testing was done to ensure the addition of this feature would not impact any other parts of the program. The testing showed that nothing was affected with the addition of face detection.

4.4 Feature Matching

One of the problems faced with using Cascade Classifiers was the need for training when trying to identify a random object. While *OpenCV* provides some pre-trained models for face detection, the user may want to detect objects or settings other than humans. The first option for this would be to give the user the ability to train a custom Cascade using the web application and DSL. However, training a custom cascade requires hundreds of positive samples (images with the object you're trying to detect) and thousands of negative samples (images without the object you're trying to detect)[1]. These samples need to be verified manually then *OpenCV* can be used to train the Cascade, this is lengthy process and can take several hours. Chapter 2.6.2 shows two of *OpenCV's* algorithms that could be used as alternatives to overcome the need to have a large dataset and go through a lengthy process of training data. For the reasons outlined in Section 2.6.2 Template Matching was not explored and the project looked directly into Feature Matching.

4.4.1 Dependencies and Implementation

The *NodeJS* bindings for *OpenCV* do not have support for visualizing the distinct features on a reference image and it's closest matching feature on another tested image. Hence for visualization purposes the *OpenCV-Python* bindings were used to write a script which shows examples of matched features.

A single picture of a coffee cup was taken using the drone and this image was compared with other pictures of the same coffee cup in different angles and distances. In all examples the image on the left is kept constant and only the 50 best matching points are shown.

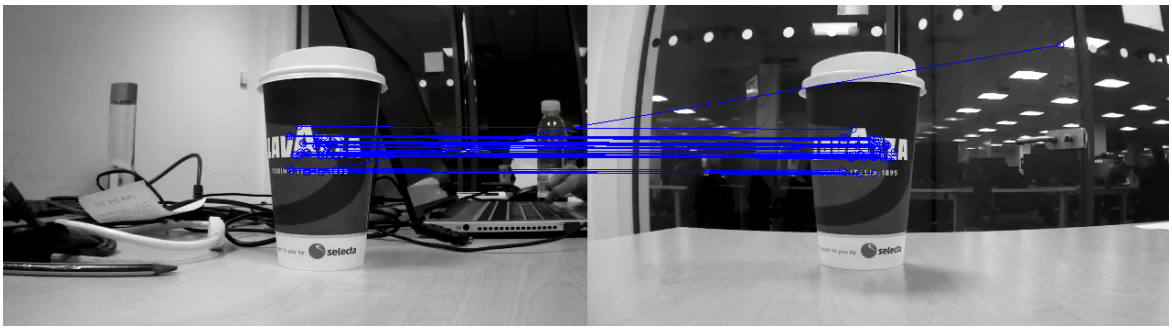


Figure 4.5: Feature matching example with coffee cup at 30cm from the drone's camera in both images

Figures 4.5-4.10 simulate scenarios where the images from a drone are used to detect a coffee cup. This includes examples which take into account of scaling by having

[1]A large dataset is not a hard requirement, however smaller datasets lead to poor detection performance.

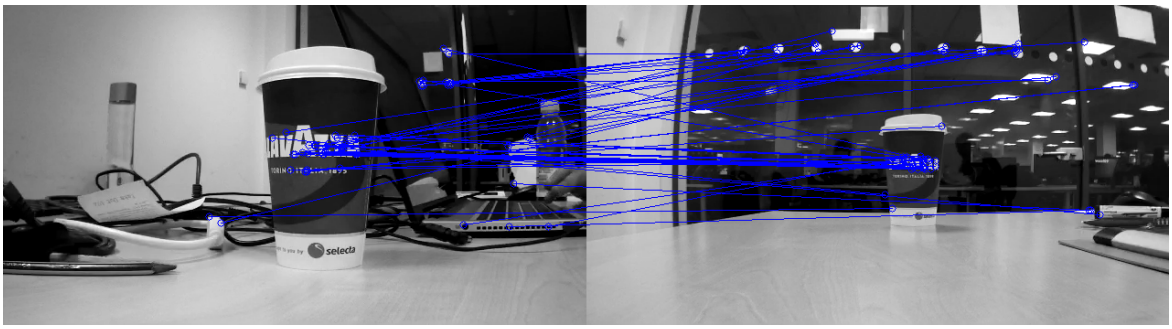


Figure 4.6: Feature matching example with coffee cup at 30cm from the drone's camera in the image on the left and 60cm for the image on the right.

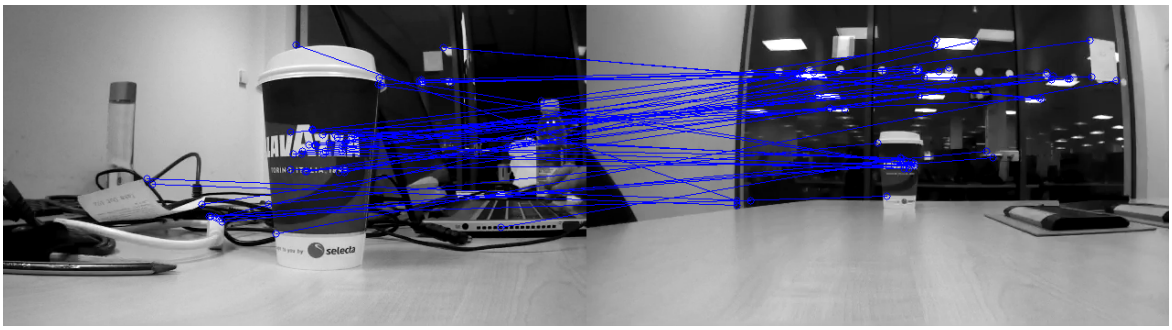


Figure 4.7: Feature matching example with coffee cup at 30cm from the drone's camera in the image on the left and 90cm for the image on the right.

the cup at varying distances from the drone as well as examples where the drone is held at different angles. An example of fish-eye distortion is also shown where the cup is very close to the drone's camera and as a result the image is distorted.

To be able to use the Feature Matching a measure of similarity is required. Section 2.6.2 explains how the "distance" parameter is used in *OpenCV* to determine how closely matched two keypoints are. In the *Node-OpenCV* library there is function which extracts and matches keypoints from two images; the function is already optimized to filter out bad matches then it returns the mean value for distance of the good matches. This will be used for testing the matching in different conditions and a suitable dissimilarity score will be used to trigger a sequence of flight events if a desired object is detected.

The initial testing was done using the same images from Figures 4.5-4.10. The reference image was kept constant and the test images were measured for dissimilarity.

The results in Table 4.1 shows how feature matching can adapt to scaling. There is a small difference in the dissimilarity score when moving the cup back to 60cm from the drone's camera, and a more noticeable difference when moving back to 90cm. On the other hand, the testing for examples with a different angle and an at a distance where the cup is distorted due to being too close to the camera showed that the image similarity function react poorly in these conditions, with their dissimilarity

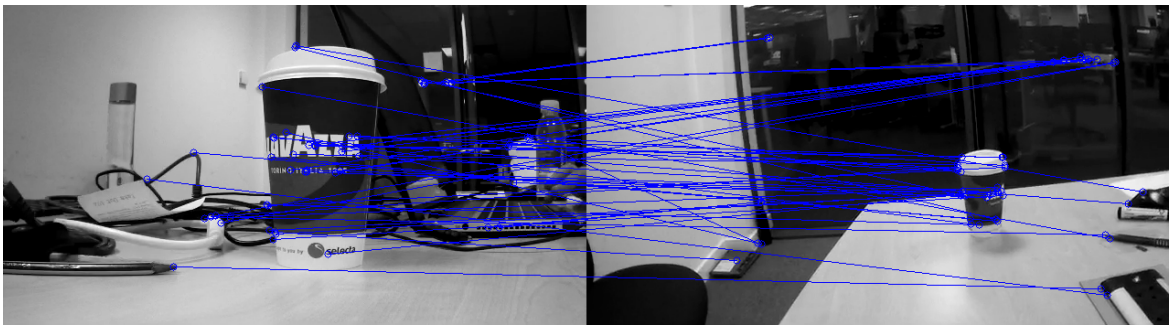


Figure 4.8: Feature matching example with coffee cup at 30cm from the drone's camera in the image on the left and 60cm for the image on the right. The drone was held at an angle to see how the feature matching is affected.

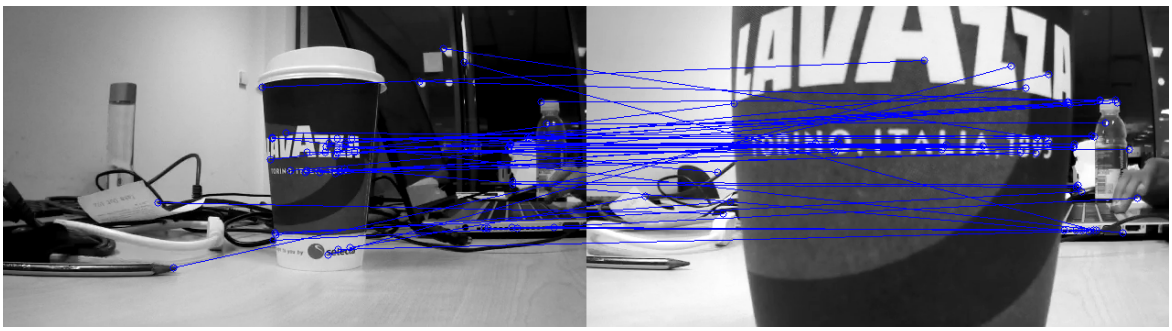


Figure 4.9: Feature matching example with coffee cup at 30cm from the drone's camera in the image on the left and 10cm for the image on the right. This example serves to show the effects of "fish-eye distortion" and how it affects the feature matching.

Figure Number	Condition	Dissimilarity
4.5	Cup is at 30cm from camera but the surrounding objects are different.	26.75
4.6	Cup is moved to 60cm from the camera to test the effects from scaling.	30.4
4.7	Cup is moved to 90cm from the camera to test the effects from scaling.	40.54
4.8	Cup is kept at 60cm from the drone but the drone is held at an angle.	62.96
4.9	Cup is moved to 10cm from camera to test effect of fish-eye distortion.	58.20
4.10	No cup is present in the tested image.	62.50

Table 4.1: Testing feature matching different conditions.

score being close to the that of the control image (without any cup present). The results from this initial testing was used to set a reference score for detection; where the value 40 would be used as a successful detection.

The next step was to interface the PNG stream from the drone with the image similarity function in *Node-OpenCV*. A recursive "setTimeout" function was written to test

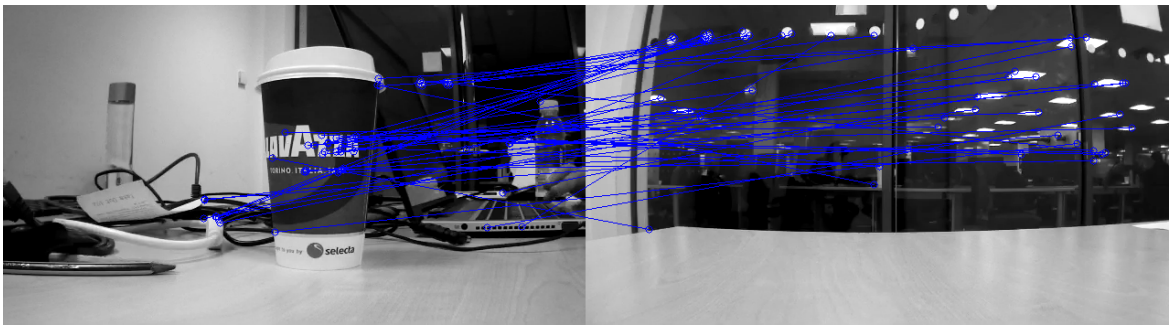


Figure 4.10: Feature matching example where the object to be detected is not in the image.

the similarity of the latest image captured by the drone, compared with a reference image, in regular intervals. The function appeared to work at first but when testing it in the middle of a flight an *OpenCV* assertion error would crash the whole program. Some investigation revealed that the program would crash in scenarios where the drone's camera was facing a white wall or if the drone's camera was covered. It was later discovered that others in the community were having the having the same problem and have posted about it in the *Node-OpenCV* repository issues page. This error was caused due to the image similarity function not handling featureless images so the descriptor matcher was throwing an error; a solution to this problem was found on a forked repository that solves this issue and simply returns a "null" dissimilarity score in the case that the feature extractor does not find any features [23]. At this point the Raspberry Pi was configured to use this forked repository instead to progress through the project.

Further testing was then carried out by using the console output to show the dissimilarity score of every tested frame. In this case it was shown that pointing the drone to areas without distinctive features (e.g. a white wall) the dissimilarity would show a "null score, while areas without the reference object would show a score between 40-100 with occasional dips below 40. Hence, a dissimilarity of 30 or below was deemed suitable for a positive classification; this formed the condition that triggers execution of the drones commands in the case of a positive detection.

4.4.2 Changes to DSL and Regression Testing

Initially, a keyword "FEATUREMATCH" with an associated string "image_name" was added; where "image_name" refers to an image name that can be found in a gallery directory. A block surrounded by curly brackets was also added to allow the use of the basic drones commands on a successful feature match event. These were used in the code generator to generate the appropriate code. This implementation was not suitable due to computational power of the Raspberry Pi, as testing with both feature matching and face detection at the same time would run on the development environment (laptop) but would crash on the Raspberry Pi. The solution was to limit the DSL to only accept one global conditional where the drone would either scan for

face detection or it would try to do feature matching between the latest frame and a reference image.

4.5 Viewing Generated Code

At the beginning of development of the web application a division was added in order to show the generated *NodeJS* code that would execute server side. This was done as a way to see what the program was doing and this was also helpful for debugging. In one of the encounters with a user tester (a coding instructor), the tester was shown a demo of the web application and he had some concerns regarding the display of generated code. The user mentioned that for younger kids showing the generated code can be both confusing and intimidating. On the other hand, it is a common exercise to ask the older kids to identify a segment of the code that is responsible for a particular action. Furthermore, the user also mentioned that the generated code felt out of place compared to the code shown on the IDE.

To tailor the web application for all ages the generated code was hidden behind a tab. Moreover, for users that may want to see the generated code, syntax highlighting and line numbers were added for the code. This was a simple change as *Google's* "prettify" script provides support for this specific issue [24].

```
var arDrone = require('/usr/lib/node_modules/ar-drone'
var cv      = require('/usr/lib/node_modules/opencv');
var http    = require('http');
var fs          = require('fs');
var face_cascade = new cv.CascadeClassifier('/usr/lib/
var promise         = require('/usr/lib/node_modules/p

//var option = new Object();
//option.imageSize = "1280x720";
//var client = arDrone.createClient(option);
var client = arDrone.createClient();


client.config('video:video_channel', 0);


var pngStream = client.getPngStream();
```

(a) Before

```
1. var arDrone = require('/usr/lib/node_modules/ar-drone
2. var cv      = require('/usr/lib/node_modules/opencv')
3. var http    = require('http');
4. var fs          = require('fs');
5. var face_cascade = new cv.CascadeClassifier('/usr/lib
   /data/haarcascade_frontalface_alt2.xml');
6. var promise         = require('/usr/lib/node_modules/
7. var PaVEParser = require('/usr/lib/node_modules/ar-dr
8.
9. var client = arDrone.createClient();
10.
11. client.config('video:video_channel', 0);
12.
13. //Override camera config if using downward camera
14. client.config('video:video_channel', 3);
15.
16. var pngStream = client.getPngStream();
17.
18. var lastPng;
```

(b) After

Figure 4.11: Changes in visualisation for generated code in the web application before and after using Google's "prettify.js" script.

Figure 4.11 shows a segment of the generated code and how it is more readable after adding syntax highlighting and line numbers. The code is not shown when the website is first loaded, the user must manually enter a "debug-mode" tab which shows the generated code. A "re-generate" button was also added to give the user the ability to regenerate the code if he made changes to the code written using the DSLs editor, this allows the user to see updated code without having to deploy the drone.

4.6 Image Gallery

Given that the program already uses a PNG stream for *OpenCV* image processing, the user should also have the ability to take snapshots using the drone. A user interface should also allow these images to be displayed in a gallery.

4.6.1 Changes to DSL

Within the DSL a "SNAPSHOT" keyword with an associated string "image_name" was added. This allows the users to call the snapshot commands and give the image a name which will be used to save the image to the gallery. This command was also a simple addition to the generator as the only thing required was to use *NodeJS's File System* module to save the last image from the PNG buffer to a gallery directory when needed.

4.6.2 Changes to Server

In order to make the web application dynamic the HTML file corresponding to the images in the gallery has to be regenerated when there are changes to the directory where the images are kept. This required implementing a directory listener that starts when the server is launched and closes when the server is stopped. *Java* provides support for concurrent execution using the *Executor* interface; where concurrent execution refers to running tasks simultaneously. This was deemed a suitable implementation for the directory listener as it wouldn't affect the operation of the main thread, which is responsible for the server. The added executor monitors the directory and if there are changes to the directory it will regenerate the HTML file with the suitable changes. The monitor was scheduled to check the directory every 2 seconds.

4.6.3 Additions to Web Application

In web application the *AJAX* framework can be used to update parts of the web application by loading scripts from the server. For the web application this was used to load the HTML file (lightbox.html) that describes the gallery content. Furthermore, scripts were used for presenting the gallery in a user-friendly way as well as providing support for full screen viewing and scrolling through the pictures[2]. These scripts are not personal work and were adapted from a tutorial found in *w3schools* [25].

The web application also had to take into account of snapshots taken after the web application had loaded; since the DSL provides support for taking snapshots. In this case the web application division for the modal gallery has to be reloaded if

[2]Refer to Appendix C.4 if interested in the way the gallery looks.

the HTML code has changed. Instinctively, the first solution that was thought for this was regularly polling the server, every 5 seconds, and reloading the gallery content using *AJAX*. This solution worked but it was considered to be costly when taking into account of the network traffic.

At the time of testing *Firefox's* network monitor was reporting a data transferred of 4 kilobytes when fetching the lightbox HTML file (the size increases or decreases depending on the number of images present). This meant that for this specific service the data consumption, over a period of 5 minutes (300 seconds), can be calculated as follows:

$$Consumption = \frac{TotalTime}{PollInterval} * SizeofHTML \tag{4.1}$$

$$Consumption = \frac{300s}{5s} * 4kB \tag{4.2}$$

$$Consumption \approx 240kB \tag{4.3}$$

A method of reducing the data consumption was explored where the server was polled using an exponential back-off function. This is a type of function that when finished executing waits for an interval then restarts; the interval increases exponentially after every execution. This function was configured to run when the deploy button is pressed, since this is the only time a new photo can be taken, and it starts with an interval of 1 second; the interval is then increased by 25% after every execution. Furthermore, instead of requesting the full HTML file only the file's details were requested where the date in which the file was last modified can be extracted. This was done using a Web API called *XMLHttpRequest (XHR)* which can be used to do HTTP HEAD request. The HTTP HEAD method is a way of polling the server for just metainformation; such as the date a file was last modified. After extracting the data, if the file has updated then the division is reloaded with the new content and the polling is stopped.

A similar calculation to that shown above can be done to see the reduced data consumption for this method. In the same time frame of 5 minutes the exponential back-off function will run 25 times (Since $1s * 1.25^{26} > 300s$). This time the network monitor on a *Firefox* browser was only reporting 203 Bytes being transferred per request for this service, since only the metainformation was needed. If the HTML had updated an extra 4kB are transferred to update the web application (the size increases or decreases depending on the number of images present).

Assuming a change was detected on the last poll, the data consumption after the changes can be calculated as:

$$Consumption = DataTransferredPerPoll * NumberofPolls + SizeofHTML \tag{4.4}$$

$$Consumption = 203B * 25 + 4kB \tag{4.5}$$

$$Consumption \approx 9.075kB \tag{4.6}$$

The results for this showed significantly lower data consumption when reloading a division through it's regenerated `HTML` file. This resulted in a 96% reduction of transferred data using this method when comparing to the previous implementation. The project progressed using this implementation but it was later discovered that this was considered bad practice in web application networking; Chapter 4.11 addresses the issues with this implementation.

4.7 Video Recording

The *node-ar-drone* library, used for controlling the drone, provides access to a video stream. The drone's camera transmits an **h264** stream which could be programmed to be written to a file locally and it's contents can be shown on the web application.

4.7.1 Dependencies

The main dependency that was required for this feature was *FFmpeg*. This program can be used for handling media; some examples of its uses include video conversions, extracting frames, and wrapping videos in containers. A container is often referred to as a box which pieces together the audio, metadata, and frames in a video; common examples are `mp4` and `avi`.

FFmpeg was installed in the Raspberry Pi along with some of the dependencies it needed. These were all added to the installation script for ease of use in other Raspberry Pis that haven't been configured, if needed.

4.7.2 Implementation and Changes to DSL

Having installed the required dependencies a keyword "RECORDEDFLIGHT" with an associated string "video_name" were added to the DSL. To use this command the user must declare it before the main flight sequence with an identifier for the video recording. This keyword was then used in the generator to add code that would listen to the video stream and pipe it into a binary file.

Having added compatibility for storing the video in a binary file, a bash script was written which uses *FFmpeg* to wrap the video into an `mp4` container; this was required for compatibility with multiple devices and browsers. The bash script was programmed to be executed after the flight has finished. Moreover, another bash

script was also used to extract the first frame of a video, using *FFmpeg*, and storing it as a `PNG` image, this would allow for thumbnails to be generated for the videos.

4.7.3 Changes to Server

The server was configured with another directory listener for the videos. The implementation was similar to that explained in Section 4.6.2; where executors were used for concurrently monitoring a directory. The executors generate updated `HTML` code corresponding to the added or removed elements in the directory.

4.7.4 Changes to Web Application

Within the web application a new tab was added where the user can access the video recordings. In this tab the first element is an `mp4` video player pre-loaded with the latest video that was added to the directory. Below the video player, thumbnails for other videos are listed sequentially in order of date modified. When a thumbnail is clicked it's corresponding video is loaded to the main video player.

The web application also needed to adapt to a video directory that could change dynamically. In Section 4.6.3 an exponential back-off function, triggered on click of the "deploy" button, was used for dynamically updating the image gallery. The same function was initially used for updating the video playlist. The project progressed to other features using this implementation but, as stated before, continuously polling the server is considered bad practice. Towards the end of the project this was amended, refer to Section 4.11 for the issues with this implementation and the final solution.

4.8 Connection Feedback

After a debugging session, trying to understand why the drone would not deploy, it was seen that the drone had simply run out of battery. In order to avoid users having the same problem a feature was added where the web application shows if there is an active connection to the drone.

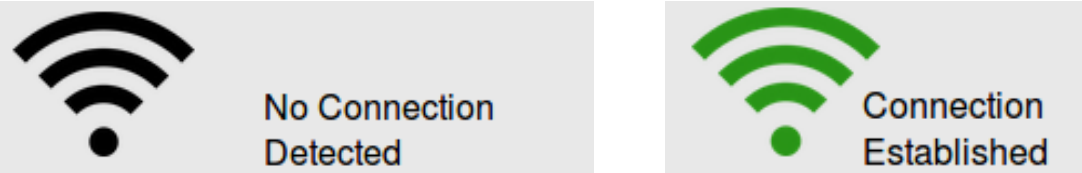
4.8.1 Changes to Server

An executor, similar to that used in Chapter 4.6.2, was added to start another thread which regularly pings the drone to check for connectivity. Recall, in Chapter 2.5, it was mentioned that the drone defaults to a static IP address of "192.168.1.1". This address was used together with the *Java* class *InetAddress*, which can be used to test if an address is reachable, in order to see if the connection with the drone is

alive. The server also generated the appropriate HTML code if it detects the active connection or if it doesn't find an active connection.

4.8.2 Changes to Web Application

The application was configured for polling the server for updates. XHR requests were also used for this feature to only retrieve metainformation and reduce data consumption, in this case the polling had to be regular. When the application receives metainformation for the HTML file and there is an updated date in which it was last modified, the application requests the full HTML file and reloads the division which indicates the connectivity. Figure 4.12 shows the visual differences, where a green icon indicates the drone is connected and can be deployed.



(a) Drone is not reachable. **(b)** Drone is connected to server.

Figure 4.12: The icon is highlighted either as green or black depending on the status of the drone's connectivity.

4.9 Landing Site Recognition

One of the user testers, the managing director, was intrigued by the use of *OpenCV* for image processing in drones. The user has taken part in arranging events where one of the challenges was autonomous flight navigation in timed conditions. The user was particularly keen on knowing if the image processing can be used to add variations to the drone challenges. The user asked about using the drone to recognise a specific site and land on it; more specifically he asked if the drone could be programmed to land on a table if the drone correctly identifies it.

To test the program's ability for landing site recognition an experiment was devised where three landing sites were used and the drone has to fly over them, and upon successful recognition, it should land on the correct one.

4.9.1 Changes to DSL

In order to carry out this experiment the DSL was added with another keyword "USE-DOWNWARDCAMERA" which can be used to tell the generated program to use the camera from below. The camera feed that's received when using the *node-ar-drone*

library defaults to using the front camera and the drone does not offer support for using both cameras at the same time. Hence, in the generator this keyword is used to override the camera configuration for using only the camera below. This addition has affected the snapshot and video recording feature as they are now dependent on which camera configuration the user chooses and it can't be changed mid-flight.

4.9.2 Landing Site Experiment

In the experiment, 3 distinctly different landing sites were printed to A3 pages, each of these landing sites had a width of 26cm. The sites were arranged in a straight line on the floor and they were separated by a distance of 2 meters, from the centre. The starting position was marked 2 meters away from the first landing site and wasn't changed throughout the experiment. An autonomous flight sequence was then written using the web application where the only commands used were "WAIT" and "FORWARD" to make the drone fly in a straight line, the program was tested first to ensure that unless interrupted by the landing site recognition it would fly past all the landing sites. Figure 4.13 shows a visualisation of the experiment.

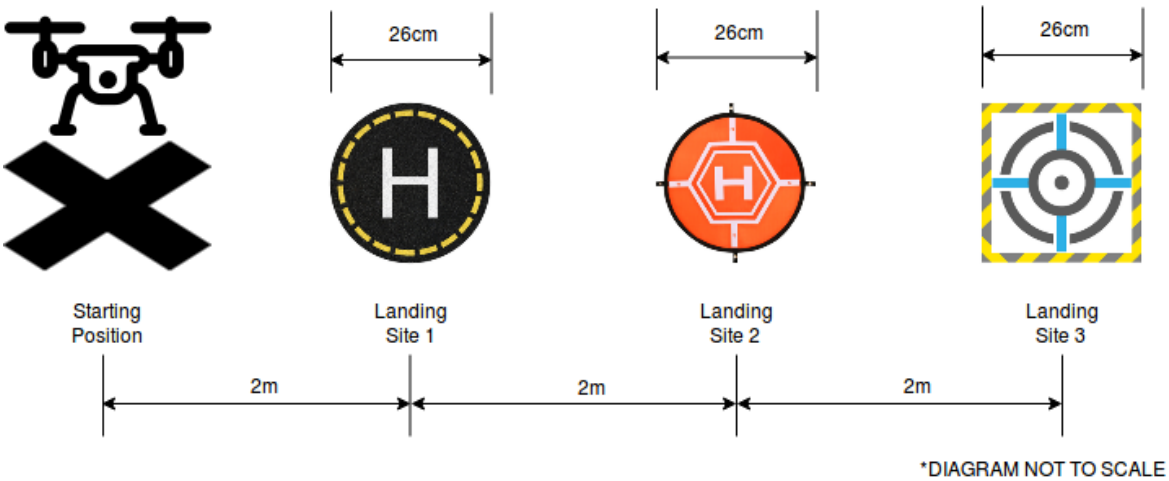


Figure 4.13: Experiment for testing landing site recognition.

Landing Site 1	Landing Site 2	Landing Site 3	Total
2/5	0/5	0/5	2/15

Table 4.2: Results for successful landing upon recognition of landing pads.

Table 4.2 shows the results of this experiment where 5 attempts were carried out for each landing site. The results shows a very poor success rate with the drone landing correctly on the first landing site 40% of the time. The remaining landing sites did not have any correct landings. Every flight for this experiment was recorded for post

experiment analysis, the videos revealed that a major problem with experiment was drone's stability [3].

Despite using the same flight sequence for all tests it can be seen on the recordings that the drone tends to deviate from the intended path, this particularly affected the second and third landing sites as in some cases the landing site wasn't even on the camera's view. Moreover, the bottom camera is low resolution (240p) with dull colours compared to the front camera. These are some of the limitations of using inexpensive drones; in this case optimisation of the feature matching algorithm wouldn't help produce better results.

With the successful landings the distance from the centre of the landing sites were recorded. In one case the drone landed 55cm in front of the first landing site and in another successful detection the drone landed 11cm in front of the first landing site. There were no observed occurrences of false positives where the drone would land on the wrong landing site.

4.10 User-Functions

A popular request amongst the user testers was the ability to have user defined functions. In particular, the coding instructors mentioned that this would make the program significantly better as a programming teaching tool for kids.

4.10.1 Changes to DSL and Generator

Within the DSL a parser rule called "SuperCommand" was added, this rule can either take an instance of "Commands" or a string "func_name". Another parser rule was also added called "UserFunction" which takes in a string identifier "name" and a set of "Commands". This allows for the user to define his personal functions at the end of the program with its associated commands, and call this function just by the string ID on the main sequence.

The generator was modified to adapt to this change. Initially the generator would step through the list of commands and generate the code corresponding to the command type. With the addition of aggregated commands, in the form of user defined functions, the generator had to search for the matching function and generate the code for the commands declared in the function each time the function is called.

[3] Videos of the experiment can be found in the following playlist: `https://www.youtube.com/playlist?list=PL3UsPW0LRPsDQY0i9sfMRxINKk00Wesb8`

4.10.2 IDE Validation

The coding instructors mentioned that the majority of times a program written by kids would fail was due to misspelling or missing tokens, such as semi-colons. Hence, the IDE had to be programmed to detect and highlight function calls for functions that have not been declared. Moreover, it also had to detect cases of duplicate function declarations and throw an informative error.

Xtext has a built in class, *NamesAreUniqueValidator*, that was used for detecting cases of duplicate function declarations. This class can be used for composed checks of the specific identifier "name"; the same identifier is used for the "UserFunction" parser rule. To use this class it must be referenced in the DSL's MWE2 (Modelling Workflow Engine 2) file; this file is responsible for generating components required for a DSL and also declaring the rules for how these components interact. In this case the *NamesAreUniqueValidator* was referenced in the component "validator". The result of this is duplicate functions are highlighted and an error is shown on the IDE next to these functions.

The validator also had to be programmed to cross reference function calls and function declarations. The details of different resources in a DSL can be accessed through an interface called *IResourceDescription*. This interface is responsible for exporting files "in order to make elements of one file referable from another file"[26]. The *Eclipse* documentation shows an example for extraction of the attribute "name"; this can be used to check against the attribute "func_name". The example was modified to extract the names of all user declared function and they were stored in an ArrayList. For every function declaration this ArrayList is queried to check if the function exists. Finally, a custom error was added which highlights calls to in-existent functions and shows the message "No matching function!" on the same line.

4.10.3 Main Program Syntax Graph

User defined functions were the final change to the DSL. To put the full DSL into perspective a syntax graph can be used; a syntax graph is a graphical representation for a language's grammar. Figure 4.14 shows the representation for the main program, this image shows which paths are compulsory (e.g "main_sequence"), which paths can be bypassed (e.g. "on_event"), and which paths can be repeated (e.g. "UserFunction"). In the graph the boxes in grey are either terminal rules or parser rules; where terminal rules take a "single atomic terminal token" and parser rules are a "tree of non-terminal and terminal tokens" [27]. The white boxes are tokens that are only needed due to semantics, they do not affect the code generation.

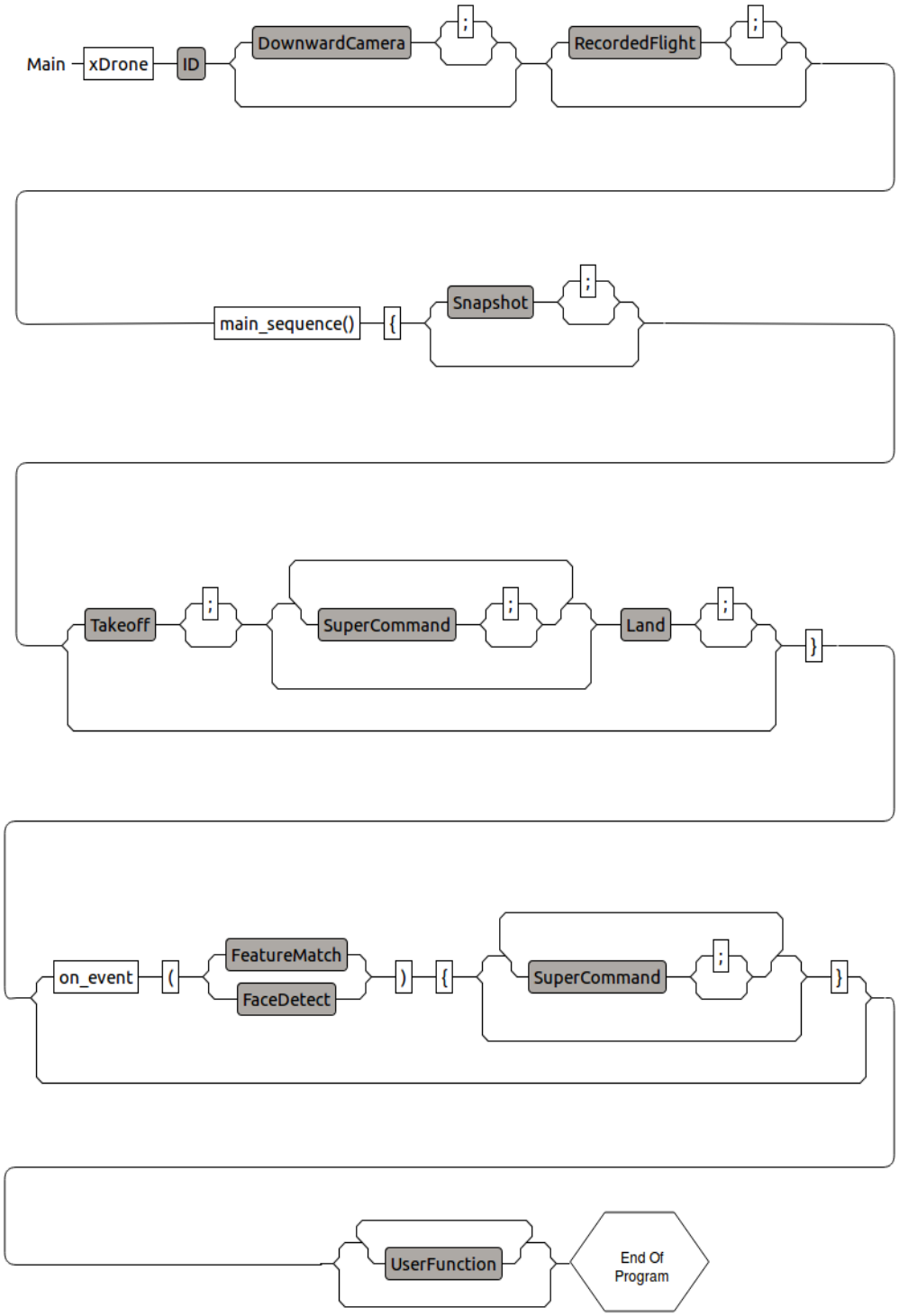


Figure 4.14: Syntax graph for the completed DSL.

4.10.4 Parser Rules Breakdown

The components of the parser rules can also be represented using their syntax graphs as shown in figures 4.15 and 4.16.

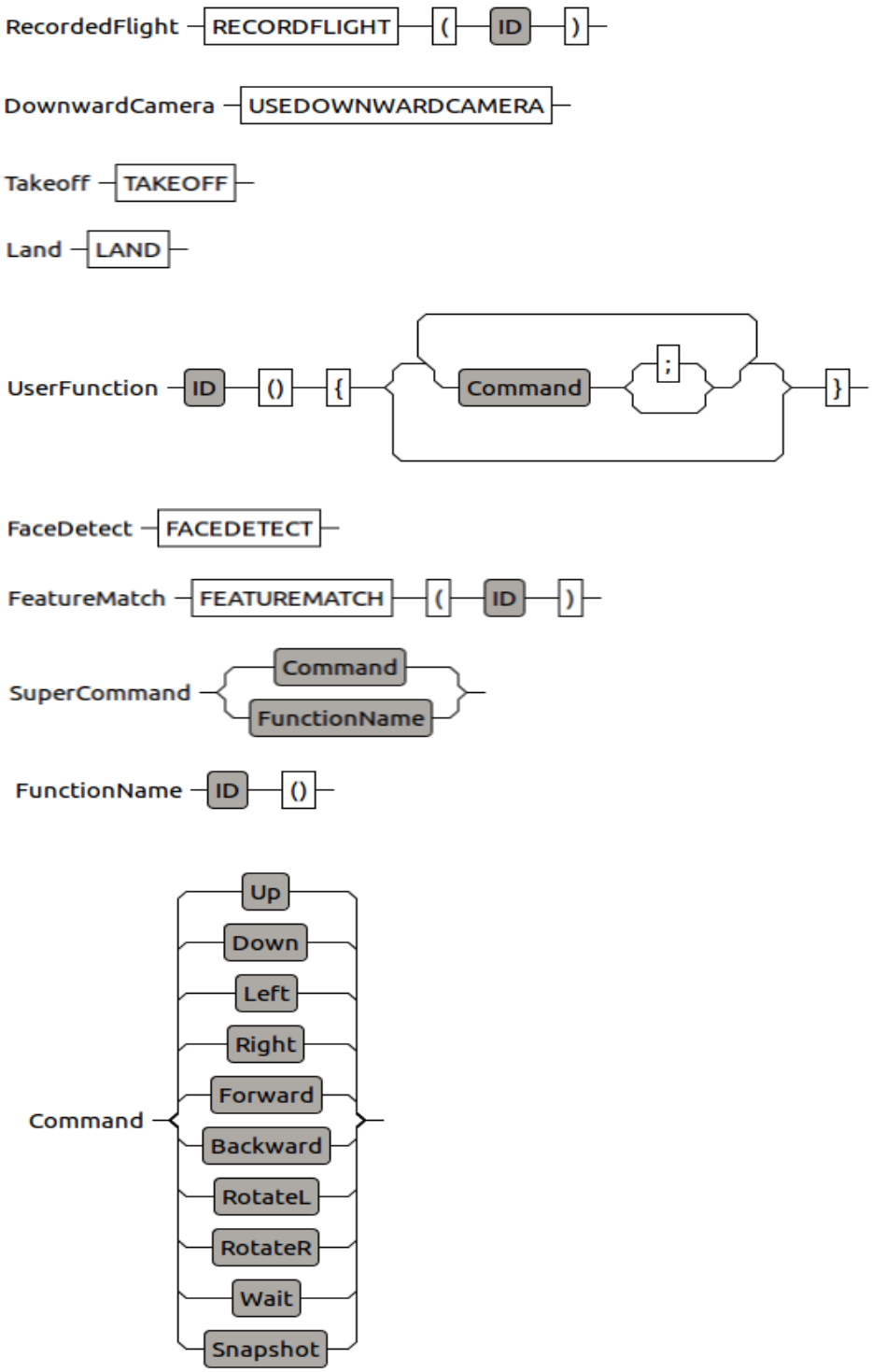


Figure 4.15: Parser rules syntax graph (1).

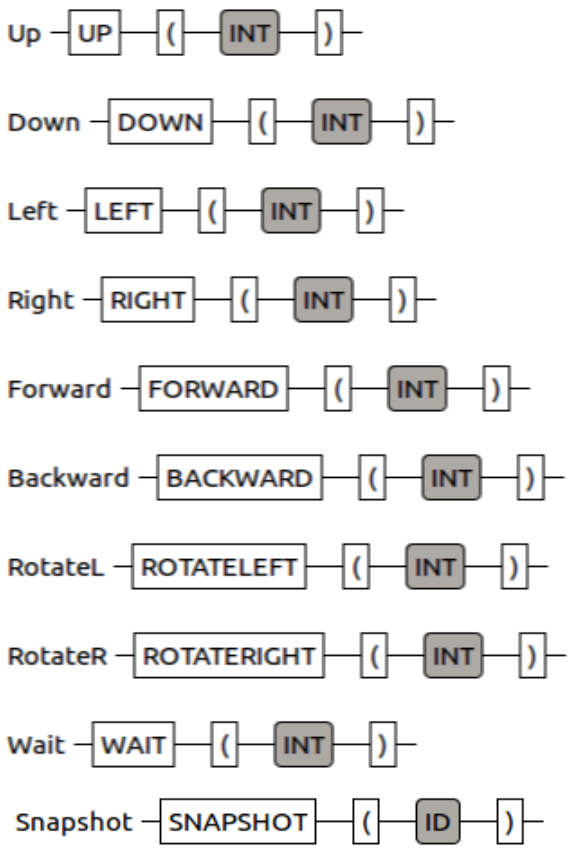


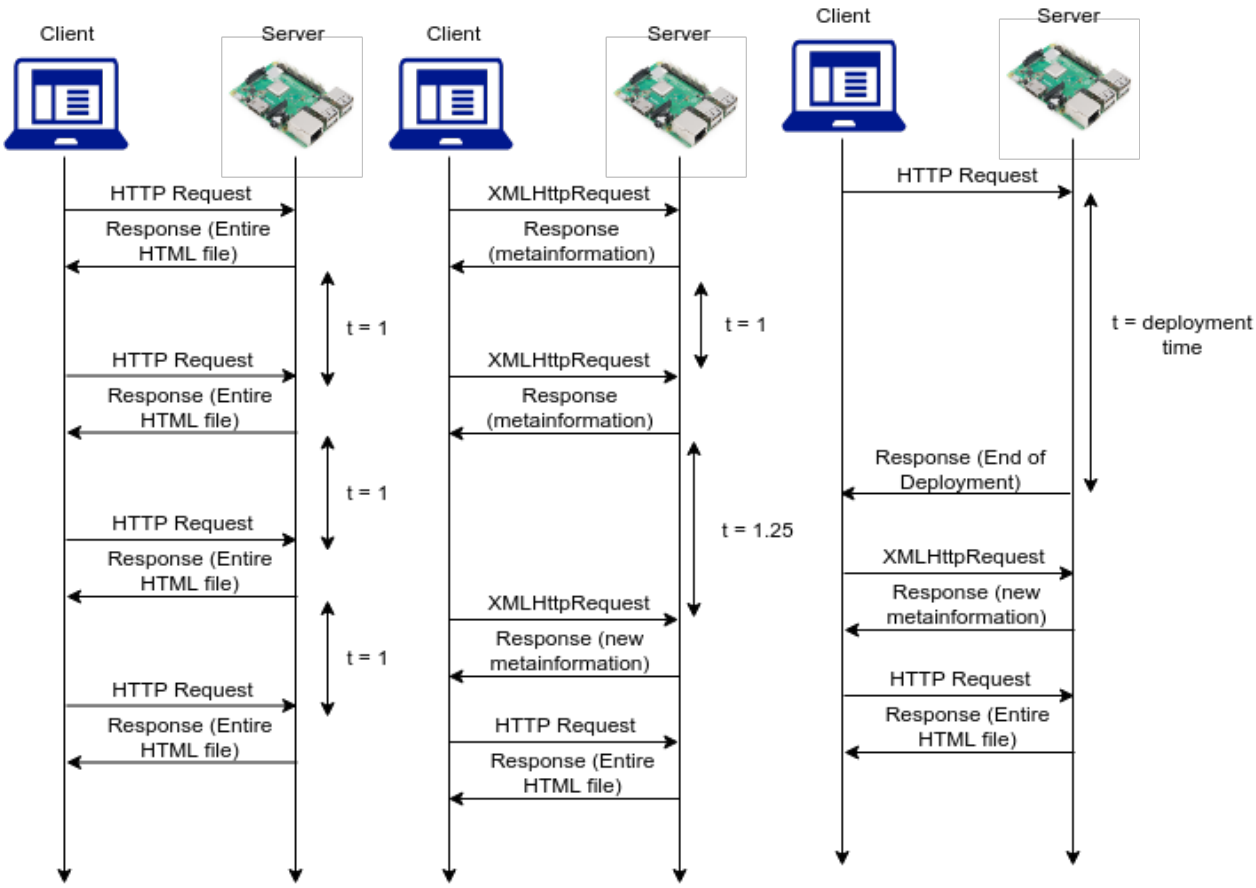
Figure 4.16: Parser rules syntax graph (2).

4.11 Request Blocking

It was realised late in the project that regularly polling a server for updates is a considered bad practice in networking. The initial implementation was not carefully thought out and had to be corrected through blocking a request. In this technique a client requests the server for new information but the server doesn't respond immediately; the server wait until it has new information then it sends the response.

Figure 4.17 shows the differences in the polling techniques tested in the project. The improvements between figure 4.17a and figure 4.17b are caused by the polling frequency decreasing over time; unless the "deploy" button is pressed where the polling interval is reset back to it's initial value. Figure 4.17c shows a blocked request; which is significantly better since a request is idle throughout the time the drone is deployed. After the deployment is finished the server can update the client with metainformation for the HTML files. The client device will then know if it needs to update the web application or not. This eliminates the need to continuously poll the server for updates and the web application will consume even less data.

To implement the request blocking, the XHR request (for checking HTML file changes) was programmed to execute only after the the deploy request; which runs the bash



(a) Regular polling

(b) Polling in Exponential Intervals using XHR

(c) Blocking deployment request then letting client request update

Figure 4.17: Polling techniques explored throughout the web application development.

script that deploys the drone. The bash script is run using the *Java* class "ProcessBuilder" which provides support for blocking a thread until the execution of the script is finished. This allows the web application to know when the deployment is finished; hence only one update request for the HTML file is needed as opposed to polling in intervals.

To summarise, Figure 4.17a was a regular polling implementation were all requests were for the full HTML file that describes the gallery or video player contents. In figure 4.17b all requests were *XMLHttpRequest* for the metainformation of the HTML file, and the polling here was executed in exponential intervals; except in the case of new metainformation where the full HTML file would load. In figure 4.17c the first request is for the the drone deployment service, this request is blocked throughout the deployment time, and the second request is an *XMLHttpRequest* for the metainformation of the HTML file; with the response of new metainformation the web application reloads the HTML file. Only the final implementation updates the gallery and video player on acknowledgement that the drone deployment was complete; this is the only time new images or videos could be added to the directory.

Note that solution was only implemented for the image gallery and video gallery. The connection feedback also makes use of regular polling to check if there is an active connection the drone. Due to time constraints a better solution for this feature was not implemented.

4.12 Final Architecture Revision

A user asked about being able to take the drone and fly it in a park; the architecture has limitations that do not permit this. One of these limitations is that the Raspberry Pi needs to be plugged in to a power socket; the solution to this is to simply use a portable power bank. A more serious problem is that the architecture needs the Raspberry Pi and the front-end device to be connected to the same network.

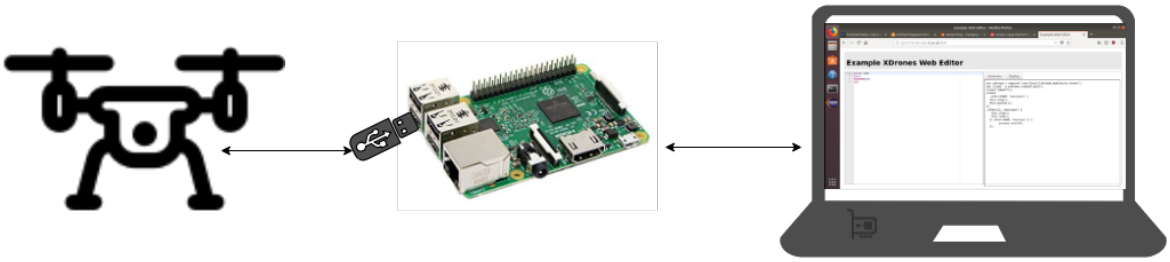


Figure 4.18: Revision of architecture using Raspberry Pi as an access point.

Figure 4.18 shows the ideal architecture which not dependent on an external network. In this architecture the Raspberry Pi connects to the drone's access point using the external NICs while the built-in wireless adapter is used to broadcast a WiFi connection to connect with the front-end device.

Chapter 5

Evaluation

This Chapter reflects on the implementation of the product and what could have been done differently to have a more robust and usable application. The usability of the application is also put to test with the help of the dedicated testers mentioned in Chapter 4, where the users were tasked with completing a set of tests using the application's user guide in Appendix C. Lastly, a comparison of the product with the existing products, referred in Chapter 2.4, is done to put the project's contributions into perspective.

5.1 User Testing in Person

The dedicated users testers, referred in Chapter 4, were all put to a test that lasts one hour. These tests reflect on the both the usability of the application and the quality of the user guide in Appendix C as the users were told that they had to complete each task in the test using just the user guide. The tasks are outlined below:

1. **Task 1 (beginner)**: The user must write a program that makes use of the applications flight recording feature. The program must also instruct the drone to takeoff and take a snapshot mid-air. Finally, the user must navigate the web application to view their snapshot and recorded video.
2. **Task 2 (intermediate)**: The user was told to use the autonomous flight commands and be able to take-off, then navigate through obstacles laid out in a room, then land at a specified position.
3. **Task 3 (advanced)**: The user was told to make use of the DSL's ability to do image processing to navigate the same path as that specified in the task above but instead of landing the drone must recognise an object, using feature matching, then navigate back to the starting position through a different route.

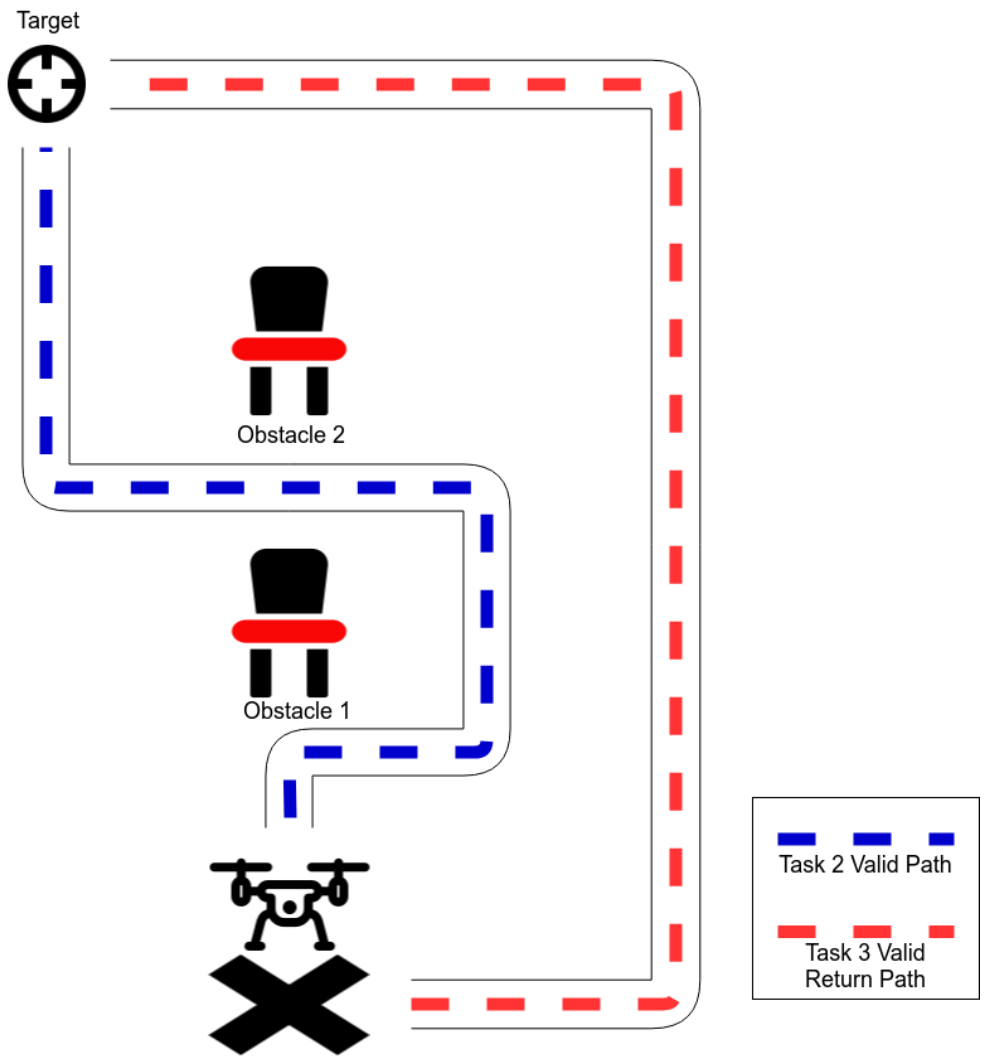


Figure 5.1: Example of user navigation test.

Figure 5.1 shows an example of the paths the users were required to navigate in order to pass task 2 and 3. Note that the tests were not identical for all users since the some users asked to do the testing in their own place of work. However, 3 users did the test at the Imperial Junion Common Room where the obstacles chosen where two chairs. For task 2, the users were told they had to fly the drone form the starting position then go through the gap between the chairs and finally land at the target location. In the case of task 3, they had to repeat the flight but in the target location an object was placed where the users had to detect it then return to the starting position through a different path.

The users were given as many tries as they liked for each task, provided they did it within the 1 hour slot. Before the test was started the drone was also calibrated externally, using *Parrot's Android* application.

Task 1	Task 2	Task 3	Total
5/5	4/5	0/3	9/13

Table 5.1: Results for successful task completion by the testers.

Table 5.1 shows the results for the tasks the users were asked to complete. For task 1 every single tester was able to complete it successfully within 2 tries. After multiple tries for task 2 all users were eventually able to complete it successfully, except for one user. In the case of the user that failed the test, he crashed the drone and the drone's frame was broken so the user could not continue the test and hence failed task 2 and didn't go as far as task 3. An interesting observation seen in task 2 was the way the users programmed the flight. Most users programmed it in increments where the user would write one command and deploy the drone, then the drone would be placed back in the starting position and another commands was added or adjusted, and the process was repeated until the drone reached the target position. However, one user wrote the whole program first then after deployment they would adjust the flight commands accordingly.

The final task yielded poor results, where not a single users was able to complete it successfully. Only 3 users took part in this task as 1 user couldn't complete the tasks due to breaking the drone's frame and 1 users finished task 2 just as their 1 hour time slot was coming to an end. In fairness to the testers, one of them was able to get the drone within 1 meter of the target object but the feature matching algorithm failed to detect it and as a result the commands the user declared to be executed on object recognition did not execute. The other testers were unable to get the drone close enough to the final target as the drone is not very stable; it was noticed that even rerunning the exact same code could have the drone deviate from it's path significantly and hence the drone wouldn't follow the exact path every time.

5.2 User Validation Using a Survey

After the testing the users were each asked to complete a survey with the aim of validating the product and it's features[1]. At this point all users had first hand experience with programming, using xDrone, and navigating the web application. The key responses from the survey are shown in Figures 5.2 - 5.5.

Figure 5.2 shows that on average the user testers rate the application a 4 out of 5 as teaching tool for programming. Individual responses for this question shows that one person gave a rating of 3; since the survey recorded names of testers it was seen that the person that gave this rating was a coding instructor. The survey also recorded written feedback for the layout of the programming language and this specific tester responded to the question "What would you add/change?" (in reference

[1]The complete survey results can be found in `https://docs.google.com/forms/d/1lXXYS6g34Ap_4FJqxmXOCyH3zPlZtYcYr_2UvSS-Euk/viewanalytics`

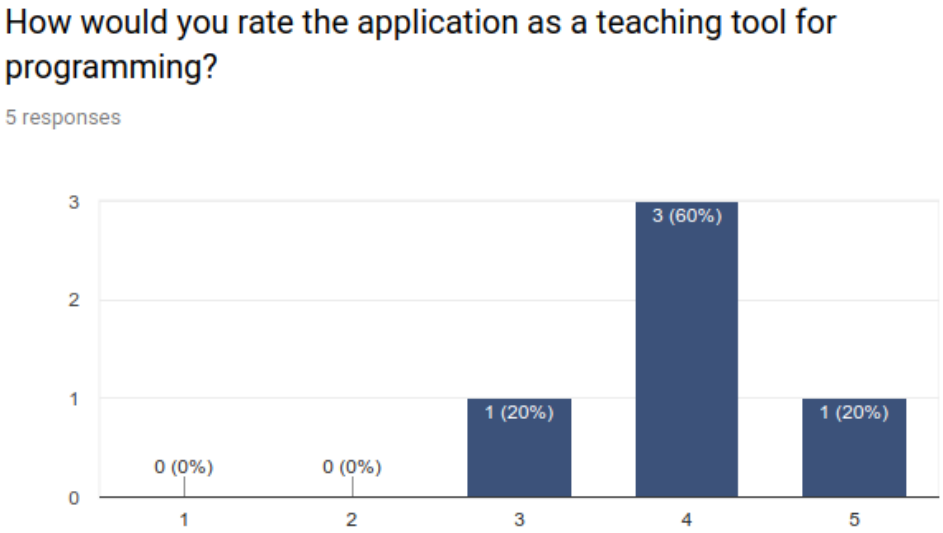


Figure 5.2: Validation for application as a teaching tool.

to the layout of the programming language). Her response was, “Nothing! I found the language simple without it being too minimal to make sense of. Great for young users.”. From her written response and rating, it can be seen that the tester finds the language adequate for younger students due to it’s simplicity but the minimal functionality means that the language is not very suitable for her more advanced students. Other users also commented on the lack of basic programming functionalities; more specifically they mentioned about lack of loops and conditionals which are vital for learning programming.

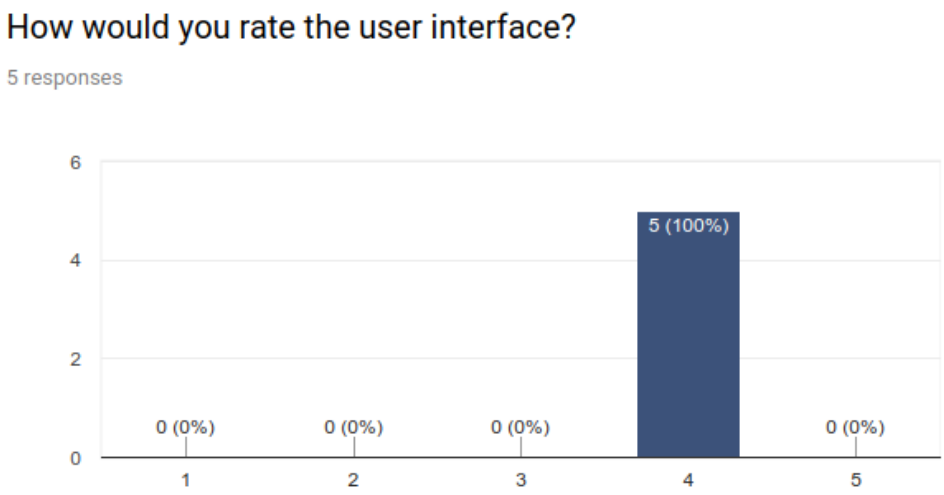


Figure 5.3: Validation for user interface.

The results regarding the user interface, shown in Figure 5.3, show that every single

user rated it a 4 out 5; this is a strong showing for the application as the user interface is a key factor for users when it comes to adopting a new application. The users were also asked to give constructive criticism on the implementation of the video player and the image gallery. The responders pointed out a flaw in the implementation where if a video or image is given the same name as a video or image that already exists, in their respective directories, then the file would be overwritten and the user would lose the previous file. Another user also said that the feature matching rule could be improved by implementing a way for the IDE to recognise an image through drag and drop, rather than declaring it by name in the DSL. This feature could prove very challenging as the IDE is generated automatically through *Xtext's* web integration tools and it's only a text editor.

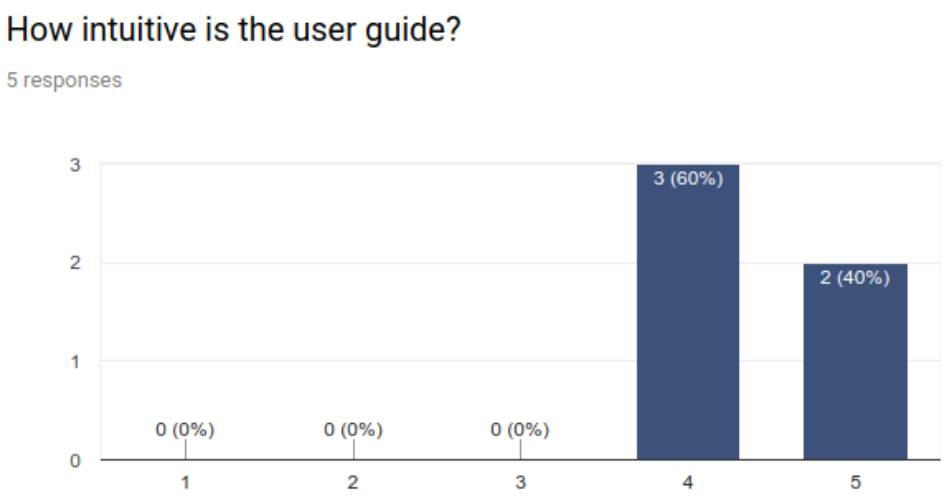


Figure 5.4: Validation for user interface.

Ratings were also collected for the user guide; every programming language provides documentation which introduces basic concepts for the language through examples. The user guide for *xDrone* was very well received with the user's giving it an average rating of 4.4 out of 5. It was also satisfying to see that the user tester without any programming background gave it a 5 out of 5 and he able to pass two of the tasks.

The users were also asked to give a rating for how innovative they thought the application was. Figure 5.5 shows the distribution for the answers of this question; the results for this were also very satisfying. The average rating was 4.6 out of 5; the rating says it all.

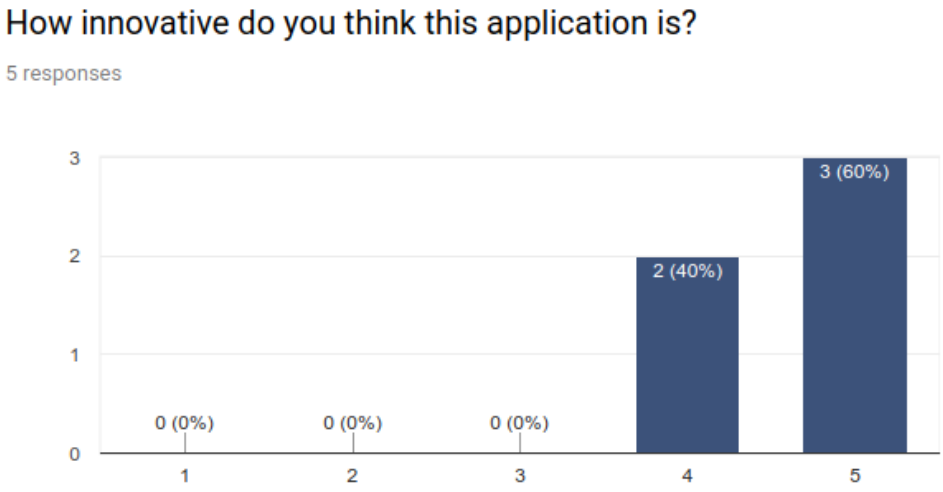


Figure 5.5: Validation for user interface.

5.3 Implementation of The Components

After the final validation from the users and the testing throughout the project, the implementation for each component must be revisited. This section identifies implementations that should have been done differently.

5.3.1 The DSL

The DSL was simple and intuitive, unfortunately this also meant that as a teaching tool it is limited in functionality. When speaking to the user testers, particularly the teaching instructors and the managing director for the technology camp, it was clear that the lack of loops and conditionals was an downside of the DSL. The managing director mentioned that it would be a very good visual demonstration for kids if you could program a drone to fly in a square by using a loop. At the moment multiple declarations of “FORWARD” and either “ROTATELEFT” or “ROTATERIGHT” are needed to accomplish this task; this is not an ideal way of programming.

The DSL also made use of *OpenCV* which can make the drone execute a different set of commands to those declared in the main sequence, if a face is detected or an object is recognised. The current implementation does not allow a program to go back to the main sequence to execute the remaining commands after it has finished executing the sequence under “FACEDETECT” or “FEATUREMATCH”. This is clear limitation and ideally the user should be given the option to have the drone execute the remaining code before landing. Another limitation is there is only one set of commands that can be executed in the case of object recognition or face detection. Testing for the face detection was done using a test program that instructs the drone

to take off and wait until a face is detected in the camera feed, this showed that there were problems false positive classifications, refer to Chapter 4.3.4. The solution used, where three consecutive frames where required for positive classification, proved to be robust and the program was less prone to executing code due to false positives. This solution also brought negatives. By making the detection criterion stricter, the program is also more prone to delays by having to process 3 frames instead of one or possibly more in the case of false negatives.

Two user testers also shared their inputs in the structure of the language. They had conflicting requests where one of the testers was happy with the structure and the way commands are declared between a pair of curly brackets. Another user had a conflicting request where he mentioned that making the language white-space aware could be very helpful for kids because it forces them to get into good programming habits (indentation). Input from more testers should have been gathered to reach a general consensus whether this could be helpful or not.

Lastly, the DSL provides support for aggregated commands that the user can declare at the end of the program. They were implemented in a similar way to user defined functions in languages such as *C++* or *Java*; where a user can declare a function and add commands to it then call it in the main program. However, the implementation was made in a restrictive way where user defined functions only take in flight commands; a better implementation would've allowed the DSL to take functions within functions. The reason this was done was because the DSL does not have any support for conditional statements so it wouldn't be possible to break from recursive functions. In general this feature was well received amongst the testers that are already aware of user defined functions for programming. Testing for this feature involved writing custom functions and declaring them in the main sequence then looking at the output for the generator. The feature works exactly as it was intended and the implementation of the generator together with the IDE's validator made the feature fail proof.

5.3.2 Image Gallery and Video Player

As mentioned in Chapter 5.2, the users found that when taking snapshots or recording flights the user could overwrite a previously saved image or video if named identically. This is a limitation of the implementation of these features, which were otherwise robust and well received by the users. The lightbox used for the image gallery as well as the video player, which loaded videos on the event that a corresponding thumbnail was pressed, did not fail at any point throughout the user testing or personal testing; this reflects on a good implementation.

5.3.3 Server Updates

The implementation of the features that need to be regularly updated by server was not ideal. It was realised late in the project that the initial way of continuously polling the server was considered bad practice in networking; a better solution was made to the image gallery and video gallery but it still has drawbacks and should be fixed. One of the drawbacks is *HTTP* timeout, where the client's browser can close a connection if it taking too long. The timeout on the latest version of the *Firefox* browser and the latest version of the *Google Chrome* browser is 300 seconds [2]. This means that without tweaking the *HTTP* timeout manually, the application will not fetch updates if the deployment service takes longer than 300 seconds. The user will have to manually refresh the web application to get the latest media content captured in longer flights. It should also be noted that the battery life for the *Parrot AR Drone 2.0* is claimed to be 12 minutes in their official website, so flights that last over 300 seconds are possible.

The correct implementation for pushing notifications from the server to the web application would be to use the *WebSocket API*. This allows for bidirectional transfer of data between a server and client and hence can be used to send server updates with low latencies to the client. It also does not have the problem of timeouts since *WebSockets* make use of "ping-pong messages", which is a type of message transmitted regularly to check health of the connection [28]. Moreover, *WebSockets* establish a connection between client and server once, after that the connection is kept alive. HTTP requests, on the other hand, need to initiate connection from client each time so it has a bigger overhead, in relation to data transfer.

5.3.4 Displaying Generated Code

The generated code can only be used as a debugging tool at the moment. The user can see the generated code in *NodeJS* but that's the only use for it. The display of the generated code can be seen with syntax highlighting for *JavaScript* as well as line numbers, this was a good addition as it makes it easier to interpret the code. Overall, it a simple feature that works as expected.

5.3.5 Computer Vision

The project explored feature matching as a way to recognise objects through the camera without having to make the user train their custom classifiers. While this feature felt like a very good addition to the DSL it's usability proved to be limited, as seen in the testing for landing site recognition in Chapter 4.9. The intentions were to have the user simply pass a reference image to the program and in real time the drone would be able to recognise it. In the case of landing sites the drone

[2] Some browsers, particularly older ones, can timeout in even shorter times.

was not stable enough and would deviate from it's intended path, making it unable to recognise the landing sites. The experimental results (2/15 or $\sim 13.3\%$) show that the implementation of this feature is practically unusable in real life scenarios. Furthermore, the recorded videos using the bottom facing camera showed the lack of quality for the images which could have also affected the testing.

5.3.6 The Architecture

The use of an inexpensive drone, the *Parrot AR Drone 2.0*, made this project ideal for teaching institutions or hobbyists. However, one aspect of the architecture that was not considered is that the user needs to also have a Raspberry Pi 3B+ and and extra NIC (USB WiFi dongle) to be able to use the application. This increases the cost of the full system and hence could be regarded as a negative since the cost could discourage adoption.

5.4 Existing Software Comparison

Two applications, *Tynker* and *Scratch*, were discussed in Chapter 2.4 due to their use in making drone programming easier. Non-programmers can abstract away from programming drones using general purpose languages by using *Tynker* and *Scratch* to create programs by dragging and dropping commands to form a program. *xDrone* differentiates from these applications because it makes drone programming easier but by using textual programming instead of graphical programming. Hence, within the context of existing application, *xDrone* slots somewhere in between these applications and general purpose languages.

The difference in programming style between *xDrone* and the other applications brings both advantages and disadvantages. One of the major advantages of the other applications is the drag and drop feature; since kids tend to have short attention spans programs that are more visual can help captivate their attention. On the other hand, *xDrone* can be a better transition to the more traditional form of programming as it relies on textual programming and also introduces some key programming concepts, like user defined functions.

The compatibility with front end devices is also different for these applications. *xDrone* is by far the hardest to configure as there is only one installation script for dependencies that was tested using a Raspberry Pi 3B+; this is an integral part of the architecture. Moreover, the installation script only deals with library dependencies; changes to the network configuration files were made manually. While configuring *xDrone* may be a disadvantage, the application will then work on multiple front-end devices; tested environments include *Windows 10*, *Ubuntu 16.04*, and *Android 7* with *Firefox* and *Google Chrome* browsers. *Scratch* is limited to devices running on *Windows* or *macOS* operating systems while *Tynker* is limited to *Android* and *iOS*. Furthermore, a *Parrot Minidrone Spyder* was aquired as it is compatible with *Tynker*

and first hand experience with the application would have been beneficial. However, when attempting to download the application, on a *Samsung S7* running on *Android 7*, the application store reported, "this application is not compatible with your device".

xDrone has also pushed the user interface for these drone programming applications by providing better support for media content. In the midst of the social media era it is quite surprising that *Tynker* only has basic support for taking photos and the *Ryze Tello* extensions for *Scratch* do not appear to have support for media content at all. *xDrone* is unparalleled in support for media content with simple and intuitive ways to take multiple photos throughout an autonomous flight as well as being able to make fully recorded flights. *xDrone* also presents these images and videos within the application in a user-friendly fashion.

xDrone's also differentiates itself from existing drone programming application by providing support for computer vision in a simple and intuitive way. While the testing for computer vision showed poor results it is still a step in the right direction. At the time of writing, applications for drone programming with support for computer vision have not been identified.

Chapter 6

Conclusion and Future Work

In this chapter the contributions and achievements of the project are summarised. The chapter also outlines future developments that could be made, provided more time is available.

6.1 Contributions

In this project a prototype DSL and web application were developed; with the aim of making drone programming easier by focussing on a specific domain. The prototype, *xDrone*, was developed with support for basic autonomous flights. In addition, it has built-in functionality for real time face detection and feature matching using the drone's camera feed. From which, the user can also define conditional commands that are executed on successful detection or recognition. This was a standout feature making *xDrone* a bit more unique compared to existing software aimed at making drone programming easier; the existing applications that were used for comparison do not have support for image processing.

The DSL was developed to be user friendly to also target users that do not have a programming background or are just starting to learn programming. Dedicated testers were used in this project where they were asked to complete a set of challenges using only a user guide written for the language. The results were promising with the majority of testers being able to operate the drone successfully for two (out of three) of the devised challenges; one of these testers did not have a programming background prior to testing *xDrone*. However, the project's intent to provide conditional commands in the case of face detection or object recognition proved unsuccessful; every tester that attempted a challenge which required the computer vision implementation was unable to complete the challenge successfully and this a clear area that need improvement for the project.

The web application was well received with the user testers. The application provides a user interface that gives access to an IDE which is used for writing a program.

Validation rules were added to the IDE to show informative messages in case of syntax or grammar errors. Moreover, the web application also has a section which allows users to browse through their gallery images or play recorded flight videos. This was also well received with some users saying that this is a good tool for people who frequently use social media and would like to post some videos or images captured by the drone. It should also be noted that the applications used for comparison do not have the ability to let the user capture video feed programatically while *xDrone* has this support.

6.2 Future Work

Some users requested changes to be made in application and architecture, but not all requests were fulfilled. These request will be outlined in this section along with potential extensions for the project.

6.2.1 The DSL

Future development of the project will need a parser rule which takes in an integer and can make the drone repeat commands in a loop. Conditionals such as if-statements should also be considered as they would further improve the application for teaching environments.

A user tester thought it would be very interesting to have the drone recognise a target then navigate automatically towards it, then look for another target and again navigate automatically towards it; an approach for this problem would be to add laser sensors for depth perception and on detection move towards the object until the drone is close enough to it. This would be a challenging but impressive feature to add to the DSL.

6.2.2 Use of Generated Code

A user asked to explore the possibility of having two IDE's; with the DSL on the left and the *NodeJS* on the right. The user justified this request by saying more advanced users should have the option to generate and view the *NodeJS* code but also make their own personal additions to the code without being restricted to what the DSL supports. This is another challenging addition that could be considered for future work; in this case the generated code would have to be loaded to an embedded text editor and another deployment service linked to this editor would be needed.

6.2.3 Image Gallery and Video Player

There is also no support for deleting an image or video directly through the web application. At the moment the user currently has to delete it directly in the Raspberry Pi; a more friendly user interface should just provide the ability for the user to delete the files they do not require. Furthermore, the way images are named could be made more user-friendly by using the same naming convention that is used in mobile phones; where a camera can take an image and give the image a name which corresponds to the date and time. This would solve the issue with images and videos being overwritten if the same name is already in use for an existing file.

6.2.4 Security Implications

The drone's access point has the default configuration where the connection is open and any user can access it directly. In paper by Johann-Sebastian Pleban et al. the security vulnerabilities of the *Parrot AR Drone 2.0* are explained [29]; where potential hackers can get access to the drone's videostream or even hijack the drone. They also discuss ways of making the connection more secure by encrypting the drone's WiFi network using a configured wpa_supplicant file. This extension is possibly the most important and should not be overlooked if the project is to progress further.

6.3 Conclusion

Overall, the project fulfilled it's aim of developing a web application and DSL for drone programming. The DSL was intuitive and user testing proved that it has potential to be used as an entry-level tool for programming. Furthermore, the project make use of DSLs for writing programs that allow the drone to interact with the real world

Bibliography

[1] M. Fowler, *Domain-specific languages*. Pearson Education, 2010. pages 3

[2] L. Bettini, *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd, 2016. pages 3, 4, 5

[3] Techopedia, "What is an Abstract Syntax Tree (AST)?." https://www.techopedia.com/definition/22431/abstract-syntax-tree-ast. Accessed: 2018-09-06. pages 4

[4] G. Tomassetti, "A guide to parsing: Algorithms and terminology." https://tomassetti.me/guide-parsing-algorithms-terminology/, 09 2017. pages 4

[5] Eclipse, "Xtext documentation." http://www.eclipse.org/Xtext/documentation/2.7.0/Xtext%20Documentation.pdf, 09 2014. pages 5

[6] Tynker, "Code This Drone — Parrot Drone Programming — Tynker." https://www.tynker.com/learn-to-code/code-this-drone/. Accessed: 2018-09-03. pages 8

[7] Ryze, "Ryze Tech." https://www.ryzerobotics.com/news/launch_news_page_en. Accessed: 2018-09-03. pages 9

[8] E. Brown, "Linux-based quadrocopter gains flight recorder." http://linuxgizmos.com/ar-drone-2-quadrocopter-gains-cool-features/, 2013. pages 9

[9] S. Piskorski, N. Brulez, P. Eline, and F. DHaeyer, "Ar.drone developer guide." https://jpchanson.github.io/ARdrone/ParrotDevGuide.pdf, 05 2012. pages 10

[10] S. Soo, "Object detection using haar-cascade classifier," *Institute of Computer Science, University of Tartu*, 2014. pages 11

[11] P. Viola and M. Jones, "Rapid object detection using a boosted cascade of simple features," in *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on*, vol. 1, pp. I–I, IEEE, 2001. pages 11

[12] Q. Chen, N. D. Georganas, E. M. Petriu, *et al.*, "Real-time vision-based hand

gesture recognition using haar-like features," in *Instrumentation and Measurement Technology Conference Proceedings*, pp. 1–6, Citeseer, 2007. pages 11

[13] M. Vidanapathirana, "Real-time human detection in computer vision." https://medium.com/@madhawavidanapathirana/https-medium-com-madhawavidanapathirana-real-time-human-detection-in-computer-vision-part-1-2acb851f4e55. Accessed: 2018-08-26. pages 11

[14] OpenCV, "Template Matching – OpenCV 2.4.13.7 documentation." https://docs.opencv.org/2.4/doc/tutorials/imgproc/histograms/template_matching/template_matching.html. Accessed: 2018-08-26. pages 12

[15] OpenCV, "OpenCV: Feature Matching." https://docs.opencv.org/3.3.0/dc/dc3/tutorial_py_matcher.html. Accessed: 2018-08-26. pages 13

[16] T. Marciniak, A. Chmielewska, R. Weychan, M. Parzych, and A. Dabrowski, "Influence of low resolution of images on reliability of face detection and recognition," *Multimedia Tools and Applications*, vol. 74, no. 12, pp. 4329–4349, 2015. pages 14, 15

[17] Engadget, "This selfie-drone recognizes, finds and auto-records its owner." https://www.engadget.com/2017/01/03/zero-zero-hover-camera-update/. Accessed: 2018-09-04. pages 15

[18] N. Dijkshoorn, H. Flynn, O. Formsma, S. van Noort, C. van Weelden, C. Bastiaan, N. Out, O. Zwennes, S. S. Otárola, J. de Hoog, *et al.*, "Amsterdam oxford joint rescue forces-team description paper-virtual robot competition-rescue simulation league-robocup 2011," pages 15

[19] M. Anastasov, "What's the difference between continuous integration, continuous deployment and continuous delivery?." https://semaphoreci.com/blog/2017/07/27/what-is-the-difference-between-continuous-integration-continuous-deployment-and-continuous-delivery.html, 2017. pages 20

[20] Eclipse, "Eclipse jetty documentation - embedding jetty." http://www.eclipse.org/jetty/documentation/9.3.x/embedding-jetty.html. Accessed: 2018-09-01. pages 23

[21] Gradle, "Gradle docs - the gradle wrapper." https://docs.gradle.org/current/userguide/gradle_wrapper.html. Accessed: 2018-09-01. pages 23

[22] "Scheduling: settimeout and setinterval." https://javascript.info/settimeout-setinterval. Accessed: 2018-09-01. pages 25

[23] T. Butcaru, "node-opencv." https://github.com/tbutcaru/node-opencv. pages 30

[24] Google, "code-prettify." https://github.com/google/code-prettify. pages 31

[25] "w3schools - How To Create a Lightbox." https://www.w3schools.com/howto/howto_js_lightbox.asp. Accessed: 2018-09-02. pages 32

[26] Eclipse, "Xtext - Language Implementation." https://www.eclipse.org/Xtext/documentation/303_runtime_concepts.html. Accessed: 2018-09-03. pages 39

[27] Eclipse, "Xtext - The Grammar Language." https://www.eclipse.org/Xtext/documentation/301_grammarlanguage.html. Accessed: 2018-09-04. pages 39

[28] baeldung, "A Guide to the Java API for WebSocket." https://www.baeldung.com/java-websockets. Accessed: 2018-09-04. pages 52

[29] J.-S. Pleban, R. Band, and R. Creutzburg, "Hacking and securing the ar. drone 2.0 quadcopter: investigations for improving the security of a toy," in *Mobile Devices and Multimedia: Enabling Technologies, Algorithms, and Applications 2014*, vol. 9030, p. 90300L, International Society for Optics and Photonics, 2014. pages 57

Appendix A

Ethics Checklist

	Yes	No
Section 1: HUMAN EMBRYOS/FOETUSES		
Does your project involve Human Embryonic Stem Cells?		✓
Does your project involve the use of human embryos?		✓
Does your project involve the use of human foetal tissues / cells?		✓
Section 2: HUMANS		
Does your project involve human participants?	✓	
Section 3: HUMAN CELLS / TISSUES		
Does your project involve human cells or tissues? (Other than from Human Embryos/Foetuses i.e. Section 1)?		✓
Section 4: PROTECTION OF PERSONAL DATA		
Does your project involve personal data collection and/or processing?	✓	
Does it involve the collection and/or processing of sensitive personal data (e.g. health, sexual lifestyle, ethnicity, political opinion, religious or philosophical conviction)?		✓
Does it involve processing of genetic information?		✓
Does it involve tracking or observation of participants? It should be noted that this issue is not limited to surveillance or localization data. It also applies to Wan data such as IP address, MACs, cookies etc.		✓
Does your project involve further processing of previously collected personal data (secondary use)? For example Does your project involve merging existing data sets?		✓
Section 5: ANIMALS		
Does your project involve animals?		✓
Section 6: DEVELOPING COUNTRIES		

Does your project involve developing countries?		✓
If your project involves low and/or lower-middle income countries, are any benefit-sharing actions planned?		✓
Could the situation in the country put the individuals taking part in the project at risk?		✓
Section 7: ENVIRONMENTAL PROTECTION AND SAFETY		
Does your project involve the use of elements that may cause harm to the environment, animals or plants?		✓
Does your project deal with endangered fauna and/or flora /protected areas?		✓
Does your project involve the use of elements that may cause harm to humans, including project staff?		✓
Does your project involve other harmful materials or equipment, e.g. high-powered laser systems?		✓
Section 8: DUAL USE		
Does your project have the potential for military applications?	✓	
Does your project have an exclusive civilian application focus?		✓
Will your project use or produce goods or information that will require export licenses in accordance with legislation on dual use items?		✓
Does your project affect current standards in military ethics e.g., global ban on weapons of mass destruction, issues of proportionality, discrimination of combatants and accountability in drone and autonomous robotics developments, incendiary or laser weapons?	✓	
Section 9: MISUSE		
Does your project have the potential for malevolent/criminal/terrorist abuse?	✓	
Does your project involve information on/or the use of biological-, chemical-, nuclear/radiological-security sensitive materials and explosives, and means of their delivery?	✓	
Does your project involve the development of technologies or the creation of information that could have severe negative impacts on human rights standards (e.g. privacy, stigmatization, discrimination), if misapplied?	✓	
Does your project have the potential for terrorist or criminal abuse e.g. infrastructural vulnerability studies, cybersecurity related project?	✓	
Section 10: LEGAL ISSUES		
Will your project use or produce software for which there are copyright licensing implications?		✓

Will your project use or produce goods or information for which there are data protection, or other legal implications?		✓
Section 11: OTHER ETHICS ISSUES		
Are there any other ethics issues that should be taken into consideration?	✓	

Table A.1: Checklist of the project's potential ethical implications.

Appendix B

Ethical and Professional Considerations

The project did involve human participants for testing and their names were collected in a survey. However, only adults were chosen and the survey also gave the option for the testers to submit their responses anonymously.

The project could also be used malevolently since it facilitates the programming for autonomous drones. In this case users can deploy the drone and make use of the support for recorded flight or taking snapshots for spying. However, the drone has a maximum range of 50 meters and this significantly restricts the potential of using it as a tool for spying. Moreover, the *Parrot AR Drone 2.0* weighs more than 250g which makes it illegal for flying in the UK without registering the device with the Civil Aviation Authority.

Autonomous drones could also be used for transporting dangerous content. In the case of the *Parrot AR Drone 2.0* there is no official documentation stating the drone's lift capacity but users on drone forums have reported maximum lift capacities in the range between 100-300 grams. This could be a problem if users choose to add sharp objects or other dangerous substances.

The project also has potential to be used as a teaching tool. It is a physical safety concern to have unsupervised children operate the drones.

Under no circumstances was the project's development driven by malevolent intent. The use of the application and the drone must comply with the Data Protection Act, the CCTV Code of Practice, and the Air Navigation Order (ANO). In particular, under the latest revision of the ANO (2016) in Article 241 it clearly states "A person must not recklessly or negligently cause or permit an aircraft to endanger any person or property.".

Appendix C

User Guide

C.1 Commands

The following commands are used specifically for autonomous navigation.

- UP (milliseconds)
- DOWN (milliseconds)
- LEFT (milliseconds)
- RIGHT (milliseconds)
- FORWARD (milliseconds)
- BACKWARD (milliseconds)
- ROTATELEFT (milliseconds)
- ROTATERIGHT (milliseconds)
- WAIT (milliseconds)

The commands are self explanatory and are used to define the flight sequence. They all take an input "millisecond" which is a timer executed after the command. For example, calling UP(1000) then LEFT(2000) will make the drone go UP for a second, then the drone will move towards the left for 2 seconds.

There is another command which doesn't affect the movement of the drone. The commands is:

- SNAPSHOT (image_name)

This command is used for accessing the drone's camera and taking a picture. The field "image_name" is a string that allows the user to give an identifier to the image.

C.2 Configurations

There are also configurations that can be used before deploying the autonomous flight sequences. The configurations are:

- USEDOWNWARDCAMERA
- RECORDFLIGHT(video_name)

Both of these are optional and do not have to be used. However, if they are used they must be specified at the beginning of the program. If they are to be used together then the command "USEDOWNWARDCAMERA" must be declared first, a syntax error will show otherwise. If neither of these configurations is used then the program will default to not recording the flight sequence and any snapshots taken throughout the flight will use the front facing camera.

C.3 Structure of the Program

The commands and configurations shown on the previous section need to be used within the DSLs rules. This section will show you examples programs so that you can write your own program.

Let's begin with the simplest case:

```
xDrone Program_Name

main_sequence(){

}
```

This program has the first field where the program name is declared and it has the main sequence field which is empty. Any program using this DSL must have these fields declared. As for the outcome of the program, since there are no commands declared nothing happens.

Let's move on to the program's configurations.

```
xDrone Program_Name

USEDOWNWARDCAMERA

main_sequence(){

}
```

(a) Camera Configuration

```
xDrone Program_Name

USEDOWNWARDCAMERA
RECORDFLIGHT(myawesomevideo)

main_sequence(){

}
```

(b) Camera Configuration with Recorded Flight

The programs defaults to using the front camera but in the figure above it shows how the program can use the downward facing camera instead. Furthermore, the recorded flight configuration can also be used, but if the user wants to use them together (i.e. Record using the downward facing camera) then the "RECORDFLIGHT" keyword must be declared after the "USEDOWNWARDCAMERA" keyword.

Just one more example before we start flying the drone:

```
xDrone Program_Name

main_sequence(){
    SNAPSHOT(my_snapshot)
}
```

In this example the "SNAPSHOT" commands is used on it's own in the main sequence. This allows the user to have close control over the drone so they can take a picture whilst the drone isn't flying.

Now that you understand the configurations and the stationary snapshot feature we can write a simple program to have the drone takeoff and land.

```
xDrone Program_Name

main_sequence(){
    TAKEOFF

    LAND
}
```

In the program above the drone will simply takeoff then land. The region between the "TAKEOFF" and "LAND" is where you can use the commands for autonomous flights. As a safety measure the "TAKEOFF" and "LAND" must be coupled.

Now let's move to an example where a flight sequence is added.

```
xDrone Program_Name

main_sequence(){
    TAKEOFF
    DOWN(2000)
    SNAPSHOT(my_snapshot2)
    RIGHT(3000)
    LAND
}
```

In this example the drone takes off, moves down for 2 seconds, it then takes a snapshot, it then moves right for 3 seconds, and finally it lands. There isn't a limit to the number of commands to be added in this block and you can use any of the commands shown in section C.1.

The drone also has support for object detection and recognition.

```
xDrone Program_Name

main_sequence(){
    TAKEOFF
    DOWN(2000)
    SNAPSHOT(my_snapshot2)
    RIGHT(3000)
    LAND
}

on_event(FACEDETECT)
{

}
```

(a) Face Detection

```
xDrone Program_Name

main_sequence(){
    TAKEOFF
    DOWN(2000)
    SNAPSHOT(my_snapshot2)
    RIGHT(3000)
    LAND
}

on_event(FEATUREMATCH(myreferenceimage))
{

}
```

(b) Reference Image Detection

In the examples above the global conditional field is used to break the sequence of the flight. The "FACEDETECT" keyword does not require any input while the "FEATUREMATCH" needs a reference image to compare with the drone's camera feed. In the example "myreferenceimage" should be found in the gallery as "myreferenceimage.png". In these example the autonomous flight sequence will progress normally but if the global conditional is triggered then drone will stop execution and will land.

Now let's make better use of the global conditionals.

```
xDrone Program_Name

main_sequence(){
    TAKEOFF
    DOWN(2000)
    SNAPSHOT(my_snapshot2)
    RIGHT(3000)
    LAND
}

on_event(FACEDETECT)
{
    BACKWARD(1000)
    SNAPSHOT(mynewprofilepicture)
}
```

In the example above, the same program is executed as before. However, there is a small change. The global conditional field now has commands in it. So the flight sequence will execute as normal but on face detection the drone will discard the main flight sequence and will instead move backwards for a second then it will take a snapshot called “mynewprofilepicture”.

Lastly, the user may want to define his own functions. For repetitive tasks it easier to call a function multiple times instead of writing out repeated commands.

```
xDrone Program_Name

main_sequence(){
    TAKEOFF
    my_function()
    my_function()
    LAND
}

on_event(FACEDETECT)
{
    my_function()
}

my_function()
{
    ROTATELEFT(1000)
    FORWARD(1000)
    ROTATELEFT(1000)
    FORWARD(1000)
    ROTATELEFT(1000)
    FORWARD(1000)
    ROTATELEFT(1000)
    FORWARD(1000)
}
```

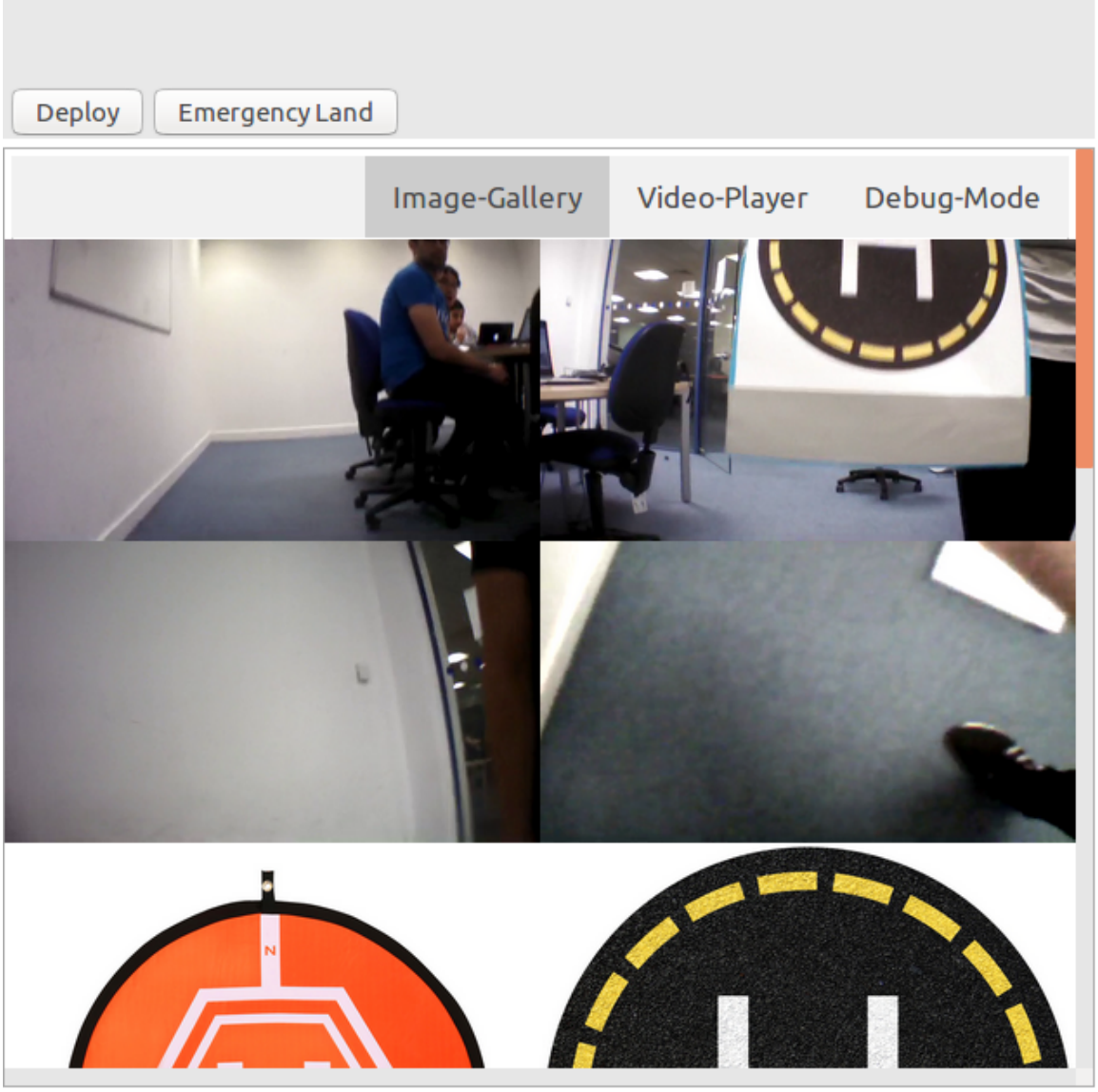
The example above shows a newly declared function called “my_function()”. When declaring these functions the name must end with a pair of parenthesis and a set of curly brackets is also required where the commands can be declared. The functions will only take simple commands for the drone, they cannot call itself in a recursive manner. Once the function is declared, it can be used both inside the “TAKEOFF”

and “LAND” field or inside the global conditional field, “on_event”.

C.4 The Web Application

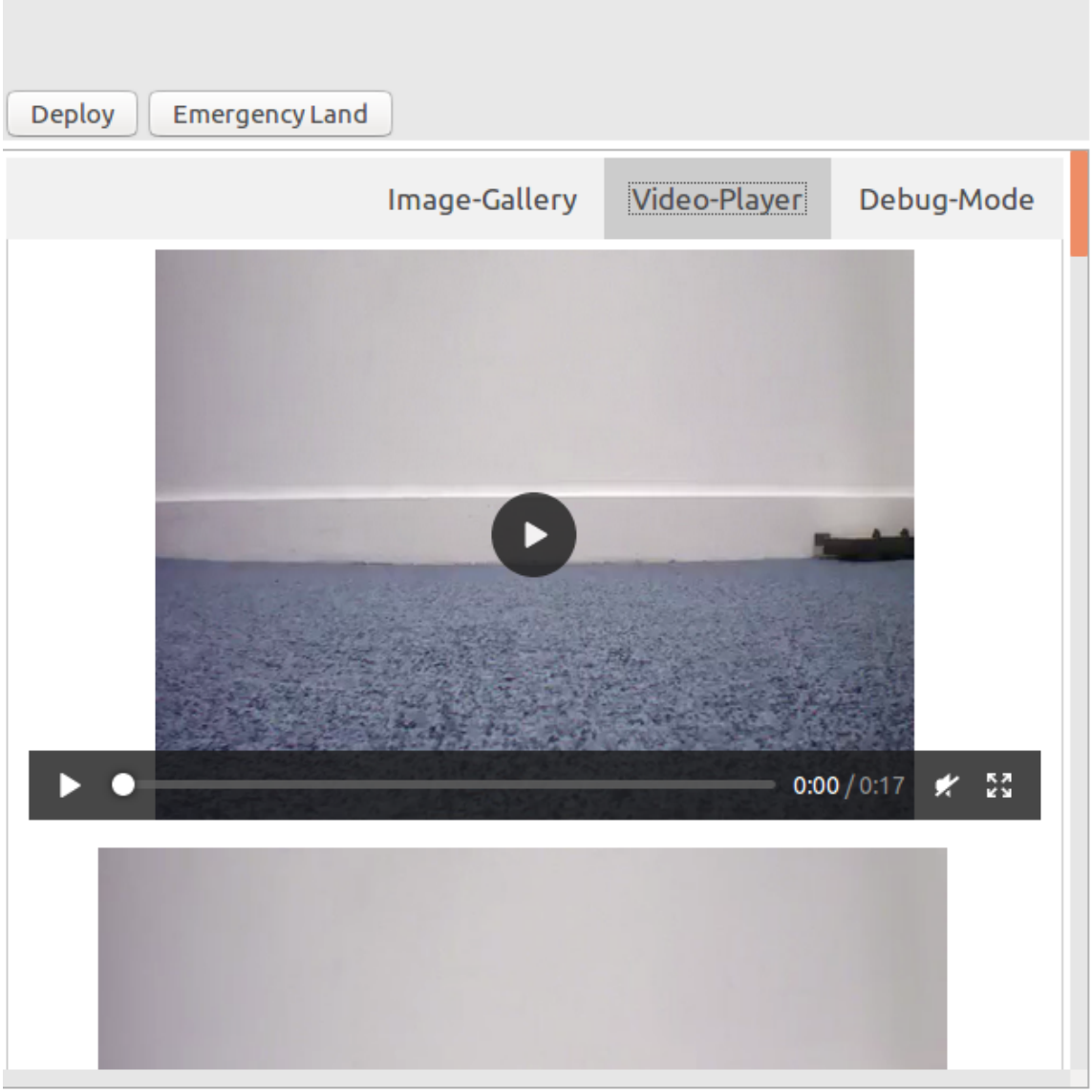
On the left side of the application an IDE is available for the user to define and write his own program. The image below shows the IDE with a sample program written on it.

The right side of the web application has two button at the top. “Deploy” and “Emergency Land”. Their functionality is self-explanatory. Below these button there are three tabs, by default the “Image-Gallery” tab is loaded when the application is first loaded or refreshed. The image bellow shows the image gallery.



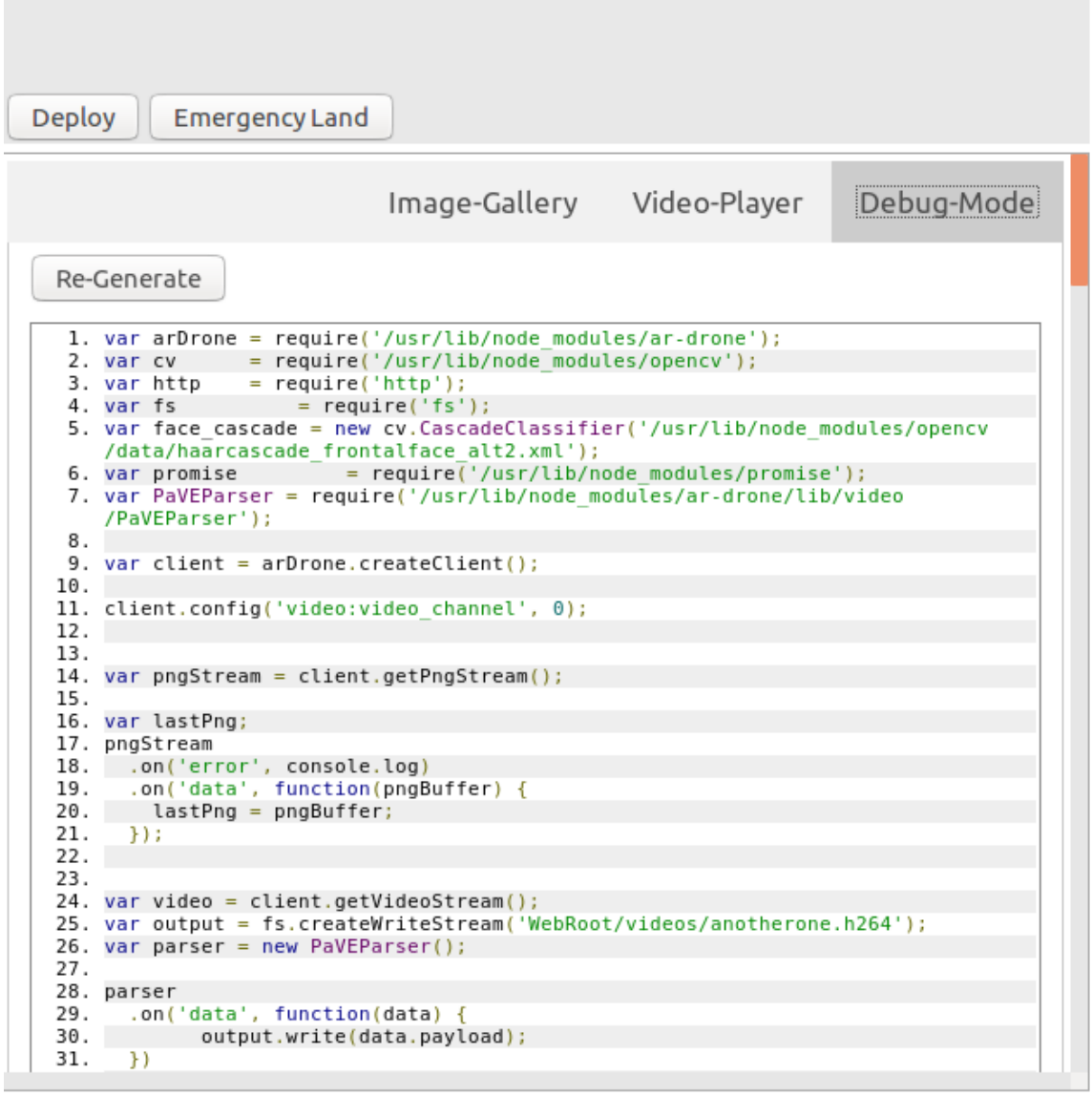
You can click on any image to view a larger version of it. These images are arranged by the date they were last added to the image directory. New snapshots taken in flight will hence be shown as the top image.

The second tab is a video gallery where you can access the videos recorded in flight.



The image above shows the video and its controls. Below the main player there is also a play-list where thumbnail images are displayed. By clicking on one of these images the video will be loaded in the player and you can view full video and make use of the video player's controls.

Lastly, there is "Debug-Mode" tab which can show you the code generated by the DSL. In this tab there is a "Re-Generate" button which also allows the user to re-generate the code if the program has changed without having to deploy the drone.



The image above shows an example of generated-code.