Imperial College London

PROJECT REPORT

DEPARTMENT OF COMPUTING

IMPERIAL COLLEGE OF SCIENCE, TECHNOLOGY AND MEDICINE

Drone Playground

Authors:
Matthew Baugh
Niranjan Bhat
Kexin Gu
George Ordish
George Soteriou
Maria Teguiani

Supervisors:
Dr Jaclyn Bell
Dr Madasar Shah

Contents

Chapter 1 Executive Summary

Teaching STEM subjects in primary schools is notoriously hard. Teachers often struggle to get students engaged in the subjects. This is often because the pupils' knowledge limits them from understanding the most interesting and impressive parts of those subjects. A common way to get around this is to use demonstrations, show the kids something that will catch their attention while also teaching them something about the subject at a high level. However this doesn't work very well with Computing, as most concepts are quite abstract or intangible. But now with Drone Playground teachers can teach computing concepts in a fun, easy-to-understand way.

Drone Playground gives students the chance to create code to control a drone and make it complete a number of tasks. These tasks cover a wide range of contexts, from fire fighting to parcel delivery, getting students to practise critical thinking and give them experience of interpreting more "wordy" questions.

As a teaching tool it is incredibly portable and easy to set up, as the entire system runs on a single Raspberry Pi and students can access it using a normal web browser (no installations required on their side). This makes it especially suited for university outreach, as it can be taken into a school and students can quickly get involved.

Overall, the main aspect of Drone Playground that makes it so engaging for students is the fact that they can see the drones flying right in front of them. This visual feedback makes identifying any mistakes much easier, helping the students understand how the code they have written runs.

Chapter 2 Introduction

2.1 Objectives

One tool which lots of schools use to introduce programming is the Scratch programming language [1]. Scratch is a block-based programming language where students write programs by connecting individual commands together like a puzzle. A big part of why Scratch is so successful is because of how simple it is, students can instantly understand the format and start writing programs right away. For Drone Playground to be an effective teaching tool it also needs to be very intuitive and accessible, so one of our objectives when making it was that a primary school student with no prior experience should be able to use the program with minimal explanation. This will make it much more practical and useful for teachers, as they will not have to spend large amounts of time answering basic usage questions.

As this project is targeting children, we know that in order to keep them engaged with our program we need to minimise waiting times. The main bottleneck of our system is flying the drones themselves, as it may take up to a couple of minutes to perform some of the longer programs. To mitigate this it is important that we're able to fly multiple drones at the same time. Simply maintaining multiple connections is not enough however, we need to also supply the teachers with the tools necessary to manage these drones effectively. For example, the teacher should be able to disable a drone if it's found to be faulty, and also add a new drone if the current number can't keep up with the number of flights. Without management capabilities like these teachers will not only be unable to fully take advantage of having multiple drones, but it may actually be more hindering than helpful.

Keeping a class full of kids under control is a difficult task at the best of times, and by adding drones to the situation there's potential for it to become overwhelming. Because of this it's important that our project does not put a huge amount of pressure and responsibility on the teacher running the class. Our software should automate as much of the drone and program management as possible, only requiring the teacher's input for approval and running of the programs. This does need to be balanced with giving the teacher control over the decisions being made, so in the case that an unforeseen problem arises the teacher can use their judgement to rectify the situation.

So to summarise, the main objectives of our project are:

- A simple, intuitive interface that students with no programming experience can easily use.
- Keep student waiting times to a minimum.
- Take as much of the burden away from the teacher as possible, whilst also providing more manual controls for them to use if they choose to.

2.2 Achievements

One of the big parts of our project that we're proud of is the UI. We're very happy with how we have managed to add a lot of functionality without making it feel too cluttered. This was especially challenging on the teachers interface, as we wanted to keep everything on one page so that the teacher didn't have to switch between views to get the information they need, but had to be careful not to overwhelm the user with information, as that could cause key notifications to be missed.

A less obvious part of our project that we're proud of is how well structured and extendable our code is. This made it much easier to add features that our supervisor requested, even when we hadn't considered them at all before that. A good example of this is the queue of programs waiting to be approved in the teacher interface. Initially this was a simple first-come-first-served queue, but Jackie recognised that there would be some situations where a teacher would like to review and approve a program further down the queue. For example if one group had not flown a program on the drone for a long time, it would be fairer to give them priority. Because of how we'd written the code it was quite easy to take our code and change it so it normally functioned as a basic queue, but if the teacher clicked on a specific program in the queue they could review it first.

We're also really happy with how we have managed the team as a whole. We have used Scrum by dividing our time into 2 week sprints, starting each with a planning session where we used Scrum poker to estimate the size of tasks, having stand-ups throughout the sprint and ending with a sprint review and retrospective session. We found the retrospective particularly useful as it gave everyone a chance to talk about how they felt the sprint had gone, and that better highlighted the problems we had. Then from there we were able to work on those problems in the next sprint, which meant that we never had a problem come up again in a later retrospective meeting.

Chapter 3 Initial State of the Project

3.1 History of the Project

This project had gone through three iterationswhen our team started working on it. It was first developed by a team at Oxford University in 2017 which focused exclusively on implementing a simple DSL developed with Xtext. It allowed writing basic programs for controlling the drone with commands like “go forward for 1 second”, “rotate by 90 degrees”, etc. [2]

It was then further developed in 2018 at Imperial College, where the language basics were the same but with an improved design and added functionality. This second iteration focused on developing features connected with the view from the drone’s cameras. The DSL was extended to allow users to specify which camera the drone should use when it recorded a flight or took a picture after executing a program. It included a website that acted as an editor and also had a gallery of all videos and images taken by the drone.

The third iteration further extended the language with a `GOTO` command and by specifying the environment around the drone. The website developed included an editor and a drone flight simulator.

3.2 Existing Solution

3.2.1 Project Architecture

In the form that was given to us, the project’s architecture was rather complex. It had multiple dependencies such as ROS (Robot Operating System) which provided a driver for enabling communication with the Parrot AR drone. Once the Xtext project was packed into a WAR file, it had to be copied into the Tomcat directory to get the server ready.

After this setup stage, the project required three processes to be run simultaneously: one for running ROS processes, one for the AR Drone driver that established the connection to the drone, and one for the Tomcat web server. Finally, the computer could then be connected to the drone via WiFi.

Since this is a tedious and time-consuming process, the project could also be built using a Docker container for each of the three processes and docker-compose to start them up together.

3.2.2 DSL

In terms of the actual DSL, it was extended from previous iterations with the ability to describe environment objects at set places in space, and a `GOTO` method used to fly the drone between two environment objects. When running xDrone programs, the input from the web editor was checked with the DSL grammar.

In order to simulate it, the xDrone code is translated into JavaScript files, which are then executed in order to perform the simulation. For flying the drone, a Python file is generated instead which controls the drone. In order to run this file, a bash script is run so that all dependencies are properly handled.

3.2.3 Simulator

One of the biggest parts of this project was the simulator next to the web editor (Figure A.1 in appendix). It also adds the objects defined in the environment part of the code to the simulation (labelled with the object's name for clarity), so that the `GOTO` commands can be visualised too.

The ability to see the behaviour of certain functions and test code without flying the drone is very useful. However, the colour scheme chosen for the simulator made it hard to use, since both the drone and the background were predominantly black.

3.2.4 Performance Assessment

The previous project achieved many of the goals it set out to do and a suggestion was that future work can focus on combining these features into an educational app, which is what our team aimed to do.

However, we soon realised that the existing architecture was very hard to maintain and extend. The code base was hard to understand as it was large due to many unused (and often empty) files and directories which were generated every time the project was built. It also auto-generated many gitignores, cluttering the git repository and still allowing binary files on git.

The server was written in Java, but it controlled the drone using Python, and simulated flights using JavaScript. This was needlessly complicated and meant that it had to use bash commands to coordinate the different parts of the system. Moreover, the frontend was hard to develop further, as it was tightly coupled to the rest of the project: the generated backend required keeping track of the ID of the editor and using cookies for session management amongst other things.

Finally, the ROS Docker image was not compatible with a Raspberry Pi and we wanted to use one to run the project in a classroom.

At the suggestion of Robert Chatley, we had an exploratory task of finding a viable replacement for the existing parser, as at that point in the project it was the only piece of the old software we wanted to use. We found the Python library Lark and successfully implemented all previous functionality in a more understandable and extendable way.

Chapter 4 Design and Implementation

4.1 Overview

Drone Playground consists of three parts: frontend (user interface), middleman server and backend. The frontend is the interface used by both students and teachers. Students use the block-based language to write programs to control drones to complete tasks, see what the program would make the drone do with a simulator and send requests to the teacher to fly the drone with their code. The teacher has a single page where he or she can check students' requests and allocate drones for approved code. The middleman server serves the frontend and also communicates with the backend by sending the code to the backend and receiving each flight's status. The backend is used for parsing the xDrone program and then executing the program on the drone.

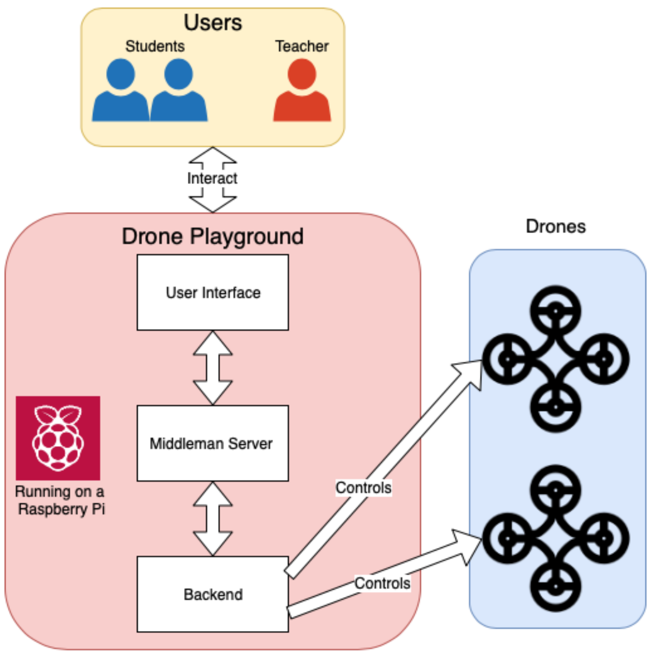


Figure 4.1: Diagram of Drone Playground's Architecture

For the frontend we use Vue.js [3], a lightweight and reactive JavaScript framework. We were able to write readable code quickly due to Vue's clear syntax, and maximise our code re-usability by extracting common parts of the UI into Vue components. The middleman server is a Node.js [4] backend with Express.js [5] framework. Node.js provides scalability and is deadlock-free due to its single-threaded event loop mechanism, making it simple to set up and easy to work with. Our backend is written in Python with Flask framework [6], including an xDrone DSL compiler and a controller using PyParrot [7] library to fly the Parrot drones.

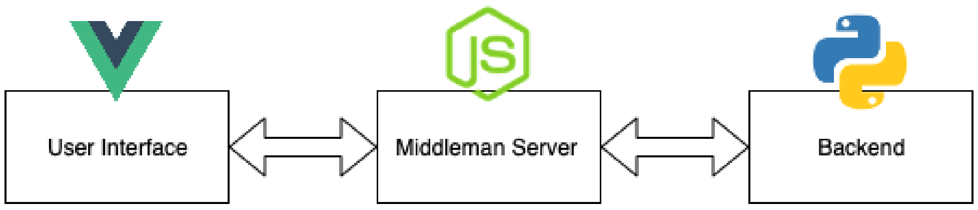


Figure 4.2: Diagram of Technologies Used in Each Component.

We decided to use this three-part structure instead of a classic frontend-backend structure in order to separate both the compiler and the drone controller from the frontend server. By extracting the compiler and drone controller out into the backend, we have a clearer structure of our code in three parts, separating it according to functionality. The backend is totally independent and could be used as a stand-alone compiler and drone controller without any ideas of students and tasks, making it reusable for other situations, for example, an xDrone language drone controller web application. This structure is also a way to avoid unnecessary code complexity. There are Parrot drone controlling libraries in both JavaScript and Python, but unfortunately the JavaScript library lacked functionality when compared to the Python library. Since our drone controlling had to be done using the Python library, it is a lot easier to implement a parser in Python and then, in the same program, go through the AST and call the library's APIs to control the drones. We could instead do the compiling in the middleman server, but we then need to generate a python script and spawn another process to run the script, which is unnecessarily complicated.

4.2 Frontend

4.2.1 Why Vue.js

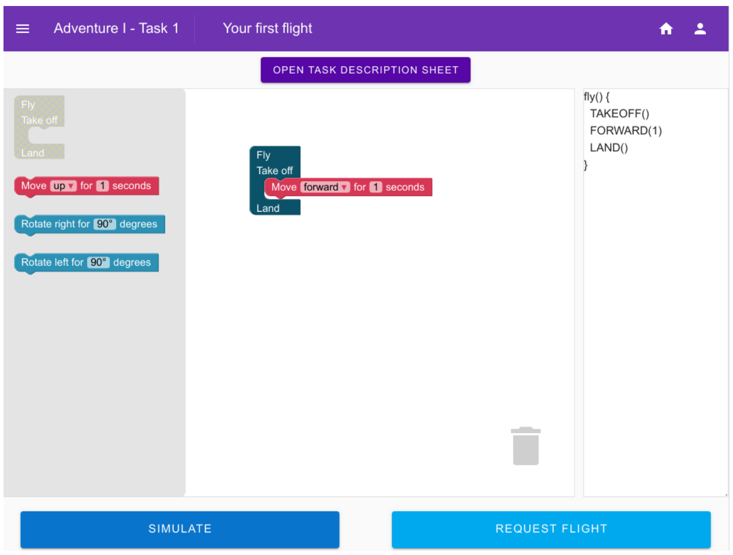
For the frontend we used Vue.js and we chose it instead of other frameworks for a number of reasons. Compared with React [8], AngularJS [9] and Angular [10], Vue is a lot smaller in size and has an easier learning curve as it only needs knowledge of HTML, CSS and JavaScript to do everything, unlike React which heavily uses JSX and CSS in JS and Angular which uses TypeScript. Code can be written in a more modular and readable way in Vue compared to Angular and AngularJS. AngularJS and Angular both specify the architecture and how the application should be built, while Vue provides a more flexible approach. In Vue, each page is treated as a view and each view is made of one or more components. Constructing an application is then simply creating reusable components and combining them to make responsive views. All components are self contained in their own files, unlike Angular and AngularJS where controllers, data binding etc. are separated from the components, making the coordination of components messy and difficult to understand.

Vue also requires less effort to optimise and has higher performance. In React, when the parent component changes all the children components are re-rendered with the parent, whether their own states have changed or not. To avoid unnecessary re-render, `shouldComponentUpdate` needs to be implemented in each component and it is not a reliable way for optimisation. In AngularJS, all the watchers in a component are reevaluated in a loop called digest cycle that is triggered when the state has changed, and the digest cycle may have to run several times to wait for the state to be stable, making optimisation very complicated, even impossible in some cases. However, Vue uses getters and setters of properties to perform dependency checking, and the single watcher of each component instance will be pushed to the async update queue once. With these approaches Vue effectively avoids unnecessary re-rendering and speeds up its performance.

4.2.2 Vuetify

On top of Vue we used Vuetify [11], a component framework, as our UI design. Vuetify provides a wide range of component templates, all followed the Material Design Specification [12] with detailed documentation, allowing us to easily design Drone Playground in a simple and consistent way. Vuetify has a weekly release cycle, which is the shortest among some of the popular Vue frameworks we looked at, guaranteeing that users get the latest version with new features and bug fixes quickly.

4.2.3 Student interaction/interface



```
fly() {
  TAKEOFF()
  FORWARD(1)
  LAND()
}
```

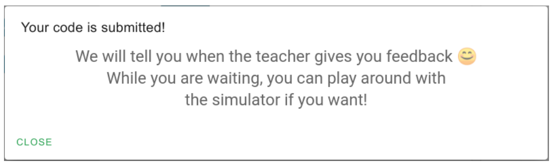
Figure 4.3: Sample Task page

Throughout the whole student side of the project we focused on keeping a consistent, intuitive and child friendly user interface. We broke down the lessons into adventures and tasks, such that each adventure has a consistent theme with a few tasks to teach the students while going through a story, in order to keep them engaged. The task view (Figure 4.3) is the main interaction point of the student. It contains four main components. The task instructions and story, the main Blockly code editor, the simulator, and the submit button.

Blockly Code Editor

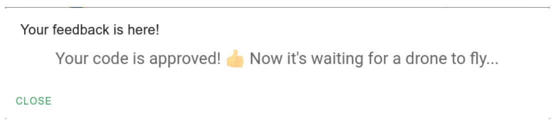
We decided to use the Blockly [13] library to create a beginner friendly interface for students, and allow them to easily get started by dragging and dropping commands for the drone. We forked the library and extended it with our own blocks, that translate directly into the xDrone language. More blocks can be added for more functionality in the future.

Code Submission


Your code is submitted!

We will tell you when the teacher gives you feedback 😊
While you are waiting, you can play around with
the simulator if you want!

Figure 4.4: Submit Code message

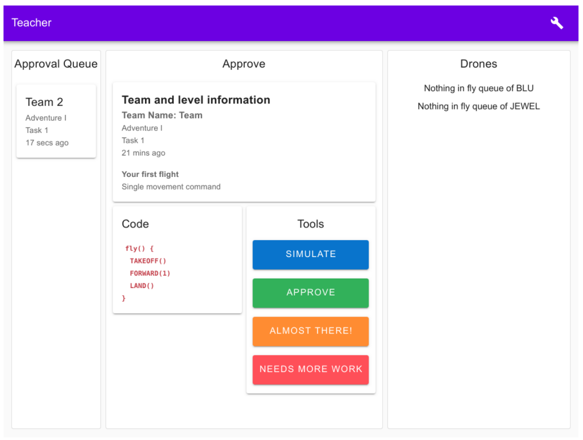

Your feedback is here!

Your code is approved! 👍 Now it's waiting for a drone to fly...

Figure 4.5: Sample feedback message

When students click the submit button, they are greeted with a message informing them that the teacher will now check the code (Figure 4.4). They are then free to change the local version of the code and continue trying new code on the simulator until the teacher approves or rejects the code they submitted. When a submission is made, the request flight button changes to cancel, so that the students can also withdraw the request if they want to. Messages similar to Figure 4.5 are shown when the teacher approves or rejects the request.

4.2.4 Teacher interaction/interface



Team 2
Adventure I
Task 1
17 secs ago

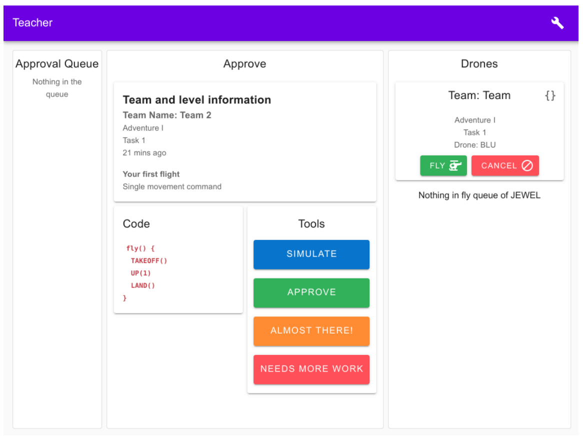
Team and level information
Team Name: Team
Adventure I
Task 1
21 mins ago

Your first flight
Single movement command

Nothing in fly queue of BLU
Nothing in fly queue of JEWEL

```
fly() {
  TAKEOFF()
  FORWARD(1)
  LAND()
}
```

Figure 4.6: Teacher page with one request in the queue and another to be approved



Nothing in the queue

Team and level information
Team Name: Team 2
Adventure I
Task 1
21 mins ago

Your first flight
Single movement command

Team: Team
Adventure I
Task 1
Drone: BLU

Nothing in fly queue of JEWEL

```
fly() {
  TAKEOFF()
  UP(1)
  LAND()
}
```

Figure 4.7: Teacher page with an approved request, assigned to a drone

The teacher interface consists of 3 main components, the Approval Queue, the Approve Tab, and the Drones Tab. The requests are stored in a first-come-first-served queue where the top request is shown in the Approval tab and all other teams in the Approval Queue (Figure 4.6). In the Approve tab, the teacher has access to information about the current team, the task they are on and the code submitted. The teacher can then choose to simulate, approve, or reject (two types: "Almost there" and "Needs more work") the request. The simulator component is the same as the student simulator and will be discussed in Section 4.2.5. When the current request is dealt with (approved or rejected), the top item in the queue will be promoted to the Approve tab. The teacher may choose to ignore the current top request, leaving it to be reviewed later, by clicking on another task from the queue.

Once a task is approved, the server will automatically assign it to an enabled drone with the fewest tasks in its internal queue (Figure 4.7). It will then be ready to fly. The teacher may then click the fly button at any time to make the drone execute the code that students submitted.

4.2.5 Simulator

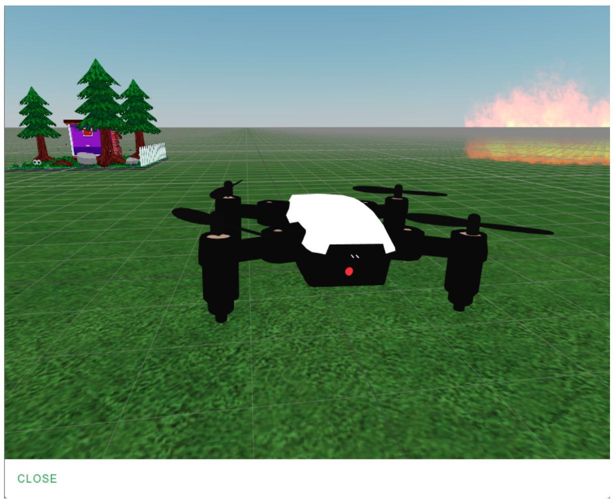


Figure 4.8: Sample Simulator component

The simulator component in Figure 4.8 uses the WebGL library Three.js [14]. On a "simulate" request, the xDrone code created from Blockly, is sent, through the middleman server to the backend server as a POST request, where it is parsed and converted to a JSON object and sent back to the simulator. The simulator will then use that JSON object to conditionally render objects in the environment (according to the task requirements) and move the drone in that environment according to the input code. We used free models found online for the environment objects and the drone. In the future, more, custom objects could be created.

4.2.6 Middleman server

We used socket.io [15] for real-time updates for both students and the teacher. The middleman server also has a small HTTP part that serves the frontend, and also forwards fly and simulate requests to the backend as can be seen in Figure 4.9.

Below are some key real-time interactions we implemented via socket.io.

Connection and WHO
On a connection, the server emits a WHO response to the newly connected user. If the user is a new team of students, the user sends back new_user along with the unique team name. An internal `uidToSocketId` map is then updated to include the appropriate mapping. If the user is an existing team (e.g. the user refreshed the page), then the user responds with an IAM along with the unique team name. The `uidToSocketId` map is then updated with the correct socketID for that user. If the user is the teacher, the user emits a TEACHER response, and then the server replies again with an UPDATE along with the current state of the server shown in Figure 4.10.

Request_flight
When a user requests a flight, a new Team Flight Request is added to the main queue along with a timestamp of the request. Then, an update is pushed to the teacher, with an updated queue.

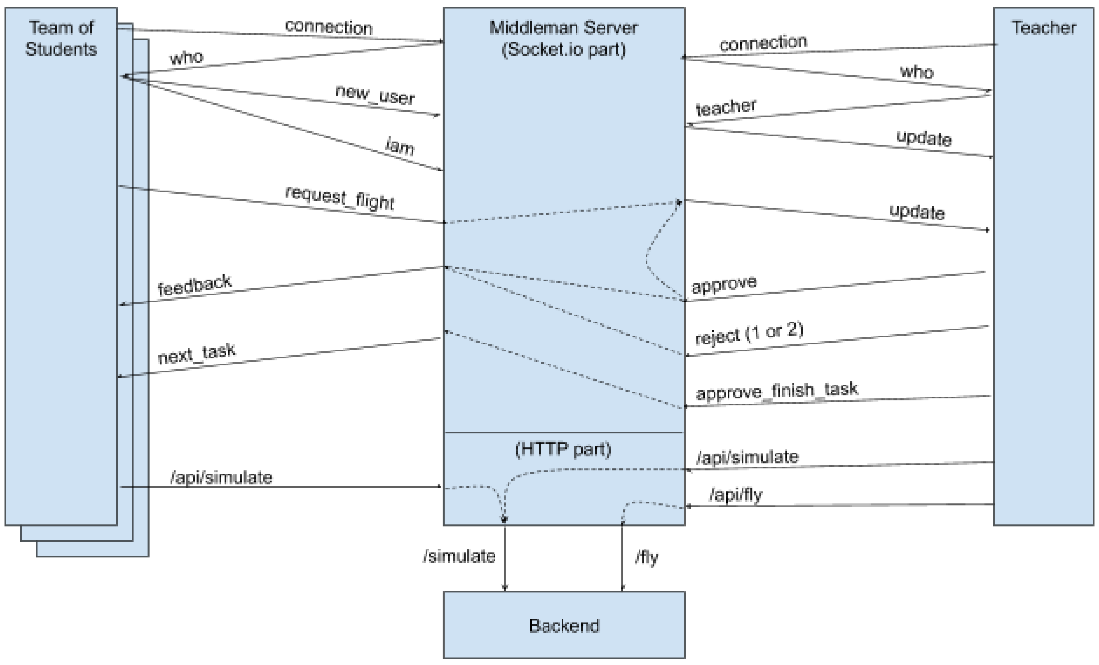


Figure 4.9: Middleman interactions

Approve and Rejects
When a teacher approves a Team Flight Request, the enabled drone with the shortest queue is selected and the team flight request is moved to that drone's queue. Then an update with the new state is sent to the teacher, along with positive feedback to the student. When a teacher rejects a Team Flight Request, it is removed from the queue. Then appropriate feedback is sent to the student.

Approve_finish_task
When a drone flies successfully and finishes the task description, the teacher can choose to 'complete' the task. This will remove the request from the drone's queue and emit a NEXT_TASK response to the user, moving them on to the next task.

Other Interactions
Other smaller socket.io quality of life interactions exist that have not been documented here, but are well documented in the code (changing the submit code button into a cancel request button if the team has already submitted code, disabling new user logins, logging all users out, disabling drones and adding new drones to the system, attention popup).

4.3 Drone Flying Server (Backend)

The backend consists of a parser for the xDrone language, a visitor that can traverse the abstract syntax tree to fly a drone and another visitor that generates code for the simulator. The backend is designed such that all the code that deals with the xDrone language and flying the drone is in a separate library, making it reusable in other applications. A Flask web server with multiple exposed API endpoints uses this library as appropriate.

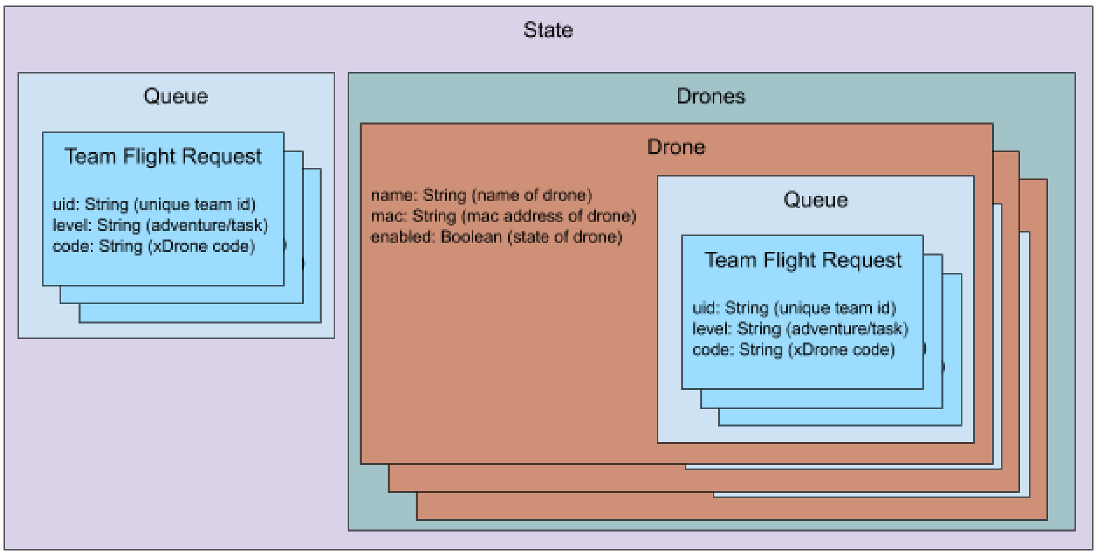

State
Queue
Team Flight Request
uid: String (unique team id)
level: String (adventure/task)
code: String (xDrone code)
Drones
Drone
name: String (name of drone)
mac: String (mac address of drone)
enabled: Boolean (state of drone)
Queue
Team Flight Request
uid: String (unique team id)
level: String (adventure/task)
code: String (xDrone code)

Figure 4.10: Middleman state

4.3.1 Why Python, and Other Design Decision Justifications

We chose to implement the backend in Python as the Parrot API was easy to use via the Pyparrot library and the group was confident using both Python and Flask, a web framework for Python. Flask makes setting up the backend server and enabling CORS straightforward and simple. CORS (Cross-Origin Resource Sharing) allows a web application running at one origin to have access to resources requested from a different origin. This was necessary for us, as both our front-end and middleman servers may be hosted on a different device. Lark [16], a parsing library for Python, enabled us to re-implement our own version of the parser with significantly fewer lines of code. Lark uses an efficient implementation of LALR(1) that is effective at resolving common terminal resolutions, as the lexer only matches the terminals that are legal at that parser state.

4.3.2 Drone Calibration

One of the limitations of the Parrot API was that it did not support high level commands like “move forwards (x) metres” out of the box - all movements had to be done by varying the pitch, yaw and roll (these are shown in a Figure **??**) angles of the drone while in flight.

Moving forward was achieved by changing the pitch of the drone, for a particular period of time. For the purposes of this project, we set the pitch or roll for movement to be a constant value of 10 degrees from the horizontal plane. However, due to real world inaccuracies such as wind, air pressure and terrain of the ground below the drone, there was no guarantee that each repetition of the same command would yield the same distance moved. We carried out a number of trials to determine how far the drone would move when told to tilt for a given number of seconds and found that on average the drone moved around 35cm horizontally per second. Similar experiments were carried out to determine how long the drone should be held in upward acceleration to achieve a certain amount of vertical elevation, after we determined that the drone rose to an average height of 83cm after executing the in-built takeoff command. These found that the drones moved at an average speed of 10cm per second vertically. All this information was used later to estimate the position of the drone when automatically grading the tasks.

4.3.3 Our Own Implementation of Lexer and Parser

Lark, a modern parsing library for Python, was used to write a parser for the xDrone language. Lark grammars are specified in a variant of EBNF, which made it fairly straightforward to use the specification of the xDrone language from Xtext implementation. After a few small modifications, the language can be parsed by Lark and Lark's automatic tree construction, which is inferred from the grammar, was used. The shape of the tree was altered slightly by using some of Lark's extensions to EBNF, specifically to reduce the depth of the tree, by combining sibling-less nodes with their parents.

Lark provides two ways of using the information in the tree: visitors and transformers. To simulate the flight, a transformer that traverses the tree and performs a simple transformation at each node is used, with some changes to the overall structure. This produces a list of commands that can be run by the 3D simulator. As we already ensure that the program is valid by the time the tree is generated, flying the drone is accomplished by simply traversing the AST and making the corresponding Parrot API calls.

The parser is easily extensible. Features can be added to the language by modifying the `.lark` file, which specifies the language. Handlers only need to be added where they are useful. For example, if the addition makes no difference to the simulation then `simulate.py` does not need to know about the new command, and similarly if the new command only used in the simulation, `fly.py` can be left unchanged.

4.3.4 Validator

The validator traverses the tree generated by the parser. The validation occurs mostly within the xDrone library. It uses the information from all the movement commands to create an estimate of the bounding box of the drones flight. This box is then compared with the predefined box that the drone is allowed to fly within. The validate endpoint of the API will return a failure if the drone is estimated to leave the box. This does not stop the flight from being submitted to a drone to fly but can be used as a warning for whether the flight is likely to be safe.

4.3.5 Automatic Grading of Tasks

The "fly" visitor controls the drone we keep track of its estimated position (using the data we collected when we calibrated the drones). Using this estimated position we calculate whether the drone has completed all the requirements for the current task. In the backend we don't need to know the exact scenario and instructions the students have been given, instead we represent the requirements in an abstract framework consisting of the following types of requirements:

- Instant actions: Drone executes action block while in a given area
- Continuous actions: Drone executes an action in one area, flies to another area and executes an action again
- Avoidance: Drone is not allowed to fly in a given area, doing so fails the task
- Landing: Drone must land in a specific area

The requirements are sent to the backend together with the student's code in a JSON format, from which we generate a list of requirements objects. Each type of requirement has its own class which extends the Requirement class. This contains a number of methods for updating whether the requirement has been completed: `update_on_move`, `update_on_action` and `update_on_land`. By default these do nothing, so each child class only needs to override the method which is relevant to it (for example the instant action only overrides the `update_on_action` method). Using classes like this makes adding more requirement types very easy, as we only need to create a new class for it and add it to the requirement generator.

4.3.6 Sanity checks and the test suite

The backend also consists of two different test suites. A parser test suite is provided to ensure that any xDrone program being received from the backend is syntactically correct, and isn't missing any key components of the program. A simulation test suite is provided and tested with example programs to ensure that the simulated final state of the program is the same as the expected final state of the program. Testing is implemented with pytest [17], a robust testing library that is used in the industry to write unit tests and small tests.

Chapter 5 Evaluation

The student-teacher interface has met all of our initial goals – it's better looking than before and adds new functionalities without making it feel cluttered or overwhelming. The simulator has also been greatly improved, with clearer and more interesting graphics, helping students to understand the task at hand. One of our objectives was to make drone programming accessible to students who had little to no experience in programming. Using Blockly has certainly achieved this, as it provides a simple interface that students find very intuitive. In the future to make it more challenging and realistic for more advanced students, the option to edit code directly could be added as this would give them an experience of writing code in a specific syntax.

Another main objective is to minimise waiting time of students, which we achieved by separating the whole process into an approval queue and a flight queue. This enables the teacher to continue reviewing submissions while waiting for the drones to finish their current flights, letting students get feedback faster. Also, as we can fly multiple drones simultaneously, the programs which are approved get run more quickly. On top of that the teacher has the flexibility to approve code from anywhere in the queue instead of only the one at the top, helping those students who haven't had a turn in a long time get feedback soon.

Our final objective, reducing the burden on the teacher, has been difficult as in many situations we have to keep them involved to ensure the drones are flown safely. Because of this we've made an effort to handle all the administrative (not safety-related) decisions for the teacher, such as which program to review next, who's turn it is to fly and whether the flight has successfully completed the task. Our supervisor Jackie (who has experience leading primary school lessons) is happy that our software makes coordinating the drones simple enough that a teacher can manage it without diverting their focus from pupils.

Outside of our objectives, our implementation of the project reduces the complexity and size of the project compared to the previous incarnation. The previous version has around 41k lines of code, compared to ours that has just 500 lines for the backend, including a parser for the xDrone language, and 19k lines of code for the frontend, which has added functionality, mentioned above, and has been written in modern JavaScript. Not only has the code complexity dropped but the project is a lot easier to set up to run on a new device. The project is intended to be used in a classroom, so reducing difficulty of initial setup for the teacher was one of our main priorities.

The feedback we received from our supervisors and peers who tested the project was overwhelmingly positive, though there was a sentiment that the drones weren't as accurate as they would have liked in terms of movement. Although they would be suitable for use with children in a big enough area, with an additional safety buffer around it, the Mambo drones we were using did not seem to be precise enough to use in a small indoor space. Using higher accuracy drones could improve this situation, however, these can be significantly more expensive than the drones we used, and so unsuitable for use in primary schools where they may well be damaged. The main problem behind this is that the Mambo drones have no localisation, meaning that they are susceptible to drifting. Drones can be made more accurate if their motor controls are combined with input (such as camera data), but drones capable of this are significantly more expensive as they require both a camera and a processor that is capable of processing the data in real time.

If it is decided that physical drones should no longer be used, there are still large parts of our project which can be repurposed and reused. For example, it would be relatively simple to keep all the language elements except UP and DOWN, and then swap the backend to be controlling a remote controlled car or robot. All the teaching elements (reviewing code, scheduling running the programs etc.) could remain unchanged. Alternatively the focus of the program could be moved from flying physical drones to using the simulator. Now that the simulator is much more expressive, it is able to clearly show the students the effects of the code they have written. The only work that would be needed to make this possible would be to link the simulator to the requirements of the task, which would not be difficult. This solution would also remove any need for teacher to intervene, letting them fully focus on helping the students.

Without changing the direction of the project, as the core functionality of the program is fully covered, the drone project could be extended further by adding elements such as achievements and unlockables. Unlockables could include pre-made routines for the drone to perform, such as swooping or flipping, with a new one being awarded at the end of each adventure. Features like this would make the experience more engaging for students, as they are worked towards real goals rather than completing arbitrary tasks.

Overall we're very happy with the quality of the software we've produced. It's successfully met all our initial objectives and iterated well with our clients, which has all come together to form a user-friendly and functional product that can be use to effectively teaching programming to primary school students.

Chapter 6 Ethical Considerations

As our project is aiming to build a teaching tool for school children, we needed to ensure the safety of drone flying in a classroom of students. This issue was mentioned early on in the meetings with our client Jackie and we worked with her in order to implement features that addressed all of our safety concerns. This is why we decided to limit the area each drone can fly in to a 4×4 metre space, which can be clearly sectioned off in the classroom so that the students will not step inside it and risk injuries. This fly area is also enforced through code, as we added automatic safeguards on flights.

To minimise the risk of drone flying, we also added more features accessible to teachers. In the teacher's view, they can review the code and run it in the simulator, so that they can make sure that it is a problem-free flight before the drone is started.

Since last year, a regulation came into force stating that drones of over 250g need to be registered with the Civil Aviation Authority if flown outside. For our project, we used smaller drones and they would be flown inside a classroom, so this regulation will not affect us. For future work involving different drones or scenarios, it is something that needs to be considered.

Bibliography

[1] Scratch - About. `https://scratch.mit.edu/about`. pages 2

[2] L. Niepolski. Simulation, route planning and environment modelling in xDrone a Domain Specific Programming language for drones, September 2019. pages 4

[3] Vue.js. `https://vuejs.org/`. pages 6

[4] Node.js. `https://nodejs.org/en/`. pages 6

[5] Express.js. `https://expressjs.com/`. pages 6

[6] Flask — The Pallets Projects. `https://palletsprojects.com/p/flask/`. pages 6

[7] amymcgovern/pyparrot. `https://github.com/amymcgovern/pyparrot`. pages 6

[8] React - A JavaScript library for building user interfaces. `https://reactjs.org/`. pages 7

[9] AngularJS – Superheroic JavaScript MVW Framework. `https://angularjs.org/`. pages 7

[10] Angular. `https://angular.io/`. pages 7

[11] Vue Material Design Component Framework - Vuetify.js. `https://vuetifyjs.com/en/`. pages 8

[12] Material Design Introduction. `https://material.io/design/introduction/#`. pages 8

[13] Blockly — Google Developers. `https://developers.google.com/blockly`. pages 8

[14] three.js – JavaScript 3D library. `https://threejs.org/`. pages 10

[15] Socket.IO. `https://socket.io/`. pages 10

[16] lark-parser/lark: A modern parsing library for Python, Implementing Earley LALR(1) and an easy interface. `https://github.com/lark-parser/lark`. pages 12

[17] pytest: helps you write better programs – pytest documentation. `https://docs.pytest.org/en/latest/index.html`. pages 14

Appendix A First Appendix

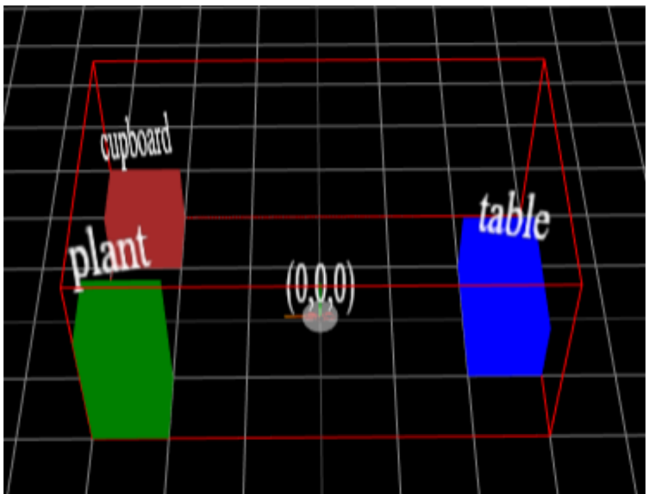


Figure A.1: Old simulator - the drone is at (0,0,0)

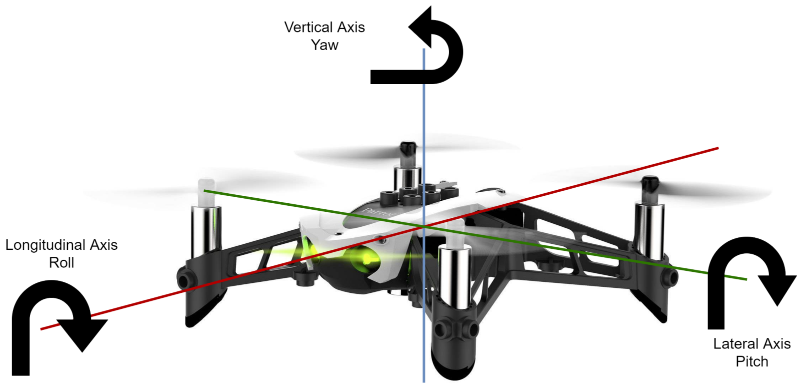


Figure A.2: Roll, Pitch, Yaw