Title:	AutoDrone
Student:	Andrew Festa
Faculty sponsor:	Jeremy Brown
Academic term:	Spring 2020 - 2195
Words:	4451
Characters:	21163

Table of Contents

1 Introduction

Most small, DIY drones are configured to be controlled completely from either a remote control or from an on-drone auto-pilot. The former are generally limited by the number of control channels on the transmitter, and the pilot must have some method for observing the environment in which the drone is operating. The latter configuration is limited by the power and resources provided by the on-board computer. The most important resources, computational capability and power, are typically heavy and expensive, making resource management and system addons a critical design decision. Can a camera be added to the board without overly taxing the already limited processing power? What about a tracking algorithm? Many approaches for these problems are becoming more and more resource demanding, such as training of neural networks or running reinforcement learning agents, and so a different configuration might be better suited to building an extensible and scalable flight control system.

The ultimate goal pursued during this project was two-fold: to explore machine learning and control systems on a resource-constrained system and to offer practical experience into the design decisions required for building such a system. An additional benefit is the end-creation of a computer controlled drone which can be readily extended for control by other control mechanisms, such as BCIs or VR headsets.

This project bears similarity to CSCI-632, Mobile Robot Programming. However, this project differs in that flight offers a particular set of challenges compared to ground-based navigation. Where ground-based navigation effectively operates in a confined, 2D space, flight systems operate in a 3D environment. This differentiating factor is also a challenge for humans first learning to fly. Apparently, the best response to going into a power-on stall is not to go full lean on the mixture, unless you're looking to terrify your CFI. Beyond that, resource management is vastly more important when controlling a system in the air. When on the ground, if power runs out, you can simply pull to the side of the road or park where ever you may happen to be. In the case of flying, this option is tantamount to crashing.

1.1 Summary of Work Accomplished

The initial tasks to be completed, as laid out in the proposal, may have been a bit ambitious given a lack of a verified drone to control or, alternatively, a simulation environment. Still, the base goals of the project were able to be mostly accomplished. There is a working drone control utility that allows for various methods of user input. The most stable methods are through manual controls on the keyboard (or clicking on the GUI, if you're a heathen) and through a speech recognition library. The gesture control is currently rather unstable. So, for safety's sake, the actual use of this input method isn't exactly recommended. Especially for indoor operation. Videos of both of these methods of operation are in the github repository.

1.1.1 Project Phases

At a high-level, the project is best seen as two parts that can be worked on in parallel: the drone control and the user-input methods. The drone control ended up being much more challenging and time consuming than expected, though I blame DJI and Ryze for that (not at all out of bitterness, mind you). Certain aspects of the user-input methods were easier to get working, although the present much more challenging issues to overcome for the long term and for improving on the system.

2 Hardware

As this project deals with operating a system in a 3D environment, a drone is the obvious choice due to its stable flight characteristics. Another option may have been a simulated environment, but where's the fun in that?

Figure 1: Tello Drone

2.1 Initial Approach

The original intent of the project was that I would create a drone from scratch, as this would allow me complete control over all the inner-working and communication protocols involved with controlling the drone. This approach started off going well until Amazon experiences several issues with shipping parts I needed to build the drone. Needless so say, I took a step back and re-evaluated where I was and what I was looking to accomplish. I had intended to work on both sides of the problem: the drone and the control system, but this delay set me back as I had to find another drone that I could communicate with and control wirelessly, or I had to build a way to simulate the drone and the environment.

2.2 Tello Drone

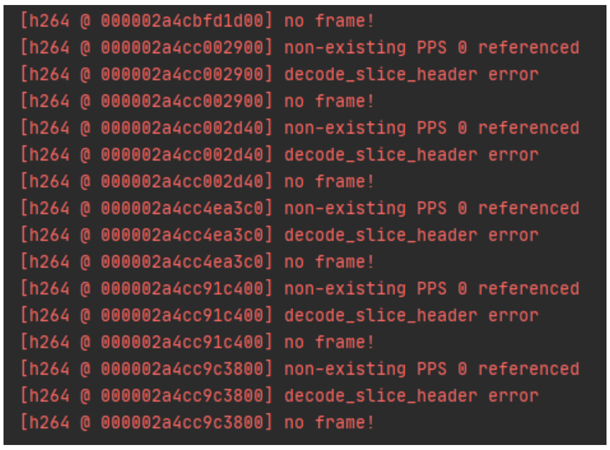
Enter stage left: the Tello drone. This seemed to be a suitable replacement for what I was looking for, and at just a little over $100, it didn't hurt too much. The little toy shown in figure 1 arrived fairly quickly, and I got to it, scrapping the majority of the prep work I had done for building the drone. That could save for a rainy day.

I hit the first snag fairly quickly: the coding environment. DJI's official library and all of their examples are written in either Scratch or Python 2.7. That just wouldn't do. But there were several other reference libraries I could look at, and DJI has an SDK document that documents the commands and how the drone is supposed to react to those commands.

2.2.1 Implementation Details

The drone expects messages to be sent over UDP to port 8889, and it responds over several ports (8889, 8890, and 11111) based on what it is sending to the host system. Port 8889 is for responding to messages, 8890 is dedicated to a state stream, and 11111 is for the video stream. That all seems to be reasonable until you get in to actually using their implementation. The first issue is with regards to how it responds to commands. Being as it's UDP, we have no guarantee that a packet does not drop or arrive out of order, and in fact, this appears to not be a completely atypical occurrence. This would be fine if each message had an associated ID or some way to correlate commands to responses, but the only response the drone ever seems to actually send back is ok or error. After playing with several solutions, the easiest solution was to space out sending messages where the response was critical. This was made easier by the fact that most of the actual *critical* messages are sent during initialization, not during actual flight. Thus, the system allows to send a message and either expect or response or not. It also keeps track of how many messages are sent that do not expect a response, so it might be theoretically possible to back-correlate which response goes with which message, but this presented much more of a headache than warranted and wasn't really all that important of an issue to solve. So, as with any good software engineering hurdle that you don't want to solve, you mark it as *out-of-scope* and move on.

Using UDP presents one more issue that was largely not anticipated. It would seem reasonable to



```
[h264 @ 000002a4cbfd1d00] no frame!
[h264 @ 000002a4cc002900] non-existing PPS 0 referenced
[h264 @ 000002a4cc002900] decode_slice_header error
[h264 @ 000002a4cc002900] no frame!
[h264 @ 000002a4cc002d40] non-existing PPS 0 referenced
[h264 @ 000002a4cc002d40] decode_slice_header error
[h264 @ 000002a4cc002d40] no frame!
[h264 @ 000002a4cc4ea3c0] non-existing PPS 0 referenced
[h264 @ 000002a4cc4ea3c0] decode_slice_header error
[h264 @ 000002a4cc4ea3c0] no frame!
[h264 @ 000002a4cc91c400] non-existing PPS 0 referenced
[h264 @ 000002a4cc91c400] decode_slice_header error
[h264 @ 000002a4cc91c400] no frame!
[h264 @ 000002a4cc9c3800] non-existing PPS 0 referenced
[h264 @ 000002a4cc9c3800] decode_slice_header error
[h264 @ 000002a4cc9c3800] no frame!
```

Figure 2: Video Decode Error

use UDP for a video stream. If we lose several packets, meh, not the end of the world. Expect for that it is streaming the video in FFMPEG. Dealing with the issues that arose, I learned that FFMPEG apparently doesn't transmit in full frames, and so if several frames are dropped, this causes errors in decoding the video stream. And given the low-resource nature of the drone, frames could be dropped fairly regularly, as previously noted, but not necessarily consistently. Sure, suppress the error and move on to the next part of the project. Except that the underlying library in Python best suited to the task (OpenCV), is written is C++. The Python calls are bindings to the C++ library, meaning that these rather annoying error messages couldn't be simply suppressed without rewriting several portions of the underling library and recompiling the source. Not a light task for such a massive library, especially on a Windows host. In the end, it's annoying, as shown in figure 2, but it doesn't actually break anything. The stream can be read and the frames saved into a video.

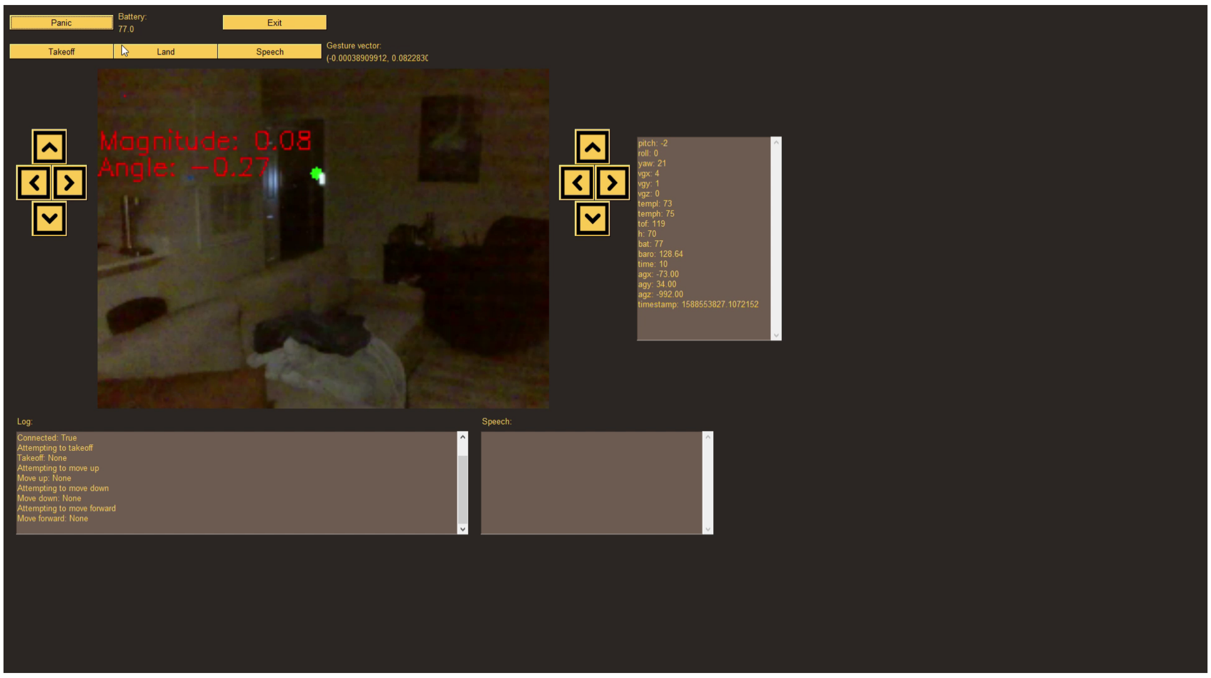
The final large issue with the Tello implementation is with respect to the actual SDK. It turns out that it's just wrong. Well, mostly wrong. There are a few parts that are right, such as the majority of sending and receiving messages. However, one little tip: the latest SDK only pertains to a particular model of the Tello drone: Tello EDU, not the base model. And the base model's firmware can't be upgraded to support the latest SDK, but I can't find any official reference to this fact. Again, however, this wasn't too major of a concern. The drone just responds with an error when you send an incorrect command. So... simply don't send that command.

Until you get to the control methods. They all respond as if everything is working properly, but the drone doesn't actually move. The actual docstring for the TelloDrone.py module describes this a bit more in detail, but most of the *control* commands don't have any effect. Instead, the drone should be controlled through use of the rc command, which takes 4 integer values as parameters. These parameters set the speed of various control surfaces on the drone and effectively operate as if you were using a typical remote control.

Sorting out these issues is, admittedly, still a work in progress. Mostly in making the operation of the drone stable. Connecting to the drone's broadcasted network can be a bit finicky, and if anything goes wrong in any portion of the code, it may leave the drone in a state where the easiest solution is to just hard-reboot the drone and go through connecting to it's network. Version 2.0 of the SDK details being able to configure the drone to connect to a wi-fi network rather than having to connect to the drone's network. But this is not an option for version 1.3, which is the version supported by the base Tello model.

3 Software

Where the hardware component of the project was focused on designing and building an actual drone, the software component was focused on various methods for processing the drone's information and controlling the drone using some method of input.



```
pitch: -2
roll: 0
yaw: 21
vgx: 4
vgy: 1
vgz: 0
templ: 73
temph: 75
tof: 119
h: 70
bat: 77
baro: 128.64
time: 10
agx: -73.00
agy: 34.00
agz: -992.00
timestamp: 1588553827.1072152
```

Log:

```
Connected: True
Attempting to takeoff
Takeoff: None
Attempting to move up
Move up: None
Attempting to move down
Move down: None
Attempting to move forward
Move forward: None
```

Figure 3: Control GUI

3.1 Design Considerations

In order to ensure that the work on the drone could process in parallel with the work on the control interfaces, the main concern was that each component of the system should be able to operate as independently as possible. The drone should not rely on any artifact in the network connections to send or receive information. The GUI should not crash if the drone is unable to connect or if one of the control methods fails. The speech recognition module should quietly proceed if our Google overlords suddenly drop from the sky and cease to exist. Basically, enforce a separation of concerns as much as possible.

Originally, the Observer pattern was used for this purpose. Testing required a mock class that listed for messages and reported what was received. This actually worked rather nicely. Updating a component was mostly confined to defining its update method. The trouble came when building the GUI. It turns out that tKinter does not like it when you communicate with it over a separate thread. At all. Like it really does not like it. So, the Observer pattern was replaced with a poll based approach where updating the GUI requires that the relevant data be pulled from each component on the GUI's update tick. Effectively, this still allows for testing each sub-component individually, although the mock polling mechanism requires a little more effort to set up.

The future plan is to move to an iterator approach, where a component can be consumed to provide its information. Each component is operating over a time-based sequence or in a changing state. Generating the next data point to be consumed would be building a dictionary of the data to report (is the drone flying, the last message sent, the last message received, etc). The consumer would then deal with the information in whatever way it saw fit. In many ways, this is a blend of the Observer pattern and the polling based approach, but it does not violate any of tKinter's desires.

3.2 Control GUI

The main benefit of the control GUI comes in that it is a single place to perform integration testing. It has the added benefit of it's aesthetic appeal, as is clear in figure 3.

The actual GUI consists of only several sub-components: the control methods, the diagnostic information, and the logging mechanism. The control methods can be further broken down into two sub-categories: simple controls and complex controls. The simple controls are those which offer a labeled button to perform that action. e.g. Connect, takeoff, land. The complex controls are those on that would equate to the toggle sticks on a traditional remote control. e.g. Up, down, left, right. While a rather arbitrary distinction, as both types can be controlled by any of the input mechanisms (keyboard, mouse, speech, etc), the complex controls can be extended to operate in a stateful manner. The current, most stable implementation simply sends an rc command to the drone which corresponds to the control selected. However, if state were to be tracked with regards to the accumulation of inputs to these controls, varying levels of speed could be set for each axis of control. In the current configuration of the input methods, this was found to be rather unstable. Partially, this was also due to the fact that the keyboard input currently only detects key-up events as this was the only option provided by the GUI framework in use. This is simple enough to change out, but the effort involved was better spent solving other, larger and more interesting problems. This is left as an exercise to the reader.

The other interesting part of the GUI is the video display. The diagnostic information just shows the current state of the drone. Whoo-hoo. Battery is getting low. Acceleration is downwards because it's on Earth. Useful, but not particularly interesting. The video, however, requires several moving parts to get working. The first is because the drone and the gesture module don't actually communicate directly. Nor do they know about each other in any capacity. Instead, the frames all pass through the GUI, which is acting as the controller in this capacity, from the drone to the gesture recognition module. The controller then performs several other actions and then asks the gesture module for the latest information about the last frame it sent over. It pulls this information back and displays it to the user on the graphic screen, along with some computed metrics including the optical flow vector, which is explained in detail in the next subsection.

3.3 Gesture Estimation

The quick overview of the gesture estimation module is that it consumes a set of frames and computes the sparse optical flow between the set of frames. In essence, this seems to act as if it were operating on a video. In reality, each subsequent frame could come from any source, which would make Tyler Durden quite happy. Frames could theoretically be spliced together from multiple sources in order to influence the computed flow or to direct attention to particular areas of the video. It also has the added benefit of separating a particular method of generating frames from testing the gesture estimation module. In fact, several test videos were recorded from both the drone and the webcam and used to develop the module independently of the GUI or the drone control mechanism.

The module itself computes the optical flow in several steps. The module performs this by going through several steps, the most important of which is computing a set of good feature picels to track at each frame. Many resources exist covering Shi-Tomasi for feature detection and Lukas-Kanade for motion estimation, so those details are omitted here. The rest of the image is masked and the point of interest is drawn on top of the original image, as example of which can be seen in figure 4. The unique portions of my use of these technique is in the restricted attention span of the keypoints and how it uses those points to generate a vector which can be used as an input method. The attention span can best be seen by watching the basic speech and tracking video located in the docs folder in the github repo. If you assume that there exists at least one previous point, then we can compute a vector containing a magnitude and an angle. The angle tells us which direction the drone should fly in, and the magnitude tells us how fast. There are two issues here. The first is that a change between any two frames is very noisy, causing the computed magnitude and angle to vary wildly. This is solved in a two-step process. The first is to take the smoothed average of the past 30 frames and use those vectors to compute the general direction. The second is to define a threshold for the magnitude. The vector was only considered "valid" if it exceeded this threshold, and through empirical testing, it was found that an adequate threshold for stabilizing the vectors was simply *1*. In other words, this value was found to be good enough through good old trial and error.

The second issue was one that did not come up until I was actually presented with it. When using the webcam or Tello videos for testing, the videos were all shot with a stationary camera. The Tello drone was not stable enough to reliably record a video while in motion. The method described above computes the relative motion of the person moving. However, since the reference frame is not moving, all of the motion can be attributed to the person. If the reference frame is moving, as would be the case is the drone is using the computed motion as input to control its motion, then this would have to be

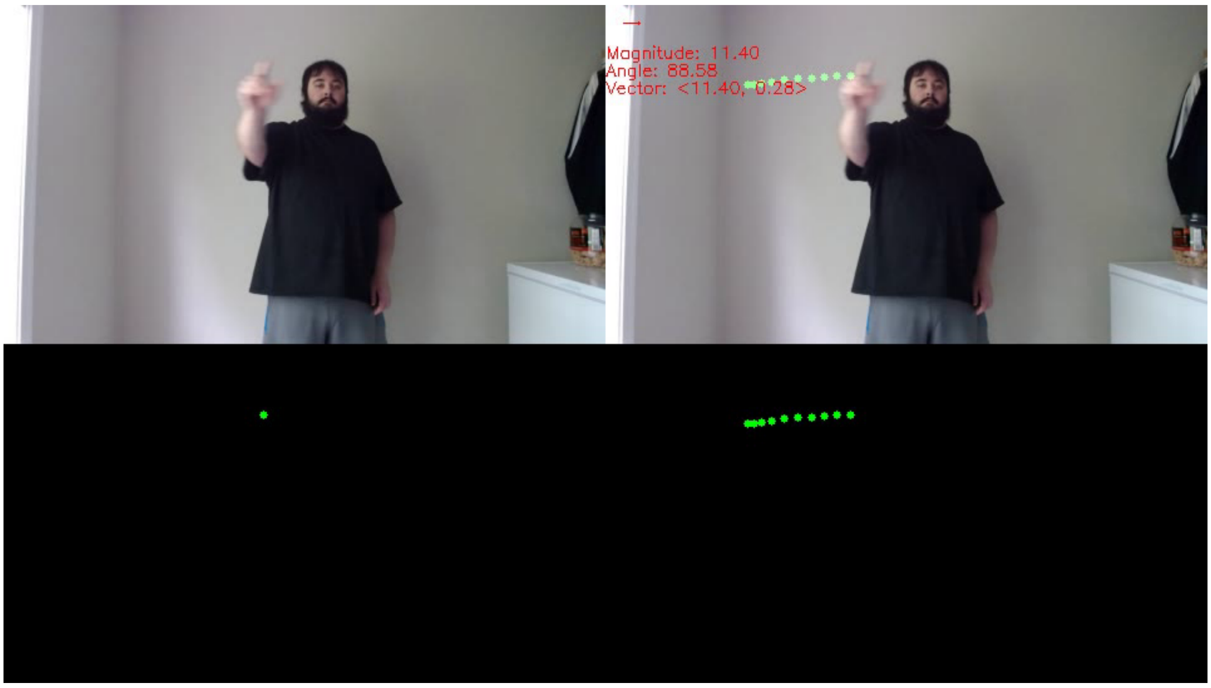


Figure 4: Gesture Estimation Display

taken into account when computing the vector input. This issues arises due to the following sequence of events. The drone computes the motion between two frames. It uses that vector to fly a particular direction (let's assume it flies up). This causes the reference frame to shift down, meaning that the next vector it computes is biased in the downwards direction. Basically, using solely this approach is a self-stabilizing system: changes made to the system are inherently offset by the effect they have on the following timestep.

Given these highlighted issues, the stability of the system is not suitable for use in a real-world system. At least not safely. The drone tended to like to fly into walls and the ceiling. Thus, while the display on the GUI shows the computed vectors, the current implementation does not use these vectors as an input to control the drone.

3.4 Speech Recognition

To get up and running with a simple speech recognition module is surprisingly easy. The package speech_recognition offers a solution that works out of the box, for the most part. However, the current implementation sends a request to Google's API to perform the actual transcribing of the audio byte. There are several offline options provided by the package, but getting Carnegie Melon's module working (which is provided as part of the speech_recognition module), is less than well documented. Thus, for an easy and usable option, using Google to perform the actual heavy lifting proved to be a more than adequate solution. It does offer an interesting conundrum, however. In order to connect to the Tello drone, the host NIC has to connect to the Tello wi-fi network. This means that it loses access to the internet and can't send the request to Google's servers. Fortunately, an easy solution for this is to simply add another network card to the system. Today, this is easily and cheaply done by simply using a USB network adapter. I happened to have several laying around, but as for a long term solution, this is definitely an area in the system that could be improved upon.

While getting the actual functionality of speech-to-text was easier than expected, the actual challenge came in the form of integrating it into the control interface. The module does not do continuous transcribing, but rather separates bytes based on audio "whitespace." I had designed the system to search for particular keywords and then start paying attention to the next few words based on that "heard" keyword. The fact that the speech is already separated simplified the actual task of tying the spoken word to a specific command, but required a redesign of how the control interface communicated with the speech recognition module. In the end, the design came out to be similar to that used by the gesture control module. The speech module stores a history of the transcribed audio bytes, which are typically only one to two words. The control GUI keeps track of which was the last message index to have been pulled, and at each tick, it asks the speech module for the next audio byte. If that byte is a valid command, it performs the specified action, be it controlling the GUI or a command for the drone.

4 Next Steps

In no way is this project in a stable or complete form. A lot of work remains, both with regards to the hardware and software components involved. The below suggestions are potential areas to expand upon the provided functionality, but many other ideas exist as well. A further set of future tasks, both long and short term, exist in the *roadmap.txt* file located in the github repo.

4.1 Hardware

The hardware can take a few different paths, and the path chosen also impacts the software component of the project. At a high-level, the distinction lies in pursuing a solitary agent or bringing in several friends to build a swarm of drones.

4.1.1 Custom Drone

Taking the solitary approach, the first step would be to try and distance myself from the Tello drone. Maybe see how well it can dodge a hammer. But the main issue is with regards to its stability and lack of an effective communication protocol from the drone to the host system. Given less of a time constraint, time should most certainly be spent on this aspect to ensure a stable, robust, and configurable drone. Hopefully for less than the price tag associated with the Tello base model.

4.1.2 Drone Swarm

An interesting take on the problem is to deploy a swarm of drones rather than a single agent. This would still require developing a custom drone, but these would all be much smaller, meaning they would also be less stable in flight. To operate as a swarm, they would require additional sensors compared to the single agent. More sensors means more weight, more power, and a greater computational drain. Sourcing of effective parts would be much more paramount and would likely benefit from development of a simulation environment prior to physically building the physical drone.

4.2 Software

As the hardware evolves, so must the software. Beyond updating the control methods to account for a custom drone, there are better techniques able to be used to address some of the issues faced with the current approach.

4.2.1 Motion Tracking

One of the easier tasks to handle may be to implement various tracking algorithms. OpenCV offers several out of the box, and these could be used to replace the gesture recognition with a higher degree of fidelity. Using optical flow estimation may be able to be expanded to account for an overall energy shift in subsequent frames, allowing for accounting for the shifting reference frame when the drone moves. But even this alteration could benefit from a more accurate and robust tracking mechanism. There's nothing requiring only a single processing technique to give the system the information it requires.

4.2.2 Pose Estimation

Along with tracking algorithms, pose estimation of the target could benefit the system in not only seeing where the target is moving, but where it will likely move in a 3D environment. People often make subtle movements prior to actually moving, and if those movements are able to be tracked and estimated, the drone may be able to anticipate the movement and react accordingly. This would be particularly useful in deploying the drone to help protect you during HvZ. No more getting caught off guard by those pesky zombies.

4.2.3 Monocular Depth Estimation

A large limitation of the current method of preforming the gesture recognition is that it cannot detect depth. In many applications, this is solved through use of using multiple images taken at various angles and computing the stereo depth of the image. The drone would be able to move around to take multiple shots, but this would fail rather spectacularly here. It makes a fatal assumption in that it assumes the target is not moving. It may be possible to use a stereo type approach with a swarm, similar to using a friend to tell you how far back to move the claw in the arcade claw game. But still, it would be preferable to be able to have a single drone perform the depth estimation rather than using multiple drones, meaning additional resources, to perform this same task. This is where monocular depth estimation comes into play. Recent papers show promising results, at least in a lab environment. Transferring it to a real-world application may prove challenging.

A Time Breakdown

Total:	**200**
Literature review	**35**
Drone	**85**
Control Interfaces	**80**

Phase	**Time (hours)**	**Sub-task**
Literature review	20	Drone documentation
	15	Computer vision processing
Drone	45	Communication library
	25	Video streaming
	15	State parsing
Control Interfaces	15	Manual Control
	25	Control GUI
	25	Gesture Recognition
	15	Speech control

Table 1: Time Spent

Phase 1: Literature Review

The initial literature review was not intended to be yield a comprehensive understanding of all the information that would be required in order to accomplish the goals of the project. Rather, it laid the base ground work that outline the general path to be taken and how the path was shaped. In short, it served to set the initial path to get started.

Phase 2: Drone

This phase of the project was focused on the construction of the drone itself. The hardware component was effectively cut short due to the inability to construct a custom drone. However, as previously outlined, using the Tello drone offered its own set of challenges.

Phase 3: Control Methods

Worked on in parallel with phase 2, the software component of the project was focused on developing methods for controlling the drone from a remote system. That is, the control mechanism would communicate with the drone over some network connection. No processing on the drone itself is responsible for controlling its flight.