Autonomous Drone

Joseph Helland, Jesse Whitaker, Patrick Cowan, Scott Glass
Dept. of Electrical and Computer Engineering, University of Utah

***Abstract*—For our senior project we modified a quadcopter so that it can fly autonomously and land whenever a camera detects a red target. The quadcopter uses four infrared sensors to avoid obstacles and an off-board camera to detect targets. Whenever the camera sees a red target, it tells the quadcopter to land. Our project was mostly successful; we were not able to attach the camera to the quadcopter due to I/O issues, but we were able to have it fly around a room while avoiding walls.**

I. Introduction

For our senior project we modified a quadcopter so that it could fly around autonomously and land whenever an off-board camera sees a red target. We used three infrared sensors to avoid obstacles and one to control altitude. We used a small camera module to communicate images to a computer and OpenCV software to analyze the images for the color red. If the image was a match, the computer would send a signal to the quadcopter telling it to land.

The original idea for our senior project was to have the camera on board the copter so that it could identify objects without the help of a computer. Due to I/O issues, we were not able to attach the camera module directly to the copter (see the difficulties section for more explanation on this). However, we were able to demonstrate these two aspects separately. We demonstrated the project by having the quadcopter fly around an empty hallway with the camera module connected to a computer. Whenever we put the color red in front of the camera module, the copter would land successfully. see figure 1 for a diagram of all the project components and how they interact.

The idea for this project came from reading an article about colony collapse disorder (CCD). CCD is a mysterious phenomenon responsible for the deaths of as much as 30 – 90 % of adult honey bees within a colony. The effects of CCD could be economically disastrous because about 130 agricultural plants in the United States are dependent on honeybee pollination (valued at about \$15 billion annually) [1]. We think our autonomous drone could be a great solution for CCD. The central computer could use an algorithm to recognize plants that need pollinating and direct the drone to the targets. CCD is just one example of where an autonomous drone could be useful. Other good examples include finding survivors after a natural disaster, performing reconnaissance in a military situation, or tracking animals for hunters.

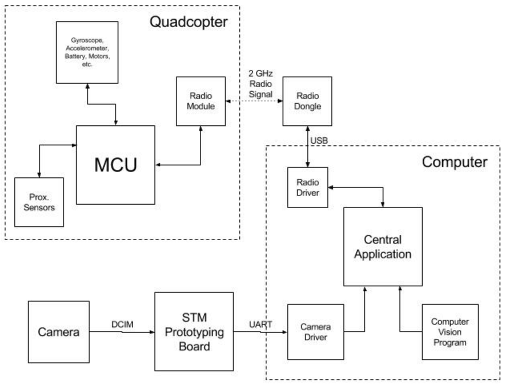


Fig. 1: Overview of project.

II. Background

An autonomous drone is not a particularly new or revolutionary concept. The military has been using drones for the past several decades and some companies even sell drones as toys such as the AR Drone 2.0 by Parrot, the FPV X4 Mini RTF Quadcopter by Hubsan, and the Phantom Aerial UAV Drone Quadcopter by DJI. Even Amazon has been doing research on using drones as a delivery system for packages. Although our project is not completely original, the aspect that sets it apart from all these other drones is scale. To date, no one has implemented a completely autonomous quadcopter as small as the CrazyFlie (which weighs only 27 grams and is as big as the palm of your hand). The closest product we could find to this is called the Zano and weighs 55 grams.

Fig. 2: Picture of the CrazyFlie 2.0 quadcopter [4].

III. Hardware

A. Quadcopter

We briefly considered making our own quadcopter for this project but quickly decided against it. Due to time constraints and the complexity of flight, we thought that making both a quadcopter and making it autonomous would be unrealistic. We even found a senior project from 2013 who attempted to create a quadcopter and it proved to be a huge amount of work [2].

Fortunately, we were able to find a quadcopter that was perfect for our purposes: the CrazyFlie 2.0 by Bitcraze (see Fig. 2). This quadcopter uses both open source hardware and software, making it an optimal platform to host our modifications. The CrazyFlie also provides a number of other useful features such as on-line documentation, a radio module and communication protocol, easy flashing via radio, and some basic flight code. The tiny quadcopter is also surprisingly powerful, with 168MHz Cortex-M4 micro controller (MCU) with 192kb of SRAM and 1Mb of flash memory [3]. The CrazyFlies MCU comes preloaded with some basic flight code. It basically uses stabilization algorithms to point the quadcopter forward and keep it somewhat level. The CrazyFlie also comes with code to interface with the radio and manage power usage.

B. Proximity Sensors

Both infrared and ultrasonic sensors were considered for collision avoidance. With the tiny payload available on the CrazyFlie 2.0, ultrasonic sensors proved to be too heavy. Sharp builds a infrared sensor module that weighs very little and functions as low a 2.7 volts. This sensor

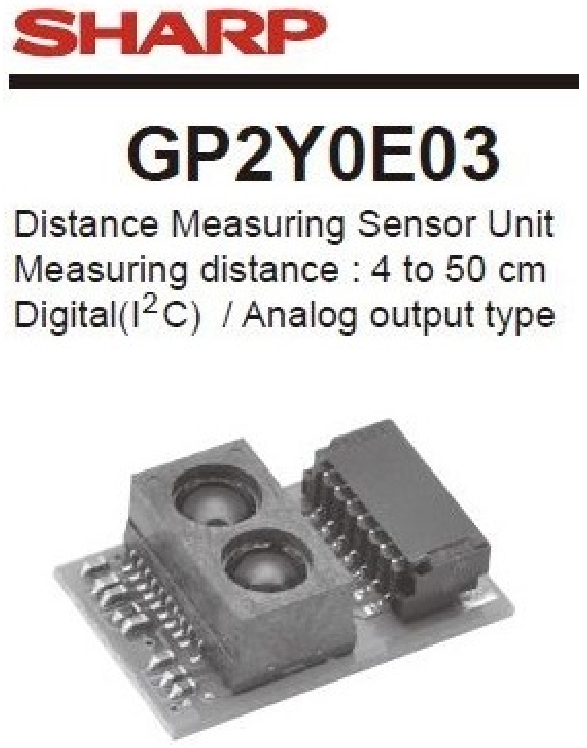

SHARP
GP2Y0E03
Distance Measuring Sensor Unit
Measuring distance : 4 to 50 cm
Digital(I2C) / Analog output type

Fig. 3: Picture infrared module used [5].

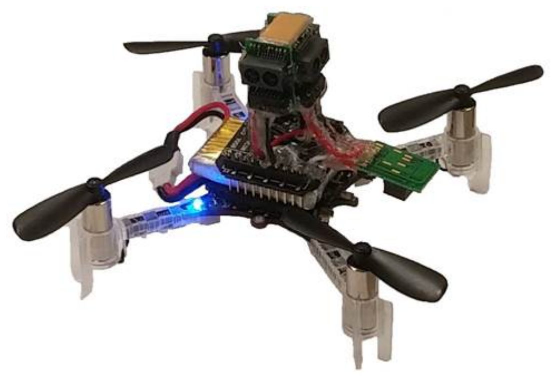


Fig. 4: Picture of the CrazyFlie 2.0 quadcopter with sensors.

was used to show proof of concept with a quick way to build it.

Four of these sensors were added to the CrazyFlie. One facing down to help with altitude hold. One forward, one left, and one right allow avoidance. The flight stability was not as good as originally anticipated and the rear blind spot allowed a lot of collisions. The rear sensor was not included to save I/O for the camera. Many

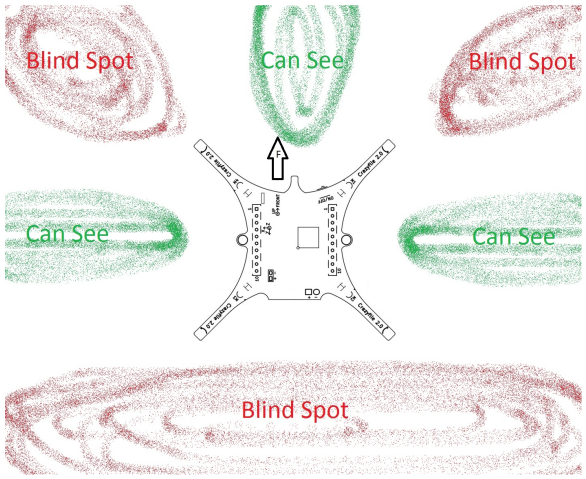


Fig. 5: Sensor blind spots.

collisions also occurred at the corners. Because of the erratic flight, a wider seeing sensor might have performed better and a rear facing sensor is needed (see Fig. 5). The original idea, rotating from a fixed position before making decisions on movement, didn't work. Overall, the goal for the sensors and object avoidance was met, but a custom PCB that incorporates all sensors would save weight and allow a more compact and stable setting for the sensors. There is a lot of room for improvement.

C. Camera

For giving the quadcopter vision, we used the OV9655 image sensor. WaveShare manufactures a prototype board for this sensor that we used to interface with the camera. The main advantages of this camera is it is built for low-power mobile applications. The camera operates on between 1.7 and 3.3 volts and the camera is only a few centimeters across [6]. The largest disadvantages of this camera is the IO that is required to operate it. Aside from the lines providing the necessary voltage, the camera has two lines for the serial camera control bus (SCCB) interface, three lines for timing information, one line for an external clock that must be no less than 10 MHz, and an additional 8 lines for the actual pixel data. The camera requires an 14 additional IO lines to communicate with the camera. To solve this IO issue, we used an additional MCU to interface with the camera.

D. Prototyping Board

For developing the interface between the camera and the quadcopter, we decided to use the STM32F407VG

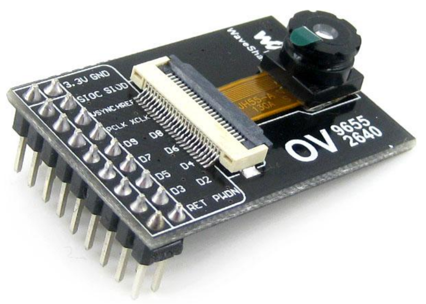


Fig. 6: OV9655 sensor on WaveShare board [7].

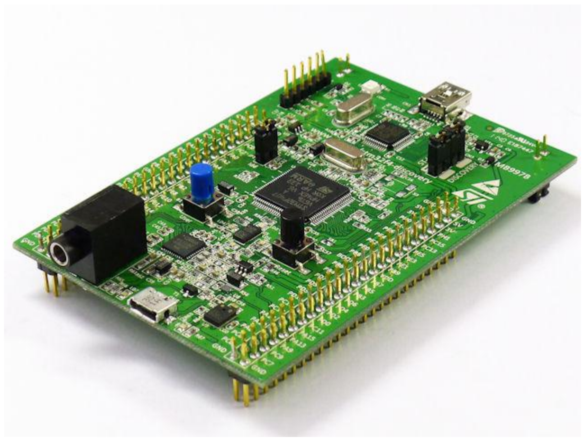


Fig. 7: STM32F407VG Discovery Board [8].

Discovery Board. This development board uses a 407VG cortex M4 processor which is in the same family as the processor on the quadcopter. However, unlike the 405 processor on the quadcopter, the 407VG has a digital camera interface (DCMI) module that allows us to capture image data without processor intervention. The 407VG also contains 192 Kbytes of RAM, allowing it to store images without the use of external memory. This prototyping board also has I2C and USART support, allowing it to communicate with both the camera and computer that controls when the quadcopter lands.

IV. Interfaces

All of the proximity sensors communicate with the MCU via analog to digital converters (ADC). The camera needs a total of 14 IO pins to operate. two of them control the camera itself, doing functions such as telling it when to take a picture. The other 12 IO pins are for transferring data. Due to how many IO pins this camera requires, we were unable to attach it directly to the quadcopter. Instead, we had the camera communicate with an STMicroelectronics discovery board which then communicated with the central computer. The camera communicates with the discovery board using two protocols: a proprietary SCCB format for setting up the cameras registers and the digital camera image (DCIM) format for transferring pixel data. The discovery board MCU will periodically transfers camera data to the central computer application through a UART interface. The central computer issues commands to the quadcopter using a 2 GHz USB radio dongle and a custom radio protocol developed by BitCraze.

V. Software

A. Collision Avoidance

We tried many different approaches to writing the collision avoidance firmware, but what worked best is the function shown in figure 8. This function takes in a proximity sensor reading as a parameter and returns a desired pitch or roll value. The x-axis shows the value returned from the proximity sensor; high numbers indicate the object is close. The y-axis shows what we want to set our pitch to; negative pitch indicates the the copter is tilted forward. If the copter does not see anything, it very slightly tilts forward at 7.5 degrees. As the copter approaches an obstacle, it slowly evens out and when it reaches about 1300, our collision threshold, the copter should be completely level. Once the copter moves past the collision threshold, it reacts extremely quickly due to the quadratic nature of the curve.

The biggest disadvantage of the approach shown in figure 8 is the quadcopters tendency to overcompensate for collisions. What would happen is the quadcopter avoids the walls just fine, but it would go so far backwards that it would run into the opposite wall backwards. We tried to compensate for this by incorporating a velocity component into the code. We would estimate velocity by polling the proximity sensors every .05 seconds and subtracting the previous reading from the current reading. If we detected that the last several velocity calculations were negative and above a certain speed, we would set the pitch forward to try to compensate. This approach had mixed success. It seemed like it helped some of the time, but the velocity readings were not consistent enough to help eliminate the overcompensation problem entirely.

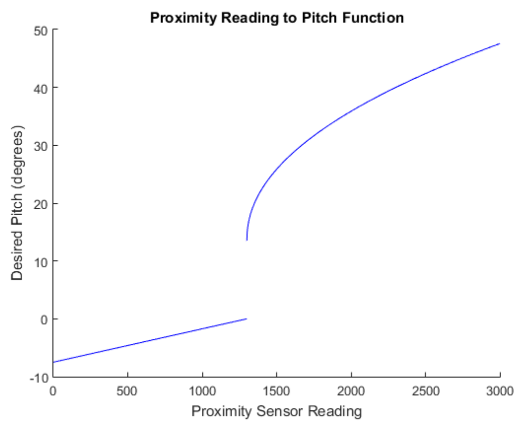


Fig. 8: Function mapping proximity values to its corresponding thrust.

B. Thrust Control

The quadcopters thrust control works using the state machine shown in figure 9. The quadcopter starts in the liftoff state, the point of which is to simply get off the ground. Thrust is set to a constant value for one second, after which it hands off to the steady state. The purpose of the steady state is to try to keep the copter at a constant altitude. The code to do this is surprisingly simple. What the code does is query the downward facing proximity sensor for a reading. If the reading is above a certain threshold, we subtract a bit from the current thrust. If we are below the threshold, we add a little bit to the current thrust. This scheme forms a feedback loop in which the copter is constantly adjusting thrust to try to reach the threshold value. A huge advantage of this approach is that the quadcopter automatically adjusts for different battery levels. Once the camera has spotted a target, the copter enters the landing phase. The landing phase is very similar to the liftoff phase, except the thrust is set to a much lower value. Once the copter has set on the ground for about three seconds, it goes back into the takeoff phase and the cycle repeats.

The thrust control program also helps a bit with the collision avoidance. If the sensors detect a collision and we are tilted away from the obstacle, we add some thrust to compensate. This gives the copter extra power when it tries to move away from a wall.

C. Client Modifications

We did not make many modifications to the client. In the GUI in figure 10, the changes we made were adding

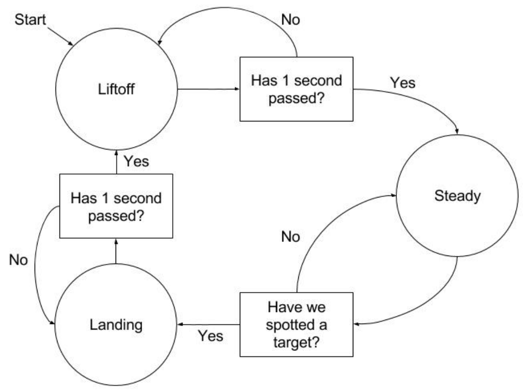
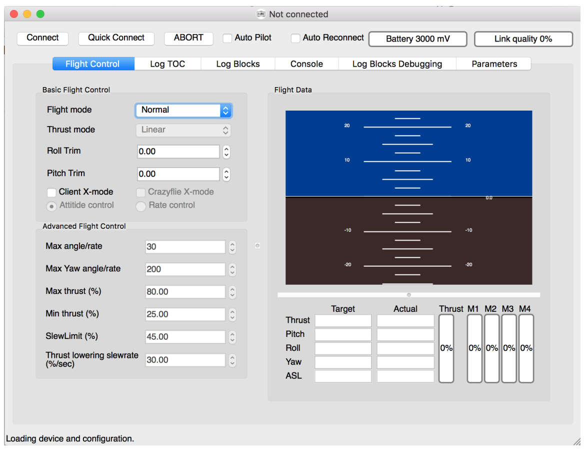


Fig. 9: Diagram of the thrust control scheme.



Flight mode: Normal
Thrust mode: Linear
Roll Trim: 0.00
Pitch Trim: 0.00
Client X-mode
Crazyflie X-mode
Attitide control
Rate control

Max angle/rate: 30
Max Yaw angle/rate: 200
Max thrust (%): 80.00
Min thrust (%): 25.00
SlewLimit (%): 45.00
Thrust lowering slewrate (%/sec): 30.00

	Target	Actual	Thrust	M1	M2	M3	M4
Thrust			0%	0%	0%	0%	0%
Pitch							
Roll							
Yaw							
ASL							

Loading device and configuration.

Fig. 10: Diagram of the thrust control scheme.

the ABORT button and the Auto Pilot check box at the top of the screen. We also added a thread to the client that constantly runs the camera and queries the computer vision program. If the computer vision program reports a match, we send a landing signal to the quadcopter.

D. Camera Firmware

We built the camera firmware using multiple open source projects as a starting point. Through experimentation and testing, we found the only reliable way to capture camera data was through the DCMI. All other attempts at interfacing with the camera through GPIO would fail due to varying reasons. Our final firmware implementation allowed the computer to send a message over a virtual serial port to notify the development board to capture another image from the camera. After the DCMI finishes capturing a new image from the camera, it uses the virtual serial port to transmit the entire image buffer back to the computer for decoding and processing.

E. Camera Software

We built camera software on the main computer to handle the encoded image data that was sent back from the prototyping board. The application is written in python and utilized the TK GUI framework built into python. We also used 3rd-party libraries for serial communication and image processing. The application is capable of communicating with the prototyping board to capture encoded image data. The application will then decode and save the image before displaying it in the UI.

F. Computer Vision

The computer vision portion of our program was written using an open source library called OpenCV. The program was written in java and was run from the command line. It takes two arguments: the picture we want analyzed and a template of the target we want. The program would use the OpenCV libraries to compare the template image to every part of the camera image and find the portion of the image that matches the template best. OpenCV returns a number that indicates how good of a match it has found. If the returned number is above a certain threshold, then we would print a 1 to the console indicating the image is a match. If the image was not a match it prints a 0.

VI. Difficulties

It turns out that flight, quadcopters, and cameras are more complicated than we expected! This section describes what went wrong when developing this project and how compensated.

- Stability - The biggest problem developing for such a tiny quadcopter is stability. It is so small that whenever the quadcopter slightly tilts it builds up a ton of speed very fast. We compensated for this by having the quadcopter react very severely whenever a collision is detected. We had decent success doing this, but it is unlikely we could get it working perfectly without more sensors or a bigger quadcopter.

- Proximity sensor inaccuracies - Every once in awhile the proximity sensors would produce strangely inaccurate readings. We compensated for this by taking the average of multiple sensor readings and removing any outliers.

- Battery level - The quadcopter is surprisingly sensitive to battery level. After about 30 seconds of flight, the quadcopter becomes so inaccurate that it is basically unusable. We tried various schemes to try to account for this but none were successful. For this reason we restricted our demonstrations to about 30 seconds before switching and recharging batteries.
- Gyroscope glitches - Every once in awhile the quadcopters gyroscope would become stuck in a state where it thinks that up is down and down is up. If this happens during flight, it causes the quadcopter to do a flip. We never did find the reason for this, but thankfully it happened rarely enough that we could ignore it.
- Client freezes - Sometimes when we tried to control the quadcopter the client application would become completely unresponsive. We could connect fine and we even got battery logging information, but the quadcopter would not respond to any of our commands. This problem happened rarely when we first started developing for this project, but it happened more frequently the closer we got to demo day. About a week before demo day, it got so bad that only about one in fifteen connections to the quadcopter were successful. We suspected it might have something to do with the laptop we ran the client from, so we switched over to a different one. This seemed to fix it at first, but the problem popped up again a few times since then. We still have not found the cause for this.
- Inconsistent thrust - Nine out of ten times the thrust control works perfectly, but occasionally the thrust jumps way too high and the quadcopter flies straight into the ceiling. This problem is probably related to the battery issue described above. We compensated for this by making the steady state adjustment lopsided. If the quadcopter is below the target altitude, it only adds a little to the thrust, but if the quadcopter is above the target altitude, it subtracts a lot from the thrust. This makes it so the copter occasionally dips below the target altitude, but it almost never goes above it.
- Balance - The quadcopters performance is hugely affected by how the battery is positioned on copter. If it is slightly too far forward or backwards it crashes into the walls. We compensated for this mostly through trial and error. If a run goes really badly we would keep adjusting the battery position until it seems normal.
- Sensor blind spots - We developed this quadcopter assuming that we would have camera on board. As a result, we used the minimum amount of GPIO for the proximity sensors. Because of this, our quadcopter has several blind spots mainly in the corners and on the back. These blind spots used to be one of the biggest causes for crashes. We compensated for this by programming the quadcopter to slowly turn as it is flying. The rationale for this is that it minimizes the amount of time any blind spot exists. In other words, the blind spot changes position through turning before it has a chance to become a problem. The solution was very effective, as it seemed to greatly reduce the number of crashes due to blind spots.
- Camera interfacing - We encountered numerous issues when building our camera interface. Throughout the semester, we attempted to interface with the camera over GPIO since our original goal was to put the camera on quadcopter. All of our attempts to use the GPIO directly to drive the camera would end with an issue that could not be solved. Typically, these issues would be caused by lack of IO. One example would be that we needed to generate a clock for driving the camera at a proper speed; however, due to the IO constraints of the quadcopter, we could not do this and keep the I2C interface needed to communicate with the camera. Ultimately , we scrapped the idea of putting the camera on the quadcopter and instead opted to use the DCMI on the discovery board.
- Image data corruption - After solving the interfacing issue, we ran into a very strange image corruption issue. When the camera captured images with too much color, the pixel data would suddenly become misaligned. After a large amount of troubleshooting various portions of the camera interface, we found the issue was being caused by our python application. For debugging purposes, we wrote the encoded serial data out

to hex file and compared the image information to data captured using an oscilloscope. At some point when python writes the image data out and reads it back in to decode the image, it becomes corrupted. Removing the file IO code from the python application solved the issue.

- Computer Vision Integration - When working to combine all of the teams work into one cohesive project, we ran into issues with the OpenCV libraries. Since OpenCV uses its own optimized version of the C++ libraries, we encountered issues getting the project to link on any machine other than the machine the application was developed on. To solve this issue, we ultimately used the Python Imaging Library to replicate most of the functionality. Unfortunately, this process lost some of the more sophisticated algorithms that were being used in the Java application; however, we were able to detect colors and properly communicate with the quadcopter to tell it to land, allowing us to demonstrate a functioning project.

VII. Conclusion

Our project works fairly well all things considered. The biggest problem is battery life; we can only let it fly for maybe 30 to 40 seconds before it starts running into walls. Flight is so complicated that it is extremely unlikely we could get it working perfectly without more sensors and way more development time. Despite our difficulties, it was still a pretty fun project to work on. We crashed the quadcopter probably more than 200 times during development, so it is surprising the thing still flies at all. There is plenty of room for improvement, but we are happy with what we accomplished.

Acknowledgments

Much thanks to Ken Stevens for his help throughout the semester. Thanks to John M. Hollerbach and Daman Bareiss for helping us get access to the Large Robotics Lab. Thanks to Jon Davies for giving us advice and extra components.

References

[1] Chelsea Gifford, “Colony Collapse Disorder, The Vanishing Honeybee (Apis Mellifera),” Ph.D. dissertation, University of Colorado at Boulder, 2011.

[2] Leif Andersen, Daniel Blakemore, Jon Parker, “Project Levitate,” Dec. 2012, University of Utah Senior Project Report.

[3] *Crazyflie 2.0 hardware specification*, Bitcraze, 2014. [Online]. Available: http://wiki.bitcraze.se/projects:crazyflie2:hardware:specification

[4] *CrazyFlie 2.0 Product Description*, Seeed, 2014. [Online]. Available: http://www.seeedstudio.com/depot/Crazyflie-20-p-2103.html

[5] Sharp, *Sharp GP2Y0E03 Datasheet.*

[6] *OV9655 Product Specification*, OmniVision, 2015. [Online]. Available: http://electricstuff.co.uk/OV9655-datasheet-annotated.pdf

[7] *WaveShare OV9655 Camera Board*, WaveShare, 2015. [Online]. Available: http://www.waveshare.com/ov9655-camera-board.htm

[8] EleckTrics, *STM32F4 Discovery Board*, 2015. [Online]. Available: http://www.elektricks.net/Wp-Content/Uploads/2015/02/STM32F4-Discovery.jpg

[9] P. Cowan, *University of Utah Project Deathray*, May 2015. [Online]. Available: http://deathray.patcowan.com