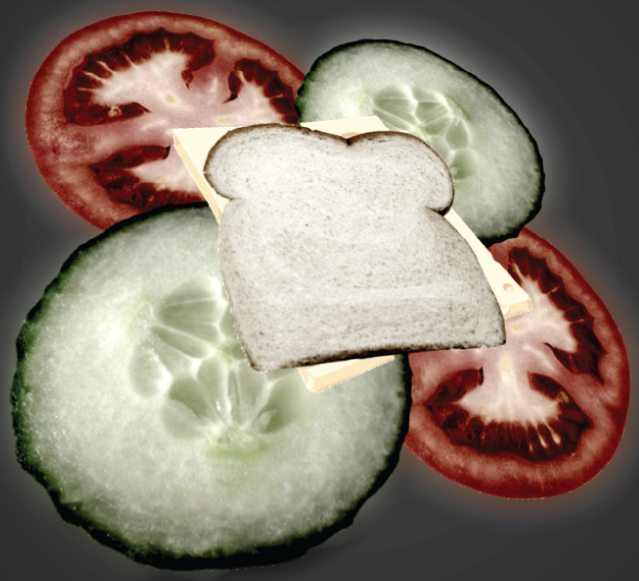
Adam Dickin | James Thorne | Rami Abou Ghanem | Jennifer Patterson

PROJECT SUMMARY

TEAM LUNCH TECHNICAL REPORT #2

TEAM LUNCH
AUTONOMOUS UAV SYSTEMS



ENEL 583/589 Team #30
http://teamlunch.github.com

Prepared on March 25th, 2013 for
Lockheed Martin Canada CDL Systems

TABLE OF CONTENTS

1 INTRODUCTION

Over the past few years, there has been a substantial increase in drones being used for military and police operations. Many of these drones do not have any system in place for collision detection and avoidance besides notifying the operator so that they can change course. This system works well when doing missions in a large open environment, but is insufficient for small, highly maneuverable drones in enclosed or cluttered spaces.

Operator reaction times are too slow to successfully avoid collision, especially when considering the latency introduced by a wireless control harness. Additionally, many of the commercial systems available do not focus on being affordable, but rather on including every feature possible. This results in a drone that is equipped for many tasks, but costs $50,000 or more. This pricing makes drone adoption difficult for casual users, and leaves little room for repair or replacement of drones involved in collisions.

There has been some improvement in this area over the past few years. In fact, some UAVs are now available to the general consumer for around $400. These drones also do not have any kind of collision avoidance, and limited control systems, and are therefore as easy to crash as their high-end counterparts. Some improvement must be made in this area for both the consumer market and the commercial market; ideally, both users would be able to use their drones safely and effectively knowing that systems are in place to avoid crashes.

Our goal as Team Lunch is to solve these problems, and to prove that every autonomous drone can be equipped with our solution. Our project will benefit our customer, Lockheed Martin Canada CDL Systems Ltd., by demonstrating how such a system can be implemented cheaply with off-the-shelf technology available to the general consumer. Our project will also benefit the rest of the UAV market, because we will prove that such features are possible at a low cost. We hope that with time, they will become standard on every UAV. Such an adoption will greatly increase the safety of drones, not only by reducing the risk of a costly crash, but also by reducing the risk of injury to persons and damage to property.

We have decided to focus on taking a commercially available, off the shelf consumer UAV, and equipping it with an autonomous collision detection and avoidance system. This system will allow us to prove that automated accident avoidance is possible during operation of a drone, with minimal operator skill required. Our drone, once outfitted with these systems, will allow unskilled operators to purchase and fly the UAV in close-quarter environments with minimal risk, training, and possibility of damage to the UAV and surrounding property. Hopefully, the outcome of our project will convince drone manufacturers to adopt the use of a collision avoidance and detection system in production model aircraft. We also hope that with the increased availability and lower cost of these autonomous drones, many more organizations will decide to use drones in their everyday operations.

2 LOW LEVEL DESIGN

2.1 OVERVIEW OF SYSTEM COMPONENTS

Our system consists of four major components: the end user, the Android application they use to control the quadcopter, the embedded collision avoidance autopilot software, and the Quadcopter hardware (including our enhanced sensor platform). The following diagram that shows the interaction between these components:

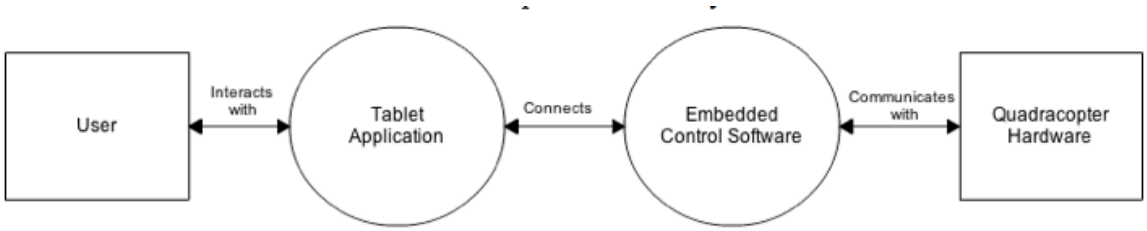


Figure 1: System Interaction Diagram

The system we have proposed is built from off-the-shelf components, in order to allow us to avoid spending time and energy recreating existing technology, such as the quadcopter and ultrasonic sensors. We are building our collision avoidance software on top of a Parrot AR.Drone 2.0 quadcopter, which already ships with control software that maintains all the onboard systems, and allows for easy and stable flight. The quadcopter employs a built-in, lightweight Linux-based operating system, which it uses to run the autopilot software as well as a Wi-Fi wireless networking interface. Off the shelf, the autopilot software is responsible for responding to operator commands (such as "fly left, 20% speed") by adjusting the rotors' velocity. The onboard autopilot also maintains altitude using an onboard altimeter, and can "hold position" using computer vision analysis of the image from a downward-facing camera.

To support our collision avoidance system, we have purchased several range sensors from a Calgary-area vendor called Phidgets. These sensors are ultrasonic, and provide us with distance readings between 154 mm and 6.5 m. These sensors work out of the box, and require no user setup - we simply connect the "+5V", "Ground" and "Data" lines to a Phidgets USB sensor controller, and read the reported distances from a simple C API.

Lastly, we will be using an Android Nexus 7 tablet with an open source tablet application that we will create to fit our needs. This tablet was selected as it is a stable, basic Android tablet, and is representative of the wide variety of Android tablets that could potentially host our control software in the future.

2.2 MODIFIED OPEN-SOURCE ANDROID PILOTING APPLICATION

The Android application for this project will be built off of an existing open source app called "FreeFlight2", created by Parrot. This application is a fully functional app that can be used to fly an unmodified AR Drone. The application will be modified and only the essential features needed for flight will be kept. Along with these modifications we plan to add several features that are specific to our modified AR Drone. The existing application is quite complicated in the way that it has been built but this has been for a good reason, since the developers at Parrot did not want to write separate code for every deployment target. The Android application specifically uses the AR Drone's flight API, which has been coded in C in order to be cross platform compatible. In order to compile this code for Android, the NDK (Native Development Kit) provided by Google must be used. The GUI orientated part of the application however is written in Java and is built off of the Android SDK. As it would not be wise to mess with the essential flight code of the drone which has likely been tested by parrot to be free of major bugs we will implement the core of our new features in java.

We intend to add three features to the software:

1. Upon connecting to drone, the user will be able to remotely launch the quadcopter's onboard autonomous collision avoidance software.
2. Addition of sound to alert the operator when a collision has been avoided, or the onboard autonomous collision avoidance software is proactively limiting the quadcopter's speed
3. A video overlay with invisible bars around the edge of the screen. These invisible bars would change color to non-intrusively warn the operator when the autonomous collision avoidance software is proactively limiting the quadcopter's speed, or the quadcopter is in danger of a collision.

2.3 ONBOARD COLLISION AVOIDANCE AUTOPILOT SOFTWARE

The onboard collision avoidance software is responsible for preventing the operator from crashing the vehicle. This will be accomplished by intercepting operator commands, modifying them based on the ultrasonic sensor data, and then retransmitting them to the OEM autopilot software that ships with the AR.Drone.

The autopilot software is written in C++ and cross-compiled to ARM from a standardized Debian virtual machine image. This allows us to work on the code concurrently, using modern development tools, and write scripts for tasks such as compiling and deploying the software image.

2.3.1 Command Interception and Modification

The Parrot AR.Drone is operated by sending a series of flight commands over Wi-Fi via UDP packets. Our autopilot software intercepts these via a Linux iptables firewall rule, which redirects these commands to a port monitored by our software. Once the commands are received, we decode them, modify them, and then re-transmit them to the onboard autopilot software. The iptables rule is installed only on the wireless network interface, not the local loopback interface, preventing the commands from getting stuck in an infinite loop.

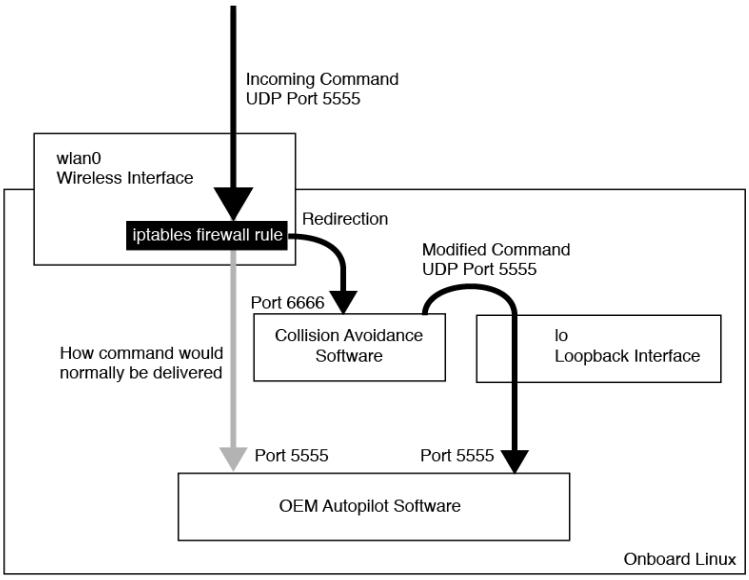


Figure 2: Command Path Diagram

The AR.Drone communication protocol is relatively straightforward. For flight commands, a UDP packet is sent with the following format:

```
AT*PCMD=%d,%d,%d,%d,%d,%d<CR>
```

The six values are as follows:

1. The packet sequence number
2. The current flight mode (hover, fly, fly with magnetic control, etc)
3. Drone left/right tilt
4. Drone front/back tilt
5. Drone vertical speed
6. Drone angular speed (spin)

Our collision avoidance software will intercept these commands, and modify flags 3 and 4 (left/right tilt and front/back tilt) based on the collision avoidance algorithm described in the next section. Once new values are computed, the command packet is recreated, and forwarded to the OEM autopilot. As far as the OEM autopilot is concerned, the operator original operator just happened to avoid the wall. This highly decoupled design allows us to ensure we don't need to modify the existing autopilot's flight, stability, ground tracking, or safety algorithms.

2.3.2 Collision Avoidance Algorithm

The collision avoidance algorithm used is a simple dynamic limiting function, implemented against smoothed sensor data. As the quadcopter approaches an obstacle, its maximum forward speed is limited based on its distance to the object. Once the quadcopter gets within a predetermined "danger" radius, the maximum forward speed is reduced below 0; i.e. the maximum forward speed is actually backwards. This causes the quadcopter to back away from the obstacle, preventing a collision.

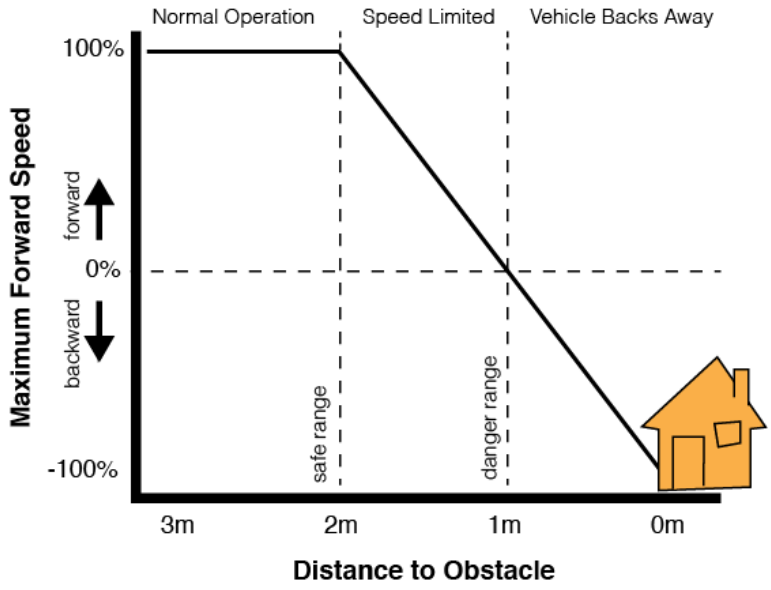


Figure 3: Quadcopter Speed versus Obstacle Distance for One Obstacle

The maximum reverse velocity is limited in the same way, based on data from the rear sensor. In the case where the maximum velocities overlap, such as if the quadcopter is constrained between two walls, the average of the two maximums is used. This allows the vehicle to automatically center itself in a crowded environment, without needing operator input.

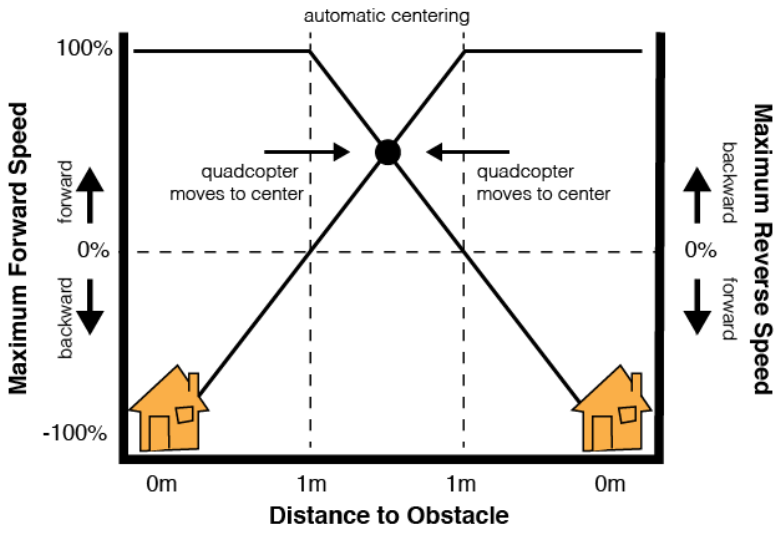


Figure 4: Quadcopter Speed versus Obstacle Distance for Multiple Obstacles

Once the front/back tilt is calculated, the same algorithm is used to calculate the left/right tilt. This allows the vehicle to avoid collisions along both of its lateral degrees of freedom. The OEM autopilot software already maintains altitude above the ground, and ceilings or other obstacles above the vehicle have been determined to be outside the scope of this project.

2.3.3 Code Structure and Design

The code is structured using a dependency injection (DI) technique, allowing us to separate the concerns of various modules, without coupling them more than is necessary. Any unrelated modules communicate through C++ interfaces (classes consisting of only pure virtual methods), allowing us to limit their coupling, and work on different modules independently. For example, the sensor data is only accessible through the I_SensorReader interface, which consists of a single "reading()" accessor function.

A complete diagram of the classes is as follows:

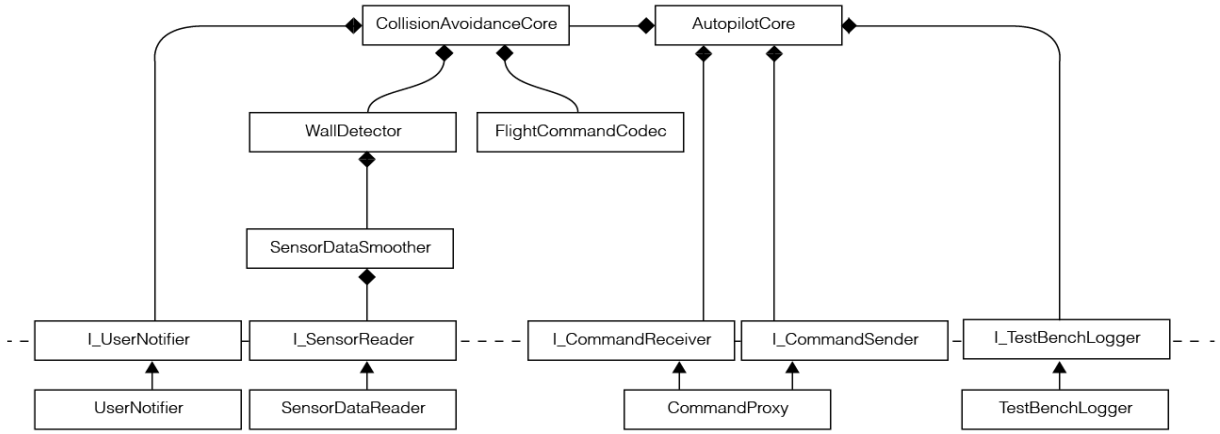


Figure 5: Class Diagram

Dataflow through the software mirrors the class design. Commands are read in from the UDP socket, modified based on the smoothed sensor data, and then written back out to another UDP socket, as in the following diagram:

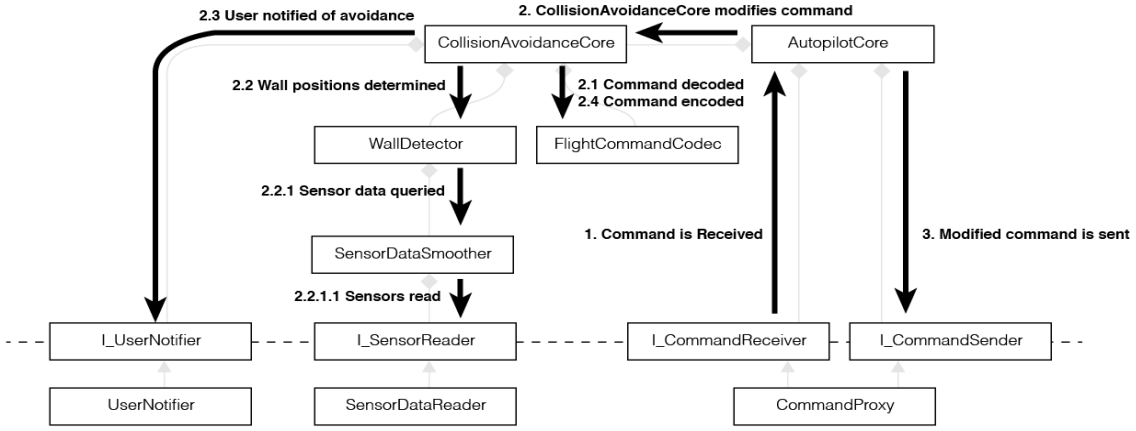


Figure 6: Data Flow Diagram

2.4 HARDWARE DESIGN

We will use the original AR Drone hull and attach ultrasonic sensors to its outer edges. The sensors will communicate to the supplied Phidget Interface kit and communicate distance information to the tablet. The tablet will interpret this information and test for potential collisions.

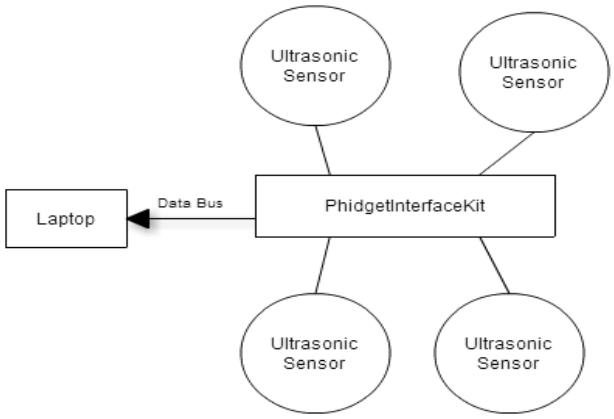


Figure 7: Sensor Connection Layout

Our initial hardware configuration had an ultrasonic sensor on each of the 4 edges of the frame. After initial testing we found that this was insufficient, giving us a blind spot at corners of about 10 degrees, as shown in the following diagram.

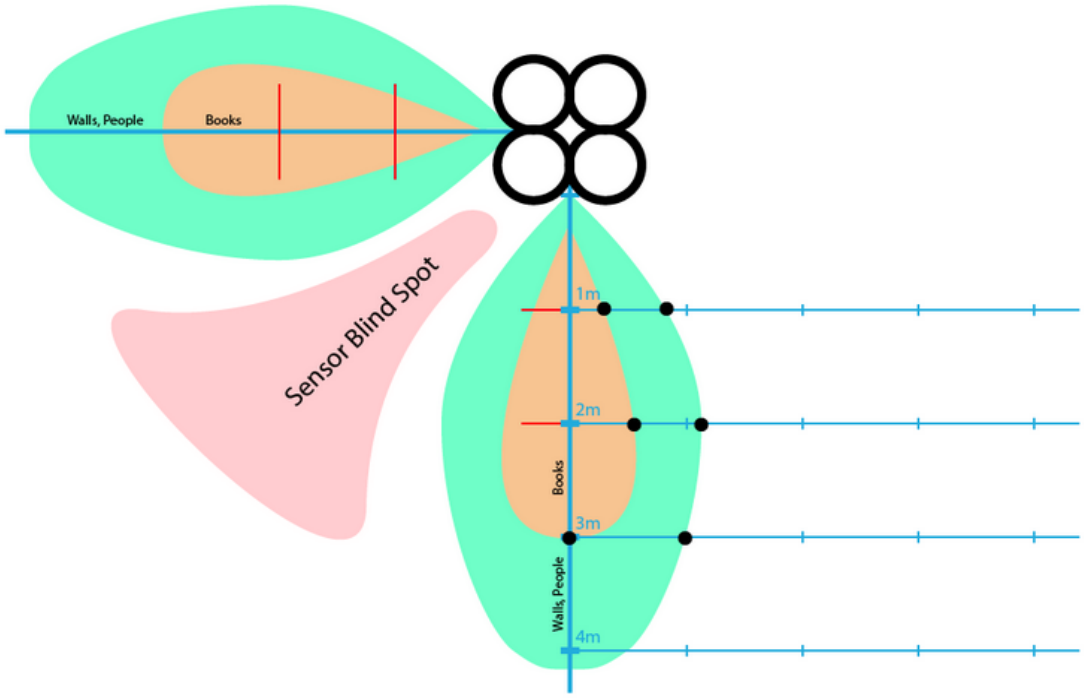


Figure 8: Former Layout with Ultrasonic Blindspots

To resolve this issue, we are adding additional ultrasonic sensors to cover the blind spot, allowing us to satisfactorily detect threats from all directions. Note that since the sensors have a conical zone, we will also detect obstacles from above and below. The new sensor arrangement is shown in the following diagram.

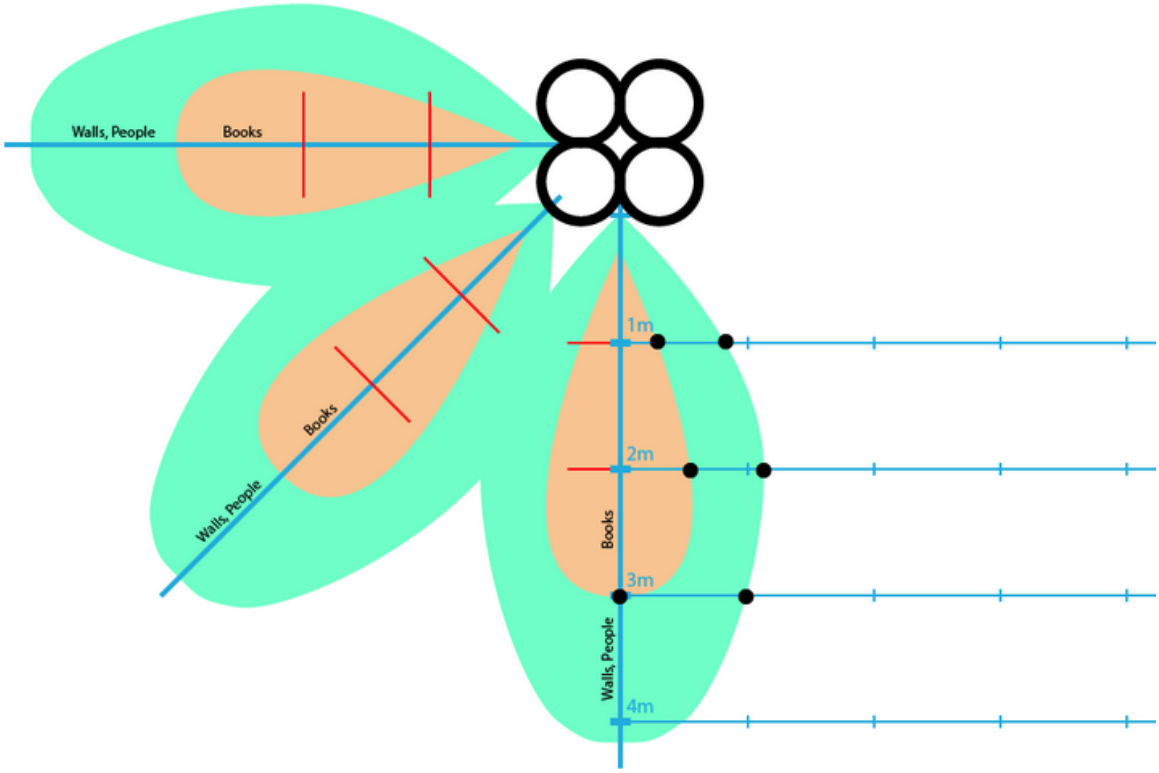


Figure 9: Updated Layout With Full Ultrasonic Coverage

3 PRODUCT DESIGN SPECIFICATIONS

3.1 EXPECTED PERFORMANCES

The product should allow the UAV in use to avoid obstacles in flight. An obstacle is defined as any solid object within the UAV's trajectory. Upon detecting an obstacle, the system will notify the user and either stop the UAV or move away from the obstacle, as required. As of this iteration, the UAV will not be able to reliably avoid wires or meshes, such as chain link fences and nets. The onboard equipment should be able to handle North American outdoor temperatures.

3.2 SPECIFICATIONS RELATED TO PERFORMANCE

The Parrot AR.Drone itself is capable of a speed of 5 m/s, or 18 km/h. It has a weight of 0.9 pounds with the outer hull attached. Flying time on a single battery is about 12 minutes. Based on this speed, we expect the UAV to be able to avoid obstacles which are stationary, or which approach at a speed up to 5 m/s.

According to manufacturer specifications of the sensors, we are capable of sensing obstacles as close as 152.4 mm, although items closer than this will also register as 152.4 mm by the sensor. The sensor can detect obstacles up to 6.5 meters away. Distance measurement resolution is 25.4 mm.

Feeling skeptical of the manufacturer's claims for the ultrasonic sensors, we collected our own distance detection data. In particular, we were interested in finding the angle of the field of view of one of the sensors. Our test procedure is outlined in the Hardware Test Plan section. We found that the ultrasound sensors have a field of view of 40 degrees. This is just short of the 45 degrees we hoped for. Additional ultrasonic sensors will be added on to the system to cover the blind spots. We also found that the size of the obstacle had an effect on its maximum distance for detection. Narrow objects, such as poles and large wires were detectable at 120 cm. Small obstacles of approximately 1 square foot were detectable at 300 cm. Larger barriers of approximately 2 square meters were detectable at 350 cm. Finally, walls were detectable at 600 cm. Since measured results from the sensors were not as effective as the manufacturer claims, the measured values will be our expected specifications.

Measured system response latency was approximately 0.01 seconds. This latency can be considered as negligible, since if an object approached fast enough to collide with the UAV before it has time to respond, the UAV could not possibly have flown out of the way in time.

Finally, the sensor and equipment manufacturer specifications allow the product to be used in temperatures ranging from -40 to 65 degrees Celsius. This is an acceptable outdoor range for most places on Earth.

4 TESTING AND REFINEMENT

4.1 ANDROID APPLICATION FINAL TESTING PLAN

The Android application will be tested with the following methods:

1. Drone will be started along with the Android application, using feature addition 1 we will load the collision avoidance system onto the drone. The drone will then be flown towards an object using the modified Android application.
2. The drone while flying will be flown at its minimum speed towards an object. As the drone enters the specified collision warning zone the red bars laid out around the screen will fade in red as the drone gets closer to the object.
3. The drone while flying will be flown at its minimum speed towards an object. As the drone enters the collision zones tones will be listened for at increasing volume as we head towards the object.

4.1.1 Android Application Testing Results

See Section 4.4 for Integration test results, which cover the Android application testing plan.

4.2 ONBOARD AUTOPILOT SOFTWARE FINAL TESTING PLAN

The autopilot software was built using an Automated Acceptance Test Driven Development (AATDD) approach originally developed by our sponsor, Lockheed Martin Canada CDL Systems. This approach is similar to TDD, but on a higher level. Before any functionality is implemented, an automated acceptance test is written that tests this functionality, by only stubbing out external parts of the software. For our testing strategy, the following software components are replaced with mock objects for testing:

- **CommandProxy**, which wraps the UDP send/receive sockets used to communicate with the control tablet and the OEM autopilot
- **SensorReader**, which wraps the libphidget library used to communicate with our sensors
- **TestBenchLogger**, which wraps the console used to communicate debugging data to the operator

A helper class called "FakeArDrone" constructs these three mock objects, as well as the entire collision avoidance core system. This allows unit tests to easily create and destroy the C++ objects, without having to create them individually.

Once the FakeArDrone has been created, a human-readable acceptance test can be created. For example, a very basic acceptance test that ensures the AR.Drone utilizes data from the front-left sensor might read as follows:

```
TEST(UsesClosestForwardSensorReadingForCollisionAvoidance)
{
   // Arrange
   FakeArDrone::setup();
```

```
    placeWallAtFrontLeftOfVehicle(DANGER_DISTANCE);

    // Act
    sendCommand(flyForward());

    // Assert
    CHECK_EQUAL(proxiedCommand(), stop());
}
```

This test consists of three parts:

- **Arrange**, in which we ask the singleton FakeArDrone class to setup a new instance, and then place a wall in range of the front-left sensor.
- **Act**, in which we order the drone to fly forward
- **Assert**, in which we ensure that the proxied command, which is the one sent to the OEM autopilot, is actually a "stop" command due to the obstacle.

These tests are written using a number of "helper" functions designed to make them easily readable. This allows our customer to review our test coverage, and understand what we are testing, without having to understand our entire system architecture. These functions are generally short and concise; for example, placeWallAtFrontOfVehicle(double distance) could read as follows:

```
void MessageHelpers::placeWallAtFrontLeftOfVehicle(float distance)
{
      arDrone().sensors().readings_[ForwardLeftSensor] = distance;
}
```

When the collision avoidance core asks the mocked sensor reader for the ForwardLeftSensor's reading, it will return the value in the 'readings_' map, which in this case has been set to 'distance'.

For the core autopilot system, we have the following automated acceptance tests:

TestSaysHelloToTestBench.cxx

```
TEST(SaysHelloToTestBench)
```

TestReportsSensorData.cxx

```
TEST(InitializesSensorsOnStartup)

TEST(ReportsSensorReadingsToTestBench)

TEST(ReportsSensorReadingsToAndroidTablet)
```

TestProxiesCommands.cxx

```
TEST(InitializesCommandProxyOnStartup)
TEST(ProxiesAllUnknownCommandsWithoutModification)
TEST(ProxiesMultipleCommandsPerApplicationCycle)
```

TestInterpretsCommandsCorrectly.cxx

```
TEST(CurrentPlatformUsesExpectedFloatingPointMemoryRepresentation)
TEST(InterpretsArDroneSdksIntFormattedFloatingPointNumbersCorrectly)
```

TestAvoidsWalls.cxx

```
TEST(DoesntCrashIntoWallInFrontOfCopter)
TEST(DoesntCrashIntoAnyWalls)
TEST(CanStillFlyAwayFromWalls)
TEST(FlysAwayAutonomouslyWhenWithinDangerDistanceOfWall)
TEST(DangerRangeWorksForRearSensorsToo)
TEST(AutoCentersWhenFlyingBetweenNarrowWalls)
TEST(NotifiesTabletWhenCollisionsHaveBeenAvoided)
```

TestSupportsDiagonalSensors.cxx

```
TEST(ReportsForwardAngledSensorDataToTestBench)
TEST(UsesClosestForwardSensorReadingForCollisionAvoidance)
TEST(UsesClosestLeftSensorReadingForCollisionAvoidance)
TEST(UsesClosestRightSensorReadingForCollisionAvoidance)
```

TestExitsHoverModeWhenInDanger.cxx

```
TEST(LeavesHoverModeWhenWithinDangerRadiusOfAWall)
TEST(DoesntAffectHoverModeWhenEntirelySafe)
TEST(DoesntAffectHoverModeWhenWithinSafetyZone)
TEST(IgnoresUnintentionalCommandsWhileOverridingHoverMode)
```

TestSmoothsSensorData.cxx

```
TEST(InstantaniousSensorReadingsDontScrewThingsUp)
```

As we develop and enhance our autopilot software, we may add tests to this list as necessary. We believe this set of tests covers our functional requirements, but it is easy to add more as needed.

4.2.1 Automated Acceptance Tests Results

The results obtained from running all of the automated tests once is shown below:

```
cd ../../3rdparty/unittest-cpp/UnitTest++ && make all
make[1]: Entering directory `/home/ardrone/workspace/project/3rdparty/unittest-
cpp/UnitTest++'
make[1]: Nothing to be done for `all'.
make[1]: Leaving directory `/home/ardrone/workspace/project/3rdparty/unittest-
cpp/UnitTest++'
g++ -o .obj/x86/allTests.o AutopilotCore.cxx CollisionAvoidanceCore.cxx
FlightCommandCodec.cxx GroundStationReportingCore.cxx SensorDataSmoother.cxx
StringHelpers.cxx TestBenchLogger.cxx WallDetector.cxx test/TestAvoidsWalls.cxx
test/TestExitsHoverModeWhenInDanger.cxx test/TestInterpretsCommandsCorrectly.cxx
test/testMain.cxx test/TestProxiesCommands.cxx test/TestReportsSensorData.cxx
test/TestReportsToGroundStation.cxx test/TestSaysHelloToTestBench.cxx
test/TestSmoothsSensorData.cxx test/TestSupportsDiagonalSensors.cxx
test/common/FakeArDrone.cxx test/common/FakeCommandProxy.cxx
test/common/FakeGroundStation.cxx test/common/FakeSensorReader.cxx
test/common/FakeTestBenchLogger.cxx test/common/MessageHelpers.cxx
test/common/RandomHelpers.cxx "../../3rdparty/unittest-
cpp/UnitTest++/libUnitTest++.a" -Wall -Werror -I. -
I../../3rdparty/libphidget/libphidget-2.1.8.20120514/ -I"../../3rdparty/unittest-
cpp/UnitTest++/src" -I"./test/" -I"./test/common/"
"../../tools/runTest.sh" .obj/x86/allTests.o .obj/allTests.passed

PASS :    InitializesSensorsOnStartup
PASS :    ReportsSensorReadingsToTestBench
PASS :    LeavesHoverModeWhenWithinDangerRadiusOfAWall
PASS :    DoesntAffectHoverModeWhenEntirelySafe
PASS :    DoesntAffectHoverModeWhenWithinSafetyZone
PASS :    IgnoresUnintentionalCommandsWhileOverridingHoverMode
PASS :    CurrentPlatformUsesExpectedFloatingPointRepresentation
PASS :    InterpretsCrazyIntFloatingPointNumbersCorrectly
PASS :    ReportsForwardAngledSensorData
PASS :    UsesClosestForwardSensorReadingForCollisionAvoidance
PASS :    DoesntCrashIntoWallInFrontOfCopter
PASS :    DoesntCrashIntoAnyWalls
PASS :    CanStillFlyAwayFromWalls
PASS :    FlysAwayAutonomouslyWhenWithinDangerDistanceOfWall
PASS :    DangerRangeWorksForRearSensorsToo
PASS :    AutoCentersWhenFlyingBetweenNarrowWalls
PASS :    InitializesCommandProxyOnStartup
PASS :    ProxiesAllUnknownCommandsWithoutModification
PASS :    ProxiesMultipleCommandsPerCycle
PASS :    InitializesCommunicationWithGroundStation
PASS :    ReportsSensorReadingsToGroundStation
PASS :    FarAwayWallsAreReportedAsZeroDangerLevel
PASS :    WithinSafeDistanceWallsAreReportedRelativeToRemainingDistance
PASS :    WallsAtVehicleNoseAreReportedAsMaximumDangerLevel
PASS :    InstantaniousSensorReadingsDontScrewThingsUp
PASS :    SaysHelloToTestBench

Success: 26 tests passed.
Test time: 0.36 seconds.
```

4.3 INTEGRATION TESTING

Integration testing was the most important testing we had to do for our final project. The testing of the Android application with the onboard autopilot software caught a critical bug in our system, which we were able to fix shortly after integrating both applications. Integration testing was done as follows:

Table 1: Integration Test Results

Test #	Test Description	Setup	Execution	Expected Result	Final Result
1	Fly quadcopter with modified application without having collision avoidance software running.	Turn on AR Drone Connect Tablet to Drone	Fly drone around room while manually avoiding obstacles	Drone should fly as normal with no avoidance behaviour.	PASS
2	Fly quadcopter with Android application. Avoidance application will be running but will be started without Android application	Turn on AR Drone Connect Tablet to Drone Connect laptop to drone to start avoidance application	Hover drone in the air. Have a person walk towards sensors in various directions	Avoidance bars should not appear on the tablet screen until a person approaches a sensor. Drone should fly away from the approaching person.	PASS
3	Fly quadcopter with Android application. Avoidance application will be started with the Android application.	Turn on AR Drone Connect Tablet to Drone Start avoidance software with the Android application	Hover drone in the air. Have a person walk towards the sensors in various directions	Avoidance bars should not appear on the tablet screen until a person approaches a sensor. Drone should fly away from the approaching person.	PASS
4	Fly quadcopter with Android application. Avoidance application will be started with the Android application.	Turn on AR Drone Connect Tablet to Drone Start avoidance software with the Android application	Actively fly the drone with the tablet. Try to fly the drone into walls and other large obstacles.	Avoidance bars should fade in as the drone approaches and obstacle. Drone should fly towards an obstacle until the object gets within its danger zone. The drone should then start to ignore user commands and will not get any closer to the obstacle.	PASS

Overall, integration testing went quite well. Tests 2-4 were repeated several times to ensure that the results from the test were deterministic. The first few times we attempted Test #3, we had issues with the Android application actually starting the avoidance software correctly. We were able to sort this issue out quickly and were happy that our integration test suite caught this issue.

5 SUGGESTIONS FOR FURTHER IMPROVEMENTS

5.1 MAJOR CHALLENGES IN THIS PROJECT

Throughout the course of this project, our group encountered many obstacles that we were able to work through. In addition, there were some particularly difficult challenges due to the fact that we pushed the capabilities of the AR Drone to close to its maximum potential.

1. The development of the application on the drone and tablet were completed separately. When the time came for integration, we ran into several issues where the autopilot software on the drone did not appear to be working when the drone was flown with the Android tablet. Plenty of time was spent debugging and finally it was determined that the source of the problem was a timing issue in the code that would start the autopilot from the tablet. The tablet would in fact issue a telnet command to setup the iptables and immediately after it would start the avoidance autopilot. However, it turns out that the application was starting before the iptables were set up correctly.
2. After the addition of the 6 sensors to the drone it was very hard to be sure if any instability while flying the drone was because we were close to the weight limit or if we were proxying the commands incorrectly.
3. The drone would have issues flying when the battery life fell below 40%; the extra draw from the sensors and microcontroller put an already highly utilized resource at an even higher utilization. Under 40% battery life, the drone would become sluggish and fly awkwardly.
4. The short flight time of the drone made live testing difficult since we would get approximately 5 to 10 minutes per battery charge that took ~1 hour. On top of that flight time, the drone only behaved correctly for the first 5 minutes due to the sluggish flight under 40% battery life.

5.2 LEARNING THROUGH THIS PROJECT

5.2.1 Android Application

None of our team had done much Android development before starting this project so there was a huge learning curve when starting with the tablet development. Since we started with an open source Android application we had to learn most of the Android API before we could make a smart code change. The following list covers the main topics of learning while creating and modifying the application:

5.2.1.1 Android ASyncTask

An Android Asynctask allows for a quick and short side process such as a quick telnet command. The Asynctask allows us to execute a command that would normally block the thread it is called from. This is useful because if we were to execute such a command from the UI thread the application would appear unresponsive while the command completed. For our application we used an Asynctask in order to quickly open up a telnet port and execute the command to setup the modified iptables on the AR drone as well as start the collision avoidance application.

5.2.1.2 Android Threading and Message Handlers

Android threads are very similar to java threads, which meant that getting a thread to run concurrently with the UI was quite simple after reading through the Android threading documentation. The thread was created in order to receive updates from the drone's autopilot about the current onboard sensor states. All of the data received on the thread had to be sent back to the main user interface thread without breaking mutual exclusion or causing any other thread related issues. In order to pass data between the two threads, Android has a Handler class, which can be inherited from. One can then override its handleMessage function in order to pass customized data between the two threads.

5.2.1.3 Dynamically Displaying Avoidance Bars

Displaying the red avoidance bars that fade in and out while flying the quadcopter required a lot of prior code analysis that was a part of the initial open source application since the screen was setup in a certain way before our changes. The current view and projection matrices had to be taken into account otherwise we would end up with either just red bars on the display or the red bars just wouldn't show. In order to make the avoidance bars appear like they were hiding and showing we then adjusted the alpha channel of the bars to the values reported by the drone autopilot application (0 for hide, >0 for show).

5.2.2 Autopilot Application

As Software Engineering students, our team initially had very little experience working with embedded systems. In order to ensure our autopilot software functioned correctly, we had to learn several important principles of embedded system design, including how to effectively share the quadcopter's limited resources with the existing autopilot software, how to develop a toolchain to easily cross-compile and deploy our software, and how to interact with our specialized sensor platform.

5.2.2.1 Sharing Limited System Resources

Modern desktop software development is very forgiving regarding resource usage. Desktop operating system schedulers generally ensure that a misbehaving program can't crash the system. On our quadcopter, interference with the flight control system could cause a literal system crash... into a wall. As a result, we had to learn to be conservative with our resource utilization, and to use profiling tools to ensure we were not exceeding what was available.

5.2.2.2 Developing a Toolchain for Cross-Compiling

As the quadcopter's embedded Linux system is based on an ARM chipset, it cannot execute software built with a standard Intel-targeting compiler. As a result, our team had to learn how to 'cross compile'; that is, configure an Intel-based compiler to create executables for an ARM architecture. This was further complicated by our automated tests, which needed to run on the Intel-based host computer, not on the ARM system. As a result, we learned how to create a portable toolchain, capable of building our autopilot software for either architecture.

5.2.2.3 Interfacing with Embedded Sensor Platform

Software Engineering usually only interacts with high-level operating system APIs, and interfaces with standard computer hardware - a screen, keyboard, mouse, hard disk, etc. Our autopilot system, however, had to interface with ultrasonic range-finding sensors, which had a single analog data output wire. As a result, our team had to learn the entire hardware communication stack - the physical wire that communicated the sensor data, through the USB sensor controller chip, through to the Linux operating system (via libusb, which we had to cross-compile), through the sensor controller API (via libphidgets, also cross-compiled), and into our C++ code.

5.3 SUGGESTIONS FOR FURTHER IMPROVEMENTS

Our project could use a few improvements that would increase the safety and overall usability of the drone. If we had more time with our project we would focus on the following improvements:

1. Find sources of unneeded excess weight and remove them. We would also replace components such as the wires used for the sensors with thinner gauge wire since that's an easy way to reduce a few grams of weight.
2. Add additional back corner sensors. This would increase the safety of the drone while backing up. We were unable to add these in the current state of the project since we already have hit the weight limit of the device.
3. Add the ability to avoid collisions while flying the drone in absolute control mode. Absolute control mode is where only your device's position is used as a point of reference for flying. For example tiling your tablet forward would always cause the drone to fly forward away from you rather than flying forward in the direction that the camera is facing.
4. Find a better tablet to deploy the application on. The Nexus 7 accelerometer requires quite a drastic change in momentum before the application will pick up that the command has changed. Nexus 7 tablet does not have vibration functionality.
5. Audio and vibration avoidance feedback to complement visual feedback.

6 CONCLUSIONS

Our 4th year project was a success overall. We met all of the initial requirements set out by our sponsor and they are excited to try a live demo of our project in the near future. The quadcopter has a working collision avoidance system that can be easily used by any new or seasoned user.

The feedback given to the user is not only useful to the user when they need to know if the drone is in a dangerous situation (when it is not in direct line of sight) but it also does not appear abruptly and is therefore not obtrusive. The sensors chosen for attachment to the quadcopter have performed well enough to do the job we set out for them. Also the overall integration of quadcopter, sensors, and Android application went extremely well with only a few minor hang-ups.

7 REFERENCES

1. "Unmanned Aircraft Regulations", Transport Canada. http://www.tc.gc.ca/eng/civilaviation/standards/general-recavi-brochures-uav-2270.htm
2. "LV Maxbotix EZ1 Specifications Sheet", Phidgets. http://www.phidgets.com/documentation/Phidgets/1128_0_EZ1-Datasheet.pdf
3. "AR.Drone Specifications Sheet", Parrot AR.Drone. http://ardrone.parrot.com/parrot-ar-drone/en/technologies

8 GLOSSARY

CDL Systems - The former company name of our sponsor. The CDL does not stand for anything.

GUI - Graphical user interface.

OEM - Original Equipment Manufacturer

UAV - Unmanned aerial vehicle.

USB - Universal serial bus.

Quadcopter - Also known as a quadrotor; a multicopter with 4 rotors.

SDK - Software development kit.

NDK - Native development kit.