SIMULTANEOUS LOCALIZATION AND MAPPING ON A QUADCOPTER

A Design Project Report

Presented to the School of Electrical and Computer Engineering of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering, Electrical and Computer Engineering

Submitted by:

Shaurya Luthra

MEng Field Advisor: Bruce Robert Land

Degree Date: May 2018

Abstract

Master of Engineering Program

School of Electrical and Computer Engineering

Cornell University

Design Project Report

Project title: Simultaneous Localization and Mapping on a Quadcopter

Author: Shaurya Luthra

Abstract:

The goal of this MEng Design Project for the School of Electrical and Computer Engineering was to integrate autonomous navigation and simultaneous localization and mapping (SLAM) on a custom-built quadcopter. The use of SLAM allows the quadcopter to not only build maps in real time, but also localize itself in these maps. By doing so, the quadcopter can eventually gain the ability to navigate indoor, and eventually outdoor, spaces – mapping down the environment it has seen thus far. By applying a variety of different aggregation methods to those mappings, the quadcopter system could eventually be repurposed for a wide variety of uses. By going one step further and integrating autonomous flight control methods such as optical flow and basic obstacle avoidance, this project demonstrates the viability of a low-cost mapping solution for various uses such as search and rescue and security.

Executive Summary:

For my MEng Design Project for the School of Electrical and Computer Engineering, I worked on integrating autonomous navigation and simultaneous localization and mapping (SLAM) on a custom-built quadcopter. The use of SLAM allows the quadcopter to navigate indoor, and eventually outdoor, spaces – mapping down the environment it has seen thus far. By applying a variety of different aggregation methods to those mappings, the quadcopter system could eventually be repurposed for a wide variety of uses. The scope of this project involved specifically testing and tuning a variety of SLAM algorithms as well as autonomous flight algorithms – utilizing Intel libraries, as well as custom code – to map an indoor space. More specifically, in regard to algorithms, this project involved developing C++ code, that takes in distance data from the Intel Aero's 3D camera, and uses it for obstacle avoidance for flight. Using ROS, and OSS Library ORBSLAM2, the final system is also able to feed in the 3D camera data into the prebuilt SLAM library and achieve real time mapping and localization. Throughout the past two semesters, this project has gained significant success. The Intel Aero quadcopter is currently able hover autonomously using optical flow and move down a hallway. While fully autonomous flight was not yet achieved due to safety concerns and limitations of sensor data, the stable hover from the quadcopter allowed manual indoor navigation while running the custom tuned SLAM algorithm, proving that the end result of the project is not only viable but completely attainable. While the project in of itself might not be completely novel – the uses intended are. SLAM running on autonomous quadcopter can be repurposed to areas where mounted cameras/security personnel are lacking and could help watch over large areas without human intervention. Furthermore, the CV/3D mapping work could be used for search and rescue work in a variety of situations from terror attacks to natural disasters – aiding first responders in reaching victims quickly and efficiently.

Table of Contents

Introduction:

With an ever-growing world with an increasing number of terror attacks and natural disasters, it is becoming more important now than ever before to be able to find survivors and victims in different areas as quickly as possible. Furthermore, in areas dominated by large open natural landscapes, and dangerous features like gorges and cliffs, it is of upmost importance to be able to find missing persons as soon as possible. Unfortunately, in the situations of terror attacks, natural disasters, and missing persons, it can sometimes take hours or even days to find people in unknown or ravaged terrain. One of the main reasons this is the case is that the ability to quickly analyze and map out a terrain to either plan a rescue route, or to just assess damage, is usually limited to very technologically advanced groups who have a lot of training and financial resources in hand. My project aims to resolve this issue by providing a relatively low cost (~$1100) solution that would allow more police departments, and rescue organizations, the real-time information they need to save lives. By using an autonomous quadcopter with mapping abilities, I created the basis for a system that can be released over ravaged or unknown terrain and provide insight to those who would need it in saving lives. The idea for this project came from a very big event in my life where I lost one of my best friends in Cornell's gorges, and it took rescue workers days to find him after he went missing. After realizing what real time maps could do for search and rescue I decided to pursue this project in place of another. This document will go through my design writeup for my MEng project, discussing the possible mapping solutions that were evaluated before arriving at the final design, as well as the final design of my project. The report's appendix also contains a user guide and a full walkthrough on getting the system to its current working state.

Alternative Solutions:

Before starting this project, I had to consider the possible solutions to my end goal – autonomous mapping in order to aid first responders in search and rescue situations. Looking into the possible platforms to map, I realized there were only two main choices, a ground vehicle, or a quadcopter – each with their own pros and cons. The benefits of a ground vehicle were low risk in regard to crashing, a stable platform to do sensing off of, and a basic background in vehicle design from my time on Cornell Baja's racing team. The benefits of a quadcopter were fewer obstacles (in air), more versatility (terrain), and quadcopters being a continuously growing field of research. After thinking more about what I hoped to see out of the project, I ended up choosing a quadcopter as the platform upon which I would develop my autonomous mapping solution. By choosing a quadcopter, I ended up taking on a much more difficult controls project but opened the opportunity for this project to grow into something bigger than a two semester MEng project. With a quadcopter, the work of this project could be adapted for a variety of different uses, as well as regions, from search and rescue in a building collapse to security for an estate – the versatility and clear industry growth of quadcopters in the near future made the selection very straight forward.

Design and Implementation:

Initial Decision Process:

Throughout the course of this project I had to make many design choices prior to settling on my final design. There were several points I had to consider, from which SLAM algorithm would best suit my needs to which quadcopter I should use given the scope of the project. Beyond this I also had to figure out the best mode of integration – combining the autonomous flight code, the Intel RealSense camera feed, SLAM, and the live quadcopter feed, into one cohesive solution

One of the first and most important decisions I had to make during the course of this project was which quadcopter to use as my base platform. During my research I came across two main platforms – the Qualcomm SnapDragon Flight starter kit and the Intel Aero RTF. While both are very powerful embedded Linux based quadcopters – for nearly the same price, the Intel aero also offered a full Intel RealSense system and a prebuilt setup with an integrated flight controller, co-processor, and FPGA based peripheral bus. Given the time frame of my project, and the amount of funds available, I decided upon the Intel

Aero RTF – as it would allow me to fast track development on SLAM and autonomous navigation and be able to flight test right out of the box. More about the Intel Aero RTF can be seen in the hardware design section below.

After deciding on the actual platform, the next set of choices I had to look into was which SLAM algorithm to use. In deciding which algorithm to use I first looked at how they might integrate with the Intel Aero RealSense camera system. Looking through the RealSense libraries, I realized that a ROS Node could publish the camera stream to a ROS topic – which would mean I could use any algorithm that could stream in ROS formatted depth data. Upon doing more research I settled on using either RTabMap or ORBSLAM2, both of which are ROS ready SLAM solutions. In the end I ended up using ORBSLAM2 solely as a result of field testing which will be discussed below.

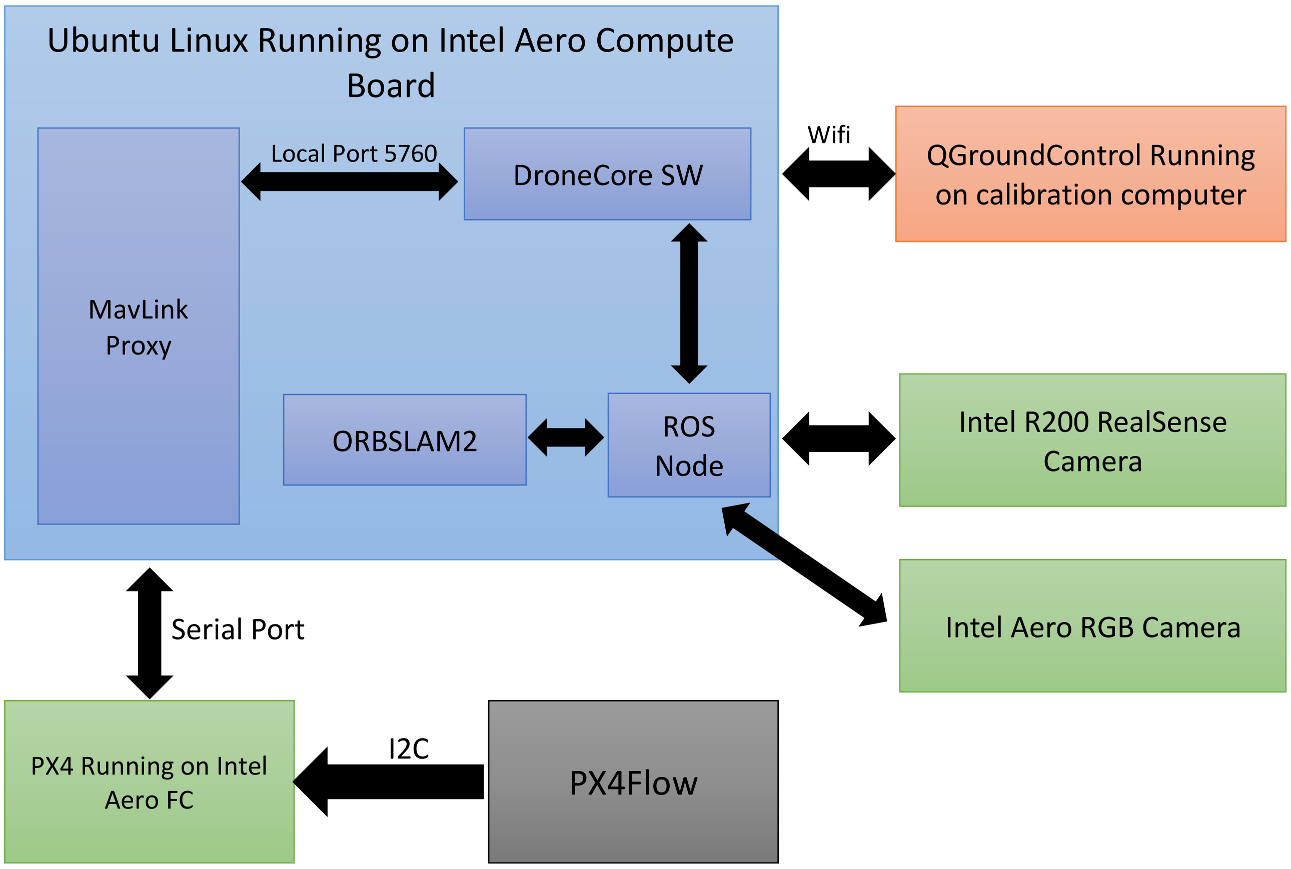
The last major decision that I had to look into was the software platform upon which to develop. Knowing that the Intel Aero natively runs the PX4 autopilot stack – I found that I could use either C++ (Dronecore) or Python (Dronekit), both of which are compatible with the Intel Aero's native flight stack. Looking into both platforms more, and how they might eventually integrate into the final solution, I found that using Dronecore would be the best option to start with – while it is newer than Dronekit and less developed, it is written in C++. This made it eaisly compatible with the Intel RealSense libraries, including the Intel collision avoidance library.

My final design consisted of the Intel Aero RTF that uses ORBSLAM2 and Dronecore libraries. My design approach based on this final setup is outlined below.

Final Design:

This project can be broken down into two main parts – software and hardware, each of which was used in order to enable both SLAM and autonomous flight. The purpose of this section is to go over the overall system architecture, and then discuss the hardware and software design and how they both contribute to SLAM, as well as autonomous flight.

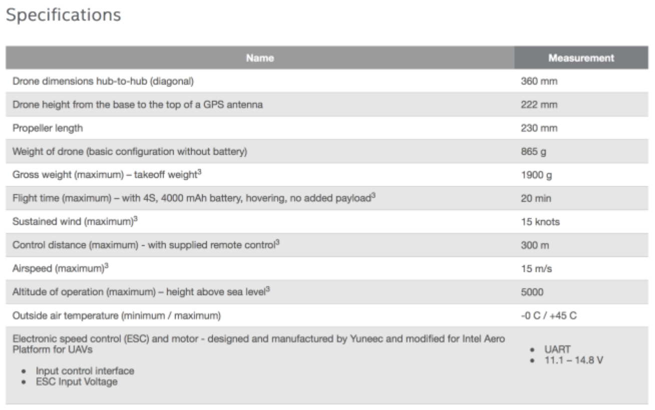
Architecture:


Ubuntu Linux Running on Intel Aero Compute Board
MavLink Proxy
Local Port 5760
DroneCore SW
Wifi
QGroundControl Running on calibration computer
ORBSLAM2
ROS Node
Intel R200 RealSense Camera
Intel Aero RGB Camera
Serial Port
PX4 Running on Intel Aero FC
I2C
PX4Flow

The above diagram illustrates the final design architecture discussed earlier in this section. The main development platform was the Intel Aero RTF's co-processor, a quadcore Intel x7 processor running Ubuntu – note the processor comes with Yocto linux, but Ubuntu was installed (details in appendix) in order to speed along the development process. This processor ran a MavLink proxy which communicated with the PX4 flight controller, the Dronecore autonomous flight software, ORBSLAM2, and ROS. Using these various software tools, the platform interacted with the Intel R200 stereo camera and RGB camera which fed in data used for mapping and localization. Finally, the PX4Flow optical flow chip communicated with the co-processor over I2C and allowed the quadcopter to have an autonomous and stable hover in a GPS denied environment indoors.

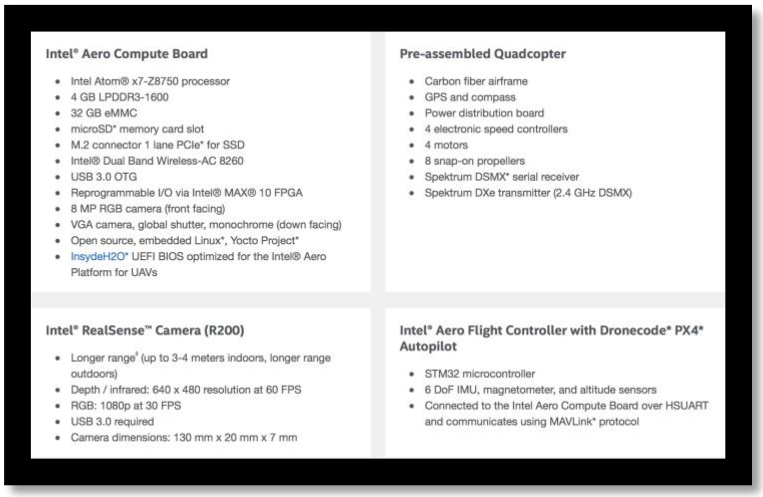
Hardware Design:

While this project in of itself was very reliant on hardware it did not involve a lot of independent hardware design. The hardware platform itself was the Intel Aero RTF Quadcopter seen below with specifications:



Specifications

Name	Measurement
Drone dimensions hub-to-hub (diagonal)	360 mm
Drone height from the base to the top of a GPS antenna	222 mm
Propeller length	230 mm
Weight of drone (basic configuration without battery)	865 g
Gross weight (maximum) – takeoff weight[3]	1900 g
Flight time (maximum) – with 4S, 4000 mAh battery, hovering, no added payload[3]	20 min
Sustained wind (maximum)[3]	15 knots
Control distance (maximum) - with supplied remote control[3]	300 m
Airspeed (maximum)[3]	15 m/s
Altitude of operation (maximum) – height above sea level[3]	5000
Outside air temperature (minimum / maximum)	-0 C / +45 C
Electronic speed control (ESC) and motor - designed and manufactured by Yuneec and modified for Intel Aero Platform for UAVs • Input control interface • ESC Input Voltage	• UART • 11.1 – 14.8 V



Intel® Aero Compute Board

- Intel Atom® x7-Z8750 processor
- 4 GB LPDDR3-1600
- 32 GB eMMC
- microSD* memory card slot
- M.2 connector 1 lane PCIe* for SSD
- Intel® Dual Band Wireless-AC 8260
- USB 3.0 OTG
- Reprogrammable I/O via Intel® MAX® 10 FPGA
- 8 MP RGB camera (front facing)
- VGA camera, global shutter, monochrome (down facing)
- Open source, embedded Linux*, Yocto Project*
- InsydeH2O* UEFI BIOS optimized for the Intel® Aero Platform for UAVs

Pre-assembled Quadcopter

- Carbon fiber airframe
- GPS and compass
- Power distribution board
- 4 electronic speed controllers
- 4 motors
- 8 snap-on propellers
- Spektrum DSMX* serial receiver
- Spektrum DXe transmitter (2.4 GHz DSMX)

Intel® RealSense™ Camera (R200)

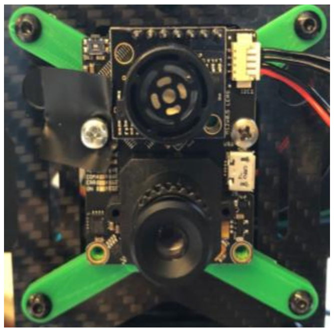
- Longer range[1] (up to 3-4 meters indoors, longer range outdoors)
- Depth / infrared: 640 x 480 resolution at 60 FPS
- RGB: 1080p at 30 FPS
- USB 3.0 required
- Camera dimensions: 130 mm x 20 mm x 7 mm

Intel® Aero Flight Controller with Dronecode* PX4* Autopilot

- STM32 microcontroller
- 6 DoF IMU, magnetometer, and altitude sensors
- Connected to the Intel Aero Compute Board over HSUART and communicates using MAVLink* protocol

Using the above platform allowed for flight capabilities right out of the box, as well as fairly straightforward hardware/software development.

The only other hardware that went into this project was the optical flow setup. The optical flow setup is what allows the quadcopter to hover stably at a given altitude without human input. During the project I used two different setups for stable flight. The first setup was using the Intel Aero's built in downward facing camera and a Garmin LIDAR v3 rangefinder. Using this setup, I was able to achieve a stable flight height, but quickly realized the wide FOV Intel camera was not particularly suitable to visual odometry. Instead I switched over the PX4Flow optical flow chip that uses sonar and a much higher resolution and focused camera to achieve a stable hover. The chip, seen below had to have its firmware modified for indoor use (expanded on here).



Software Design:

A vast majority of this project involved software integration, from working with the PX4Flow optical flow chip, to setting up ROS. The details of the software work in this project are outlined below.

In order to first get the Intel Aero RTF to a stage where I could develop on it, I had to configure everything from the operating system to the installed drivers. While there are a lot of small details, I will focus on the overall setup for this section, with references to the appendix for specifics.

Setting up the Intel Aero involved a series of steps. After first booting it up and confirming it was working out of the box I upgraded the BIOS, the flight controller, and the FPGA expansion board. After doing this I confirmed flight capabilities once again. Following these upgrades, I installed Ubuntu Linux on the x7 processor instead of Yocto Linux. I did this so that I could better develop – as Ubuntu can have packages installed without recompiling the entire kernel, whereas Yocto cannot. Following the installation of Ubuntu, I installed the Intel Aero System, Intel RealSense libraries (including the ROS libraries), the Intel Aero Optical Flow software, and finally both RTabMap as well as ORBSLAM2. The step by step instructions can be found in the appendix here.

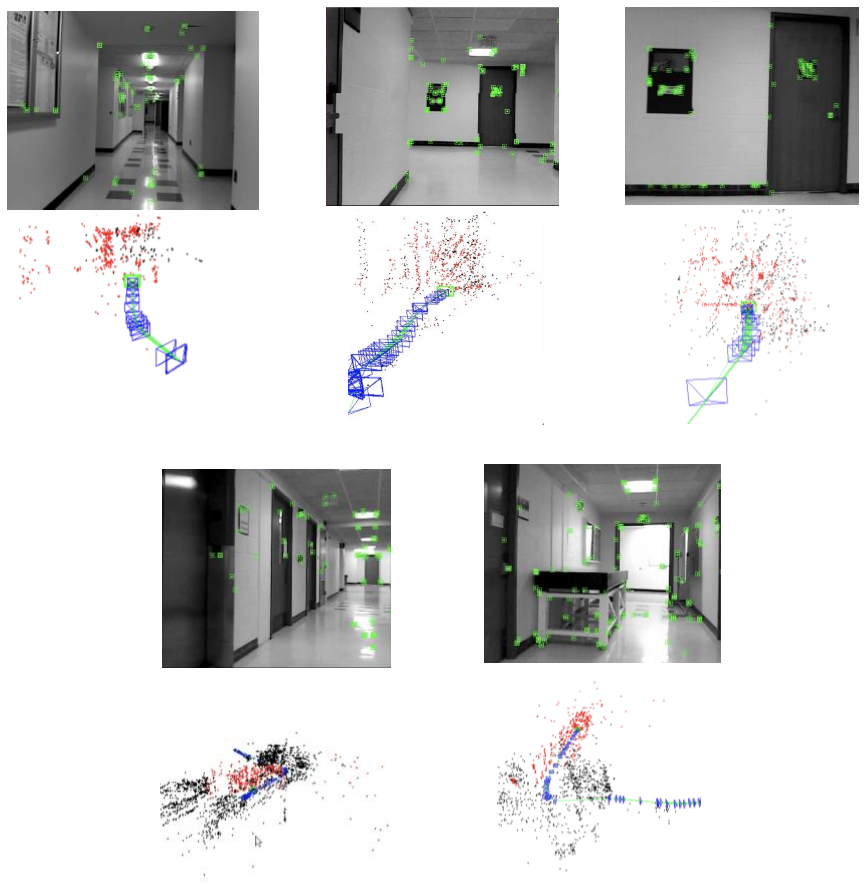
After setting up the system as above, I tested the individual components, testing various ROS Nodes, and streaming depth maps to ensure that the necessary components were functioning properly (appendix).

After confirming overall system functionality, I worked on getting SLAM up and running. In doing so I tested both RTAB-Map and ORBSLAM2. In my tests I found that by using RTAB-Map I was able to make some fairly decent maps, but could not do so when moving the quadcopter at a reasonable rate. With ORBSLAM2, however, I could build maps and localize the quadcopter at speeds that were reasonable when considering my overall search and rescue use case (walking speed in a hallway). With these tests complete I worked on using ORBSLAM2 and better integrating the open source SLAM algorithm with the RealSense R200 camera.

While the ORBSLAM2 mapping software was already developed and put on GitHub for opensource use, it was not developed with the Intel Aero platform in mind. A good portion of my time went into tuning the parameters that allowed it to best use the data from the Intel Aero R200. The parameters, shown below, were found in one of two ways. First a custom written script was written to find the camera's intrinsic parameters – calculating various fields based on these values – and second, were hand tuned through repeated testing.

Camera calibration parameters: Camera.fx: 613.305 Camera.fy: 620.215 Camera.cx: 325.150 Camera.cy: 246.264 **Camera distortion parameters:** Camera.k1: -0.069 Camera.k2: 0.079 Camera.p1: -0.0001 Camera.p2: 0.003 Camera.k3: 0.0 Camera.width: 640 Camera.height: 480 Camera.fps: 30.0 IR projector baseline times fx (approx.): Camera.bf: 36.325 Color order of the images: Camera.RGB: 1 Close/Far threshold: ThDepth: 50.0 DepthMapFactor: 1000.0	**ORB Parameters:** # ORB Extractor: Number of features per image ORBextractor.nFeatures: 1000 # ORB Extractor: Scale factor between levels in the scale pyramid ORBextractor.scaleFactor: 1.2 # ORB Extractor: Number of levels in the scale pyramid ORBextractor.nLevels: 8 # ORB Extractor: Fast threshold ORBextractor.iniThFAST: 15 ORBextractor.minThFAST: 4 **Viewer Parameters:** Viewer.KeyFrameSize: 0.05 Viewer.KeyFrameLineWidth: 1 Viewer.GraphLineWidth: 0.9 Viewer.PointSize: 2 Viewer.CameraSize: 0.08 Viewer.CameraLineWidth: 3 Viewer.ViewpointX: 0 Viewer.ViewpointY: -0.7 Viewer.ViewpointZ: -1.8 Viewer.ViewpointF: 500

The tuning was done incrementally, modifying individual parameters, and retesting, until the SLAM program was able to properly map the quadcopter's surroundings. The results of the SLAM software can be seen below.



Beyond the above work on SLAM, I spent a majority of my MEng working on getting autonomous flight up and running. In order to do so I worked with PX4 and Dronecore developers in order to get optical flow up and running for indoor use, as well as get basic autonomous flight and collision avoidance scripts up and running (found in appendix)

In order to take a step to full autonomous navigation, I first had to figure out a way to have the quadcopter hold its position (x, y, and z coordinates) in a GPS denied environment. Because GPS was out of the question I knew I had to read in sensor data – while range finders could solve the altitude hold problem, I knew they could not solve any translational (x,y) drift. For this reason, I settled on using optical flow. After experimenting with hardware as described above, my final design included the PX4Flow Optical Flow chip, which includes both a high res camera for drift, as well as a sonar for altitude hold. After installing the chip, I realized that my position hold was mediocre at best, with sharp over corrections and occasional jumps in height. To solve this issue, I ended up recompiling the PX4Flow firmware and changing certain state variables that better suit indoor use – using weaker thresholding to accept weaker pattern matching (due to less features and worse lighting indoors). I also modified the PX4Flow driver to scale up the integration_timespan which appears to make the corrections less aggressive and more suitable for indoor use (this has no mathematical proof and was achieved in trial and testing). Setting up PX4Flow can be found in the appendix.

The final bit of software work in this project went into developing autonomous control scripts. For this work I took an incremental approach, but unfortunately, I was not able to achieve full autonomy by this report's writing – for both safety and timeline reasons.

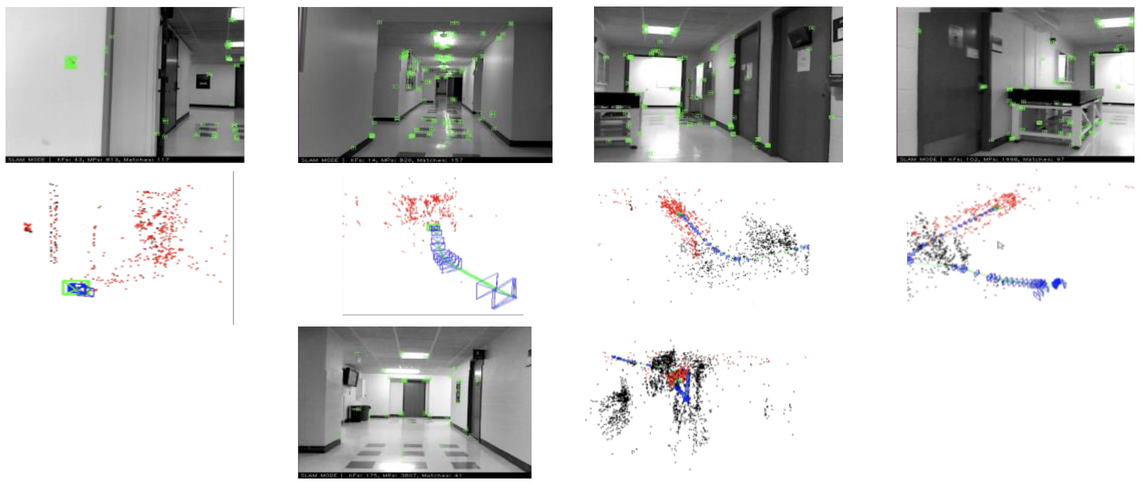
The autonomous software was written in C++ and is a combination of the Intel RealSense and collision avoidance libraries, combined with Dronecore for basic flight control. The scripts seen in the appendix were created by utilizing a variety of sample scripts provided in the various repository's websites.

Design Conclusion:

During the course of this project I was able to successfully implement mapping on a quadcopter and have the quadcopter hover autonomously in a position hold using optical flow. While full autonomy was not achieved, this project successfully creates a platform upon which future work can occur.

Results:

In my original proposal I set out to create a fully autonomous mapping system running on a quadcopter. During the course of this project, I was able to achieve a majority of my goals, as seen below I was able to achieve autonomous position hold, as well as real time SLAM.



During the course of my testing I was also able to make some quantitative measurements outlined in the tables below:

Quadcopter Results	
Flight Time out of box	20 minutes
Flight Time after running Optical Flow and ORBSLAM2	14 minutes
Controllable airspeed indoors pre optical flow	~2.16 ft/sec
Controllable airspeed indoors post optical flow	~.75 ft/sec
Drift velocity pre optical flow	~.3 ft/sec
Drift velocity post optical flow	~.03ft/sec (but self corrects immediately)

SLAM Results	
RTAB-Map mapping speed (translational movement)	~.25 ft/sec
ORBSLAM2 mapping speed (translational movement)	~1.5 ft/sec
RTAB-Map mapping speed (angular movement)	NA (failed quickly after angular movement)
ORBSLAM2 mapping speed (angular movement)	~.78 rads/sec
ORBSLAM2 Reproducibility	~90% reproducible maps (approximately 10% of features come in and out of map on subsequent trials – black shiny objects confuse the Intel R200 camera)

While I was not able to complete everything that I set out to do, given the scope of the system, and the time limitations of an MEng project, a fair amount of success was achieved, and can definitely be built upon in the future. The next steps of this project involve autonomous take-off, which was the main delay in fully autonomous flight. The reason autonomous flight had to be postponed was that signal noise made autonomous flight, and more specifically autonomous takeoff, very dangerous. Also, at the time of writing, the Dronecore software platform is currently incapable of supporting optical flow position mode out of the box. After the sensor data is cleaned up, and the Dronecore api is updated, testing of the autonomous scripts can be completed!

Overall, I was able to achieve a vast majority of what I set out to do. Along the way I was able to establish a proof of concept that illustrates sensor technology is at the point where autonomous aerial navigation is becoming cost effective, however, fully autonomous flight still requires more sensors than the base Aero RTF and PX4Flow board can offer.

Acknowledgements:

I would like to thank my advisor Bruce Land for not only supporting my project but also funding it when alternative funding fell through. I would also like to thank Kirstin H. Petersen for helping me find uncommon parts used to interface with the Intel Aero RTF Quadcopter even when my project was independent of her group. Finally, I would like to thank ECE Squad – Tim Braren, Dhruv Gaba, Patrick Wang, Rohan Ghosh, Ryan de Freitas Bart, and Aalaap Narasipura, for helping me work my hardest, and obtain my Master of Engineering in Electrical and Computer Engineering.

Appendix:

Initial Setup:

- Flashing Intel Aero Linux Distribution
  - Download image at
    - https://downloadcenter.intel.com/download/26389/UAV-installation-files-for-Intel-Aero-Platform?v=t
  - Easy method: For Linux, Windows and MacOS you can use Etcher.
    - Insert the removable/USB disk to the windows machine (it will be formatted)
    - Launch Etcher
    - Select the .iso file, the USB drive and click on the "Flash" button
  - With HDMI Screen connected
    - Wait for the image to be written and verified
    - shut down Intel Aero (unplug-wait 5s-replug)
    - press ESC to enter the BIOS
    - select boot from USB key
    - when booting from USB key, select "install"
- Update BIOS:
  - download the latest BIOS from the Intel Download Center [link] and copy onto the Aero disk space.
  - To install, first remove the previous version (v1.00.13) and then install the latest (v1.00.16)
    - rpm -ev aero-bios-01.00.13-r1.corei7_64
    - rpm -i aero-bios-01.00.16-r1.corei7_64.rpm
    - aero-bios-update
- Flash FPGA
  - Type:
    - cd /etc/fpga/
    - jam -aprogram aero-rtf.jam
- Flash Flight Controller:
  - Type:
    - cd /etc/aerofc/px4/
    - aerofc-update.sh nuttx-aerofc-v1-default.px4
- Check Install
  - Type: aero-get-version.py
  - Should see below (or newer)
    - BIOS_VERSION = Aero-01.00.13
    - OS_VERSION = Poky Aero (Intel Aero linux distro) v1.6.0 (pyro)
    - AIRMAP_VERSION = 1.8
    - FPGA_VERSION = 0xc2
    - Aero FC Firmware Version = 1.6.5

- Calibrate according to https://github.com/intel-aero/meta-intel-aero/wiki/02-Initial-Setup#calibration
- Connect to QGroundControl as stated in following link in order to confirm Aero connectivity
- Note all instructions can be found at https://github.com/intel-aero/meta-intel-aero/wiki/02-Initial-Setup

Setup Ubuntu:

- Install Ubuntu
  - Download Ubuntu 16.04.3 x64 Desktop
  - Create a bootable disk (refer to the Ubuntu documentation)
  - Plug Intel Aero to the wall power supply, the USB-OTG adapter, bootable USB key, hub, keyboard and mouse. Power on
  - Type ESC to enter the BIOS
  - Select boot manager, select your USB key and press Enter
  - Install Ubuntu as you would on a computer
  - In terms of options, my preferences are:
    - I went with a full disk install (erase Yocto and use all space for Ubuntu)
    - I do not install third party proprietary software (flash, mp3)
    - I choose to have my session opening automatically (it has networking consequences).
- Set up Intel Aero Repo:
  - Type:
    - echo 'deb https://download.01.org/aero/deb xenial main' | sudo tee /etc/apt/sources.list.d/intel-aero.list
    - wget -qO - https://download.01.org/aero/deb/intel-aero-deb.key | sudo apt-key add -
    - sudo apt-get update
    - sudo apt-get upgrade
    - sudo apt-get -y install gstreamer-1.0 libgstreamer-plugins-base1.0-dev libgstrtspserver-1.0-dev gstreamer1.0-vaapi gstreamer1.0-plugins-base gstreamer1.0-plugins-good gstreamer1.0-plugins-bad gstreamer1.0-libav ffmpeg v4l-utils python-pip
    - sudo pip install pymavlink
    - sudo apt-get -y install aero-system
    - sudo reboot
- Update BIOS
  - sudo aero-bios-update
  - sudo reboot
- Flash FPGA
  - sudo jam -aprogram /etc/fpga/aero-rtf.jam
- Flash Flight Controller:
  - cd /etc/aerofc/px4/

        - aerofc-update.sh nuttx-aerofc-v1-default.px4
- Install RealSense SDK:
    - Type:
        - cd
        - sudo apt-get -y install git libusb-1.0-0-dev pkg-config libgtk-3-dev libglfw3-dev cmake
        - git clone -b legacy --single-branch https://github.com/IntelRealSense/librealsense.git
        - cd librealsense
        - mkdir build && cd build
        - cmake ../ -DBUILD_EXAMPLES=true -DBUILD_GRAPHICAL_EXAMPLES=true
        - make
        - sudo make install
- Enable Optical Flow:
    - Type:
        - systemctl enable aero-optical-flow
        - systemctl start aero-optical-flow

Setup VNC:

- Follow the steps here:
    - https://www.linode.com/docs/applications/remote-desktop/install-vnc-on-ubuntu-16-04/
- Add Dummy screen according to post by PMLA here:
    - https://ubuntuforums.org/showthread.php?s=1d7ec44878f85eedc7376595b188983b&t=1471785&page=2

Install Robotic Operating System (ROS):

- Install ROS
    - Type:
        - sudo add-apt-repository http://packages.ros.org/ros/ubuntu
        - sudo apt-key adv --keyserver hkp://ha.pool.sks-keyservers.net --recv-key 0xB01FA116
        - sudo apt-get update
        - sudo apt -y install ros-kinetic-desktop-full ros-kinetic-rqt python-rosinstall ros-kinetic-realsense-camera ros-kinetic-mavros ros-kinetic-web-video-server ros-kinetic-visp-tracker ros-kinetic-visp-camera-calibration ros-kinetic-vision-visp ros-kinetic-vision-opencv ros-kinetic-video-stream-opencv ros-kinetic-uvc-camera ros-kinetic-usb-cam ros-kinetic-test-mavros ros-kinetic-rviz-visual-tools ros-kinetic-rostopic ros-kinetic-roslaunch python-rosinstall python-rosinstall-generator python-wstool build-essential ros-kinetic-pyros python-rosdep
        - sudo rosdep init

        - rosdep update
        - echo "source /opt/ros/kinetic/setup.bash" >> ~/.bashrc
        - source ~/.bashrc
        - sudo geographiclib-get-geoids egm96-5
- Launch Camera Node:
    - roscore &
    - roscd realsense_camera
    - roslaunch realsense_camera r200_nodelet_rgbd.launch &
- Test Camera:
    - roscd realsense_camera
    - rosrun rviz rviz -d rviz/realsense_rgbd_pointcloud.rviz

Install SLAM:

- Install RTAB-Map
    - sudo apt-get install ros-kinetic-rtabmap-ros
    - export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/opt/ros/kinetic/lib/x86_64-linux-gnu
- Install ORBSLAM2
    - Install Pangolin: https://github.com/stevenlovegrove/Pangolin
    - Install OpenCV 3.2
        - Install dependencies:
            - sudo apt-get install --assume-yes build-essential cmake git
            - sudo apt-get install --assume-yes pkg-config unzip ffmpeg qtbase5-dev python-dev python3-dev python-numpy python3-numpy
            - sudo apt-get install --assume-yes libopencv-dev libgtk-3-dev libdc1394-22 libdc1394-22-dev libjpeg-dev libpng12-dev libtiff5-dev libjasper-dev
            - sudo apt-get install --assume-yes libavcodec-dev libavformat-dev libswscale-dev libxine2-dev libgstreamer0.10-dev libgstreamer-plugins-base0.10-dev
            - sudo apt-get install --assume-yes libv4l-dev libtbb-dev libfaac-dev libmp3lame-dev libopencore-amrnb-dev libopencore-amrwb-dev libtheora-dev
            - sudo apt-get install --assume-yes libvorbis-dev libxvidcore-dev v4l-utils python-vtk
            - sudo apt-get install --assume-yes liblapacke-dev libopenblas-dev checkinstall
            - sudo apt-get install --assume-yes libgdal-dev
        - Download OpenCV: https://github.com/opencv/opencv/archive/3.2.0.zip
        - Enter the unpacked directory:

            - mkdir build
            - cd build/
            - cmake -D CMAKE_BUILD_TYPE=RELEASE -D CMAKE_INSTALL_PREFIX=/usr/local -D FORCE_VTK=ON -D WITH_TBB=ON -D WITH_V4L=ON -D WITH_QT=ON -D WITH_OPENGL=ON -D WITH_CUBLAS=ON -D CUDA_NVCC_FLAGS="-D_FORCE_INLINES" -D WITH_GDAL=ON -D WITH_XINE=ON -D BUILD_EXAMPLES=ON ..
            - make -j $(($(nproc) + 1))
        - Install using checkmake:
            - sudo apt-get install checkinstall
            - sudo checkinstall
    - Install Eigen3 according to http://eigen.tuxfamily.org
    - Make catkin workspace:
        - mkdir -p ~/catkin_ws/src
        - cd ~/catkin_ws/src
        - catkin_init_workspace
        - cd ~/catkin_ws/
        - catkin_make
    - Install ORBSLAM2
        - cd ~/catkin_ws/src
        - git clone https://github.com/raulmur/ORB_SLAM2.git ORB_SLAM2
        - cd ORB_SLAM2
        - chmod +x build.sh
        - ./build.sh
    - Install ORBSLAM2 ROS Nodes
        - export ROS_PACKAGE_PATH=${ROS_PACKAGE_PATH}:~/catkin_ws/src/ORB_SLAM2/Examples/ROS
        - cd ~/catkin_ws/src/ORB_SLAM2
        - chmod +x build_ros.sh
        - ./build_ros.sh

Running SLAM:

- Run RTAB_Map
    - roscd realsense_camera
    - roslaunch realsense_camera r200_nodelet_rgbd.launch &
    - roslaunch rtabmap_ros rgbd_mapping.launch rtabmap_args:="--delete_db_on_start" depth_registered_topic:=/camera/depth_registered/sw_registered/image_rect_raw
- Run ORBSLAM2
    - Get Camera Parameters:
        - cd ~/librealsense/examples

- Add this file:

```
#include<iostream>
#include<stdio.h>
#include <librealsense/rs.hpp>

int main(int argc, char** argv)
{
    rs::context ctx;
    if(ctx.get_device_count() == 0) return EXIT_FAILURE;
    rs::device * dev = ctx.get_device(0);
    dev->enable_stream(rs::stream::depth, 640, 480, rs::format::z16, 30);
    dev->enable_stream(rs::stream::color, 640, 480, rs::format::rgb8, 30);
    dev->start();

    const float scale = dev->get_depth_scale();
    const float rgb_fx = dev->get_stream_intrinsics(rs::stream::color).fx;
    const float rgb_fy = dev->get_stream_intrinsics(rs::stream::color).fy;
    const float rgb_cx = dev->get_stream_intrinsics(rs::stream::color).ppx;
    const float rgb_cy = dev->get_stream_intrinsics(rs::stream::color).ppy;
    const float rgb_k1 = dev->get_stream_intrinsics(rs::stream::color).coeffs[0];
    const float rgb_k2 = dev->get_stream_intrinsics(rs::stream::color).coeffs[1];
    const float rgb_p1 = dev->get_stream_intrinsics(rs::stream::color).coeffs[2];
    const float rgb_p2 = dev->get_stream_intrinsics(rs::stream::color).coeffs[3];
    const float rgb_k3 = dev->get_stream_intrinsics(rs::stream::color).coeffs[4];
    const float baseline = dev->get_extrinsics(rs::stream::depth,
rs::stream::color).translation[0];


    printf("Camera.fx: %.3f\nCamera.fy: %.3f\nCamera.cx: %.3f\nCamera.cy: %.3f\n\n",
            rgb_fx, rgb_fy, rgb_cx, rgb_cy);
    printf("Camera.k1: %.3f\nCamera.k2: %.3f\nCamera.p1: %.3f\nCamera.p2:
%.3f\nCamera.k3: %.3f\n\n",
            rgb_k1, rgb_k2, rgb_p1, rgb_p2, rgb_k3);
    printf("Camera.bf:  %.3f\nDepthMapFactor: %.3f\n", fabs(baseline*rgb_fx),
1/scale);
    return 0;
}
```

- Add this to the CMakeLists file:
    - add_executable(find_params find_params.cpp) target_link_libraries(find_params ${DEPENDENCIES})
- type cd ../build
- type
    - cmake ../ -DBUILD_EXAMPLES=true -DBUILD_GRAPHICAL_EXAMPLES=true
    - make
    - sudo make install

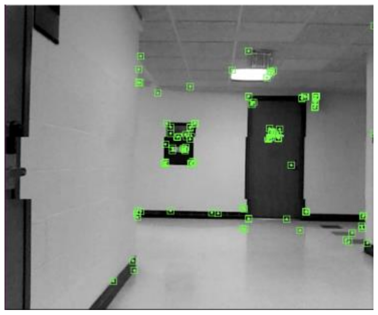
- Run parameter script:
    - ./find_params
- Save parameters above into YAML file as shown here:

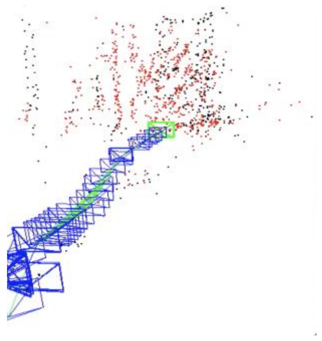
```
%YAML:1.0

# Camera Parameters. Adjust them!

# Camera calibration parameters (OpenCV)
Camera.fx: 613.305
Camera.fy: 620.215
Camera.cx: 325.150
Camera.cy: 246.264

# Camera distortion paremeters (OpenCV) --
Camera.k1: -0.069
Camera.k2: 0.079
Camera.p1: -0.0001
Camera.p2: 0.003
Camera.k3: 0.0

Camera.width: 640
Camera.height: 480

# Camera frames per second
Camera.fps: 30.0

# IR projector baseline times fx (aprox.)
Camera.bf: 36.325

# Color order of the images (0: BGR, 1: RGB. It is ignored if images are grayscale)
Camera.RGB: 1

# Close/Far threshold. Baseline times.
ThDepth: 50.0

# Deptmap values factor
DepthMapFactor: 1000.0

#-------------------------------------------------------------------------------------
-------
# ORB Parameters
#-------------------------------------------------------------------------------------
-------

# ORB Extractor: Number of features per image
ORBextractor.nFeatures: 1000

# ORB Extractor: Scale factor between levels in the scale pyramid
```

```
ORBextractor.scaleFactor: 1.2

# ORB Extractor: Number of levels in the scale pyramid
ORBextractor.nLevels: 8

# ORB Extractor: Fast threshold
# Image is divided in a grid. At each cell FAST are extracted imposing a minimum
response.
# Firstly we impose iniThFAST. If no corners are detected we impose a lower value
minThFAST
# You can lower these values if your images have low contrast
ORBextractor.iniThFAST: 20
ORBextractor.minThFAST: 7

#--------------------------------------------------------------------------------------
-------
# Viewer Parameters
#--------------------------------------------------------------------------------------
-------
Viewer.KeyFrameSize: 0.05
Viewer.KeyFrameLineWidth: 1
Viewer.GraphLineWidth: 0.9
Viewer.PointSize: 2
Viewer.CameraSize: 0.08
Viewer.CameraLineWidth: 3
Viewer.ViewpointX: 0
Viewer.ViewpointY: -0.7
Viewer.ViewpointZ: -1.8
Viewer.ViewpointF: 500
```

- After saving above YAML file,
- roscd realsense_camera
- roslaunch realsense_camera r200_nodelet_rgbd.launch &
- rosrun ORB_SLAM2 RGBD {path}/ORB_SLAM2/Vocabulary/ORBvoc.txt {path}/RealSense.yaml /camera/depth_registered/image_raw:=/camera/depth/image_raw
- Note in above we remap a depth topic because ORBSLAM2 uses a different topic then the Intel node publishes

Output should look as follows:





Setup PX4 Environment on Local Computer:

- Follow steps here: https://dev.px4.io/en/setup/getting_started.html
    - Initial setup
    - Toolchain installation
    - Building the code

Setup Optical Flow:

- For Lidar V3 follow:
    - Install Lidar V3:
        - https://docs.px4.io/en/flight_controller/intel_aero.html
    - in QGroundControl change these parameters:
        - EKF2_AID_MASK = 3 (use GPS + use optical flow)
        - EKF2_HGT_MODE = 2 (range sensor)
    - inside of an Aero terminal run:
        - systemctl start aero-optical-flow.service
        - change start to enable if you want it to run at every boot
- For PX4Flow
    - Follow steps here under image quality and output (Note you need QGroundControl 3.1.3 or earlier):
        - https://pixhawk.org/modules/px4flow
    - Modify PX4Flow firmware:
        - Download firmware from https://github.com/PX4/Flow
        - Modify lines in Flow/src/modules/flow/settings.c

```
//      global_data.param[PARAM_BOTTOM_FLOW_VALUE_THRESHOLD] = 8 * 8 * 20;
        global_data.param[PARAM_BOTTOM_FLOW_VALUE_THRESHOLD] = 5000; // threshold is
irrelevant with this value
        strcpy(global_data.param_name[PARAM_BOTTOM_FLOW_VALUE_THRESHOLD],
"BFLOW_V_THLD");
        global_data.param_access[PARAM_BOTTOM_FLOW_VALUE_THRESHOLD] = READ_WRITE;

//      global_data.param[PARAM_BOTTOM_FLOW_FEATURE_THRESHOLD] = 100;
        global_data.param[PARAM_BOTTOM_FLOW_FEATURE_THRESHOLD] = 40;
        strcpy(global_data.param_name[PARAM_BOTTOM_FLOW_FEATURE_THRESHOLD],
"BFLOW_F_THLD");
        global_data.param_access[PARAM_BOTTOM_FLOW_FEATURE_THRESHOLD] = READ_WRITE;
```

        - Build the PX4Flow firmware using:
            - make archives - this needs to be done only once
            - make
        - Copy firmware from: Flow/Build/px4flow-v1_default.build/firmware.px4 to known location
        - Update firmware to new firmware as done previously through QGroundControl
    - Modify PX4 Firmware:
        - Download from: https://github.com/PX4/Firmware
        - To line 682 in Firmware/ROMFS/px4fmu_common/init.d/rcS add
          if ver hwcmp AEROFC_V1
          then
                  px4flow start &
          fi
        - In Firmware/cmake/configs/nuttx_aerofc-v1_default.cmake add
            - drivers/px4flow
        - In Firmware/src/drivers/px4flow/px4flow.cpp change lines
            - report.integration_timespan = 1.1* f_integral.integration_timespan
        - Make PX4 Firmware using steps here for nuttx boards (aero-rtf)
            - https://dev.px4.io/en/setup/building_px4.html
    - Copy firmware from build folder to Aero RTF
    - Update firmware using same method in Ubuntu setup but with new firmware instead of old flight controller firmware
    - Connect PX4Flow to Intel Aero FC and setup according to http://www.instructables.com/id/Intel-Aero-Drone-Altitude-and-Position-Hold-Using-/
        - ONLY FOLLOW STEPS 7-9

Setup Dronecore:

- On the Aero RTF follow these steps to install Dronecore on a Linux Machine:
    - https://docs.dronecore.io/en/contributing/build.html
- Run and build the Takeoff and land example (OUTSIDE) as described here:
    - https://docs.dronecore.io/en/examples/#trying_the_examples
- Should see an output as shown here:
    - https://docs.dronecore.io/en/examples/takeoff_and_land.html
- Test custom script below, create file and build and run as an example as stated here:
    - https://docs.dronecore.io/en/guide/toolchain.html

```
//
// Simple example to demonstrate how to use DroneCore.
//
// Author: Julian Oes <julian@oes.ch>

#include <chrono>
#include <cstdint>
#include <dronecore/action.h>
#include <dronecore/dronecore.h>
#include <dronecore/telemetry.h>
#include <dronecore/offboard.h>
#include <iostream>
#include <thread>
#include <iostream>
#include <memory>
#include <vector>

#include <coav/coav.hh>

#include "coav-control.hh"
#ifdef WITH_VDEBUG
#include "visual.hh"
#endif

using namespace dronecore;
using namespace std::this_thread;
using namespace std::chrono;
using namespace std;

#define ERROR_CONSOLE_TEXT "\033[31m" //Turn text on console red
#define TELEMETRY_CONSOLE_TEXT "\033[34m" //Turn text on console blue
#define NORMAL_CONSOLE_TEXT "\033[0m"  //Restore normal console colour

void usage(std::string arg);

int main(int argc, char **argv)
{


    shared_ptr<MavQuadCopter> vehicle = opts.port ?
    std::make_shared<MavQuadCopter>(opts.port) : std::make_shared<MavQuadCopter>();

    shared_ptr<DepthCamera> sensor;
    sensor = make_shared<RealSenseCamera>(640, 480, 30);

    shared_ptr<Detector> detector;
    detector = make_shared<DepthImageObstacleDetector>(5.0);

    shared_ptr<CollisionAvoidanceStrategy<MavQuadCopter>> avoidance;
    avoidance = make_shared<QuadCopterStopAvoidance>(vehicle);


    DroneCore dc;
    std::string connection_url;
    ConnectionResult connection_result;

    bool discovered_system = false;
    if (argc == 1) {
        usage(argv[0]);
        return 1;
    } else {
        connection_url = argv[1];
        connection_result = dc.add_any_connection(connection_url);
    }

    if (connection_result != ConnectionResult::SUCCESS) {
        std::cout << ERROR_CONSOLE_TEXT << "Connection failed: "
                  << connection_result_str(connection_result)
                  << NORMAL_CONSOLE_TEXT << std::endl;
        return 1;
    }

    std::cout << "Waiting to discover system..." << std::endl;
    dc.register_on_discover([&discovered_system](uint64_t uuid) {
        std::cout << "Discovered system with UUID: " << uuid << std::endl;
        discovered_system = true;
    });

    // We usually receive heartbeats at 1Hz, therefore we should find a system after around 2
seconds.
    sleep_for(seconds(2));

    if (!discovered_system) {
        std::cout << ERROR_CONSOLE_TEXT << "No system found, exiting." << NORMAL_CONSOLE_TEXT <<
std::endl;
        return 1;
    }

    // We don't need to specify the UUID if it's only one system anyway.
    // If there were multiple, we could specify it with:
    // dc.system(uint64_t uuid);
    System &system = dc.system();

    auto telemetry = std::make_shared<Telemetry>(system);
    auto action = std::make_shared<Action>(system);

    // We want to listen to the altitude of the drone at 1 Hz.
    const Telemetry::Result set_rate_result = telemetry->set_rate_position(1.0);
    if (set_rate_result != Telemetry::Result::SUCCESS) {
        std::cout << ERROR_CONSOLE_TEXT << "Setting rate failed:" << Telemetry::result_str(
                      set_rate_result) << NORMAL_CONSOLE_TEXT << std::endl;
        return 1;
    }


    // Set up callback to monitor altitude while the vehicle is in flight
    telemetry->position_async([](Telemetry::Position position) {
        std::cout << TELEMETRY_CONSOLE_TEXT // set to blue
                  << "Altitude: " << position.relative_altitude_m << " m"
```

```
                << NORMAL_CONSOLE_TEXT // set to default color again
                << std::endl;
    });

    // Check if vehicle is ready to arm
    while (telemetry->health_all_ok() != true) {
        std::cout << "Vehicle is getting ready to arm" << std::endl;
        sleep_for(seconds(1));
    }

    // Arm vehicle
    std::cout << "Arming..." << std::endl;
    const ActionResult arm_result = action->arm();

    if (arm_result != ActionResult::SUCCESS) {
        std::cout << ERROR_CONSOLE_TEXT << "Arming failed:" << action_result_str(
                     arm_result) << NORMAL_CONSOLE_TEXT << std::endl;
        return 1;
    }
    action->set_takeoff_altitude(1.0)

    // Take off
    std::cout << "Taking off..." << std::endl;
    const ActionResult takeoff_result = action->takeoff_async();
    if (takeoff_result != ActionResult::SUCCESS) {
        std::cout << ERROR_CONSOLE_TEXT << "Takeoff failed:" << action_result_str(
                     takeoff_result) << NORMAL_CONSOLE_TEXT << std::endl;
        return 1;
    }

    auto offboard = std::make_shared<Offboard>(system);


    offboard->set_velocity_body({0.0f, 0.0f, 0.0f, 0.0f});

    Offboard::Result offboard_result = offboard->start();
    if (result != Offboard::Result::SUCCESS) {
        std::cerr << "Offboard::start() failed: "
        << Offboard::result_str(offboard_result) << std::endl;
    }

    while(!detector->detect(sensor->read()) ){
        offboard->set_velocity_ned({1.0f, 0.0f, 0.0f, 0.0f});

    }
    offboard->set_velocity_ned({0.0f, 0.0f, 0.0f, 0.0f});
    offboard->stop();



    // Let it hover for a bit before landing again.
    sleep_for(seconds(2));

    std::cout << "Landing..." << std::endl;
    const ActionResult land_result = action->land();
    if (land_result != ActionResult::SUCCESS) {
        std::cout << ERROR_CONSOLE_TEXT << "Land failed:" << action_result_str(
                      land_result) << NORMAL_CONSOLE_TEXT << std::endl;
        return 1;
    }

    // We are relying on auto-disarming but let's keep watching the telemetry for a bit longer.
    sleep_for(seconds(5));
    std::cout << "Finished..." << std::endl;
    return 0;
}

void usage(std::string arg)
{
    std::cout << NORMAL_CONSOLE_TEXT << "Usage : " << arg << " [connection_url]" << std::endl
              << "Connection URL format should be :" << std::endl
              << " For TCP : tcp://[server_host][:server_port]" << std::endl
              << " For UDP : udp://[bind_host][:bind_port]" << std::endl
              << " For Serial : serial:///path/to/serial/dev[:baudrate]" << std::endl;
    std::cout << "Default connection URL is udp://:14540" << std::endl;
}
```