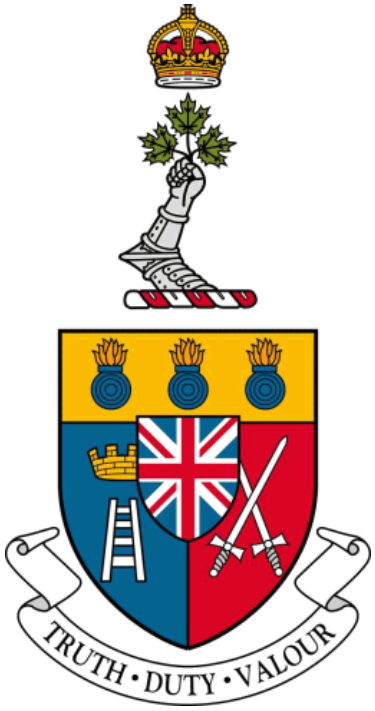
Royal Military College of Canada

Department of Electrical and Computer Engineering



Designing Coatimunde

Computer Optics Analyzing Trajectories In Mostly Unknown, Navigation Denied, Environments

DID-04 - Preliminary Design

Presented by:
Amos Navarre Hebb & Kara Stephan

Presented to:
Dr. Sidney Givigi & Dr. Rachid Beguenane

November 22, 2018

Contents

1 Introduction

1.1 Document Purpose

Using Computer Optics for Analyzing Trajectories in Mostly Unknown, Navigation Denied, Environments (COATIMUNDE) is the goal of this project. The purpose of this document is to outline the preliminary design for COATIMUNDE. That is, what the design of the project is, how it will be built, and how this meets the requirements for the project. These requirements have been outlined in the Statement of Requirements. The design of the project thus far will be shown and discussed within the Design Section. This document will then identify the constraints, risks, and difficulties that have been faced so far in the project.

1.2 Background

Both in the consumer and professional sectors the use of autonomous aerial vehicles is growing quickly. Currently these vehicles rely on skilled pilots to accomplish a very limited set of tasks. Adding obstacle avoidance capabilities to these vehicles and simplifying the task of following targets could allow for these systems to be used in many more situations. This section will give a quick background on obstacle avoidance, unmanned aircraft systems, computer vision, and the platforms we intend to use in this project.

1.2.1 Obstacle Avoidance

Obstacle avoidance is the task of satisfying a control objective, in this case moving toward a visual target, while subject to non-intersection or non-collision position constraints. The latter constraints are, in this case, to be dynamically created while moving in a reactive manner, instead of being pre-computed.

1.2.2 Unmanned Aircraft Systems

Very generally any powered vehicle that uses aerodynamic forces to provide lift, without a human operator being carried, can be considered an unmanned aerial vehicle. Currently most of these vehicles make up a single component of a larger unmanned aircraft system.

An Unmanned aircraft system (UAS), or remotely piloted aircraft system (RPAS), is an aircraft without a human pilot on-board, instead controlled from an operator on the ground. Such a system can have varying levels of autonomy, something as simple as a model aircraft could be considered a UAS without any automation capabilities. Detecting, recognizing, identifying, and tracking targets of interest in complex environments and integrate with the systems required to process and fuse the collected information into actionable intelligence while operating in a low-to-medium threat environment is the current goal of the RPAS project by the Royal Canadian Air Force (RCAF) [1].

Flying a UAS requires a secure link to the operator off-board. Maintaining this link, particularly while flying close to the ground where more opportunities for interference are introduced is difficult. This difficulty is compounded in environments where potentially hostile actors may be attempting to jam communications. This necessitates a level of automation on-board capable of maintaining flight while denied navigation information.

There are many different types of approaches for this problem, but most involve some form of identifying targets in real time and reacting as they become visible to the aircraft. This has proven successful on a flying robot traveling at high speeds [2]. This system successfully combined trajectory libraries and a state machine to avoid obstacles using very little computational power even at very high speeds [3]. Another solution to obstacle avoidance on flying robots was the creation of NanoMap [4]. This allows for 3D data to be processed at a much faster rate allowing for higher speeds of the robot [4].

1.2.3 Computer Vision

Currently there are many different ways that computers can make high-level decisions based on digital image information. There are many methods to acquire, process, and analyze data from the real world using a camera. While this is a very broad field, we intend to focus on motion estimation and object recognition. Both will be working with a video stream taken from a camera.

Motion estimation can be accomplished using direct methods which compare whole fields of pixels to each other over successive frames, compared to indirect methods which look for specific features. The information resulting from motion estimation streams can be used to both compensate for motion while analyzing other aspects of an image, and update a state machine.

Object recognition in our project will be accomplishing two tasks: identifying a marker or target which will require more involved object recognition calculations, and very simple techniques, such as edge detection, to identify obstacles that exist in the path of the robot.

1.2.4 OpenCV

The Open Source Computer Vision Library (OpenCV) of programming functions is a cross-platform and free for use collection of functions primarily aimed at real-time computer vision[5]. Most well documented techniques to accomplish all of the computer vision goals of our project have already been created and refined in OpenCV. For this reason we will be leaning heavily on OpenCV functions.

1.2.5 Gazebo

Gazebo is a robot simulator that allows for creation of a realistic environment which includes both obstacles and markers similar to those being used in the lab. It will be used to rapidly test algorithms.

1.2.6 Robot Operating System

The Robot Operating System (ROS) is a distributed message system that allows for various sensors and processors to work together to control a robot. It is open source and has been integrated already with OpenCV and Gazebo. There are many additional tools for detecting obstacles, mapping the environment, planning paths, and much more. It is also a robust messaging system that has been proven to be capable of real-time processes.

1.2.7 TurtleBot

The TurtleBot is a robot kit with open-source design and software. A TurtleBot is a robot built to the specification for TurtleBot Compatible Platforms[6]. In our case this is a Kobuki Base, an Acer Netbook running Ubuntu with ROS packages added, an X-Box Kinect, and some mounting plates.

The resulting robot looks like a disk supporting some circular shelves with a small laptop and a camera on the shelves. The base disk is 35.4cm in diameter, the topmost shelf is 42cm from the ground. The robot has a maximum speed of 0.65m/s.

1.2.8 AscTec Pelican

The Ascending Technologies Pelican is a 65.1cm by 65.1cm quad-copter designed for research purposes[7]. It includes a camera, a high level and low level processor set up for computer vision, and simultaneous localization and mapping (SLAM) research. It is also capable of interfacing easily with other controllers and can carry up to a kilogram of additional gear.

2 Design

2.1 Computer Vision

Computer optics is used as the robot will employ a camera to identify objects in the surrounding environment, specifically targets and obstacles. There are three unique manners that computer vision may be used as an input source for this project; these are target finding, obstacle finding, and optical flow. If all three are implemented completely it should be possible to create a navigation kit that uses exclusively a camera as input resulting in a very transferable and lightweight package for flying robots.

2.1.1 Target Finding

While finding arbitrary targets would be preferred, this project is going to be limited to finding special targets designed to be easily identified in a busy environment called ArUco shapes. There are existing libraries in OpenCV that are useful for identifying ArUco shapes with very little overhead.

2.1.2 Obstacle Finding

Finding obstacles will be a considerable aspect of this project. There are various algorithms which leverage the robot's motion through the environment to extract key features of the environment, indicating the potential presence of obstacles. Parallax shift is the tendency for items that are closer to a camera to appear to move more or change in size more than a background that is further away and can be used to indicate where items are and how close they are. Occlusion is where one item moves in front of another item and covers it up, this indicates that the item still in view is closer to the camera than the item that has been hidden. OpenCV contains libraries for both of these tasks, as well as others that may end up being employed to identify obstacles in the environment.

2.1.3 Optical Flow

Optical Flow is a possible source of motion information for the robot. As a camera is moved through an environment the entire image will appear to scale larger as the camera moves forward, rotate left and right as a robot rolls, and shift left, right, up, and down as the robot pitches and yaws. Calculating shifts from one frame to another for global shifts caused by movement of the camera is a mathematically intensive process beyond the scope of this project. OpenCV contains libraries to try to parse this information, but it requires a considerable amount of effort to combine this information with a model of a robot to get useful position estimation.

2.2 Analyzing Trajectories

Analyzing trajectories is a term that refers to the robot completing movements to go towards the target. To do path planning the project will implement potential fields. Potential fields function through creating vectors pointing to objects in the environment and assigning them positive or negative. The positive vectors attracts the robot towards the target and the negative makes obstacles repulsive to the robot [8]. This algorithm was selected to do path planning as the robot will not have to build a map of the environment to create path to the target [9]. Potential fields allow for the robot to change its path along the way if it discovers new obstacles. As the robot is moving the potential fields must be updated to account for how much the robot moved. This will be solved through creating a state machine.

2.3 Unknown and Navigation Denied Environments

The robot must always remember where identified target and obstacles are in relation to itself even when its own position changes. Navigation denied means that the robot will not have access to a GPS to locate itself, so it must search its unknown environment to identify targets and obstacles. An unknown environment can be defined as an environment in which the robot has no prior knowledge of

its surroundings and must identify objects in this environment itself through computer vision. Potential fields will be used to track the location of the target and obstacles through updating each objects vector. These vectors will be used in the state machine to properly create a path to the target, as they are either negative for the target or positive for obstacles. This once again is the reason that the project will be using potential fields to plan the path to the target.

2.4 Aim

The aim of this project is to design a high level control system that will allow an air robot to identify a target and move toward it, avoiding any obstacles that are in the way. Tracking targets of interest in complex environments with a flying robot is the ultimate goal of this project. To accomplish this goal we will be using a TurtleBot and then a flying robot with only one camera to identify targets and obstacles.

2.5 Block Diagram

The block diagram that is shown in Figure 1 contains the outline of the COATIMUNDE project. It is a very high level overview and does not contain references to most of the hardware in use for two reasons: the actual robot and laptop being used are relatively arbitrary as long as the software included is present on both. ROS also includes many nodes which are not represented on the block diagram. Only the nodes that are most important to our project, will have to be specifically added to ROS, or will have to be custom created are included on our system block diagram shown below.

The laptop contains the initialization, target selection, and RViz. The laptop also takes the user input when necessary and processes it appropriately. The robot will do all the target identification, obstacle avoidance, and movement decisions autonomously. This means it will take input from the camera into a video processing node and then create a corresponding ROS message to be sent to the other systems on the robot. These messages are created and used throughout the different systems on the robot. The model of the robot's environment will be found subscribing to messages from the obstacle finder node, target finder node, and the state estimate node. This model then creates a message to pass to the route planning node which tells the movement node how to update its position and speed.

Most of the nodes used in this project are either being used stock, or are very minimally modified from provided libraries. The four nodes, highlighted in grey, which are being custom built for this project are the State Estimate, Target Finder, and Obstacle Finder.

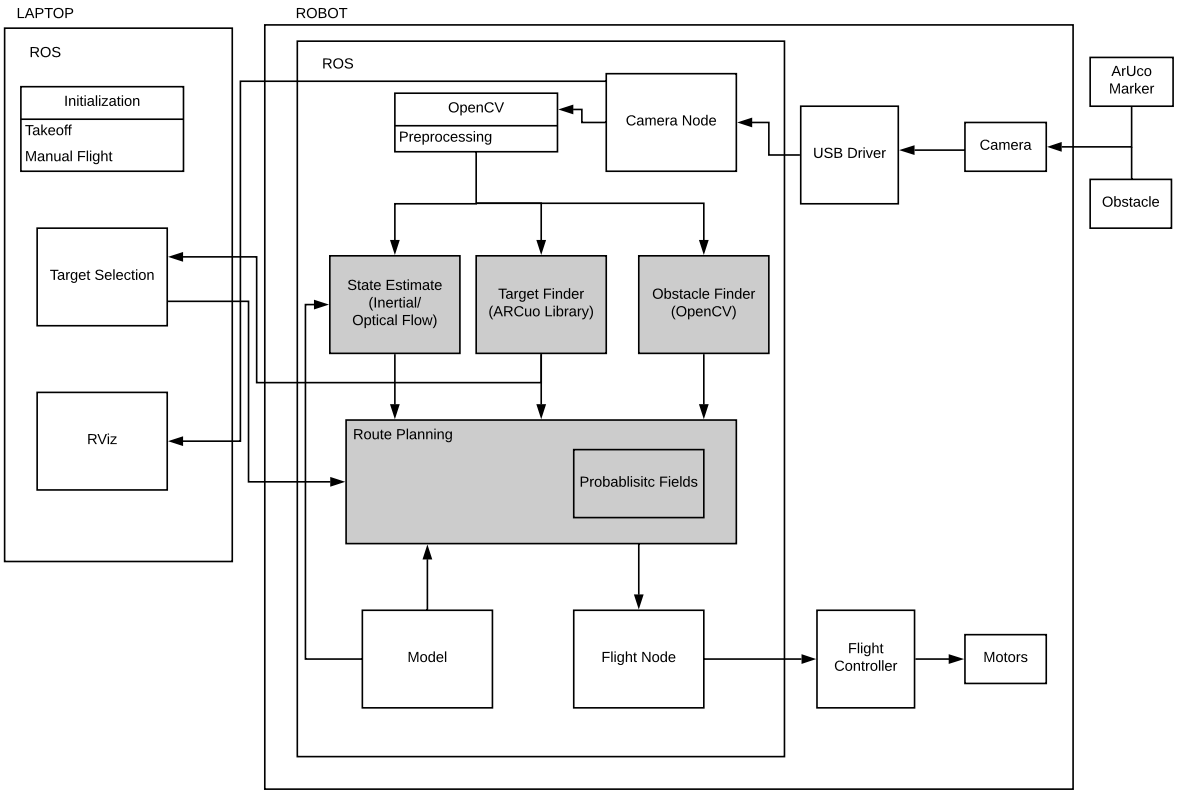


Figure 1: Project Block Diagram

2.6 Custom Nodes

All of the nodes being written for this project will be written in Python and then built in ROS to be implemented on the robot. Python is being used for this project because C++ and Python are the most supported languages for both ROS Nodes and OpenCV. While C++ could possibly be faster in execution, we believe that most of our bottlenecks will be in the actual processing of images. The library calls to OpenCV will still be executing in the lower level C that OpenCV is written in, so there would be minimal gains to be had writing in C++.

The Camera node is referenced in many locations. The camera node is built in to ROS and provides image data in various formats. Processes can to subscribe to different image formats. In general the raw image information will be used by the other nodes. Common transformations that must be applied to all functions these will be done in the common camera node to avoid unnecessary calculations.

2.6.1 Target Finding

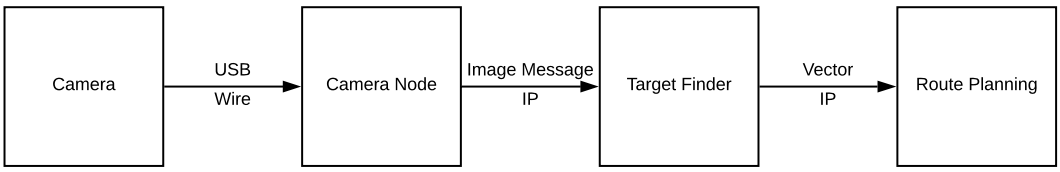


Figure 2: Target Finding Node

Camera data, after being processed by the camera node, goes into the target finder node. The target finder is mostly implementing the ArUco symbol finding libraries in OpenCV. ArUco symbols are intended for augmented reality purposes and provide very accurate position, orientation, and distance information. These values will be translated into a vector value that the other systems on the robot can understand and be published at least 10 times per second.

Given the limited field of view of the robot it may not always be able to see the target, in which case it will continue to scan for targets.

ArUco markers normally encode simple ID information. Vectors will carry the same identifiers allowing the route planning parts of the robot to know when a target has been re-spotted and over-write it or allow for finding and remembering the positions of multiple targets.

2.6.2 Obstacle Finding

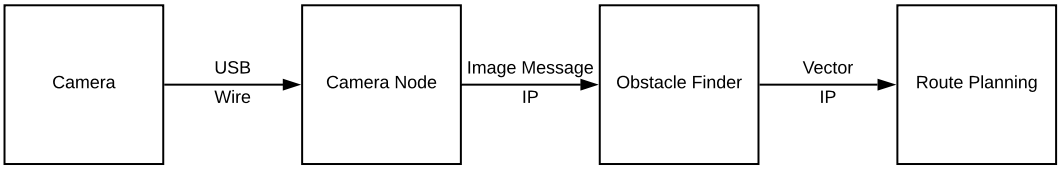


Figure 3: Obstacle Finding Node

Finding obstacles with depth information will be difficult using monocular vision. The current plan is to try to combine parallax shift with edge detection to try to parse information about nearby objects. These vectors will be published as a large ROS message at least 10 times per second.

Getting depth information from a single camera is difficult. The TurtleBot also has a suite of depth sensors built in with the camera so these may be used initially instead of trying to use parallax information from a camera to allow for testing other aspects of the system.

2.6.3 State Estimator

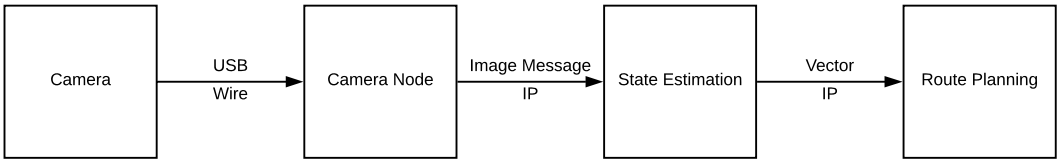


Figure 4: State Estimating Node

State estimating from camera information is very difficult and computationally expensive. The TurtleBot has stepper motors that can measure distances fairly reliably, and the Quadcopter should have accelerometers that provide changes in position more reliably than image information could.

Initially these simpler forms of measuring the robot moving through the environment will be used and transformations will be passed off to the route planning node which will adjust the vectors to nearby obstacles and targets according to how much the robot has moved.

New position estimates should be provided at least 10 times every second in the form of a transformation from the last position that the robot was in passed over a ROS message.

2.6.4 Route Planning

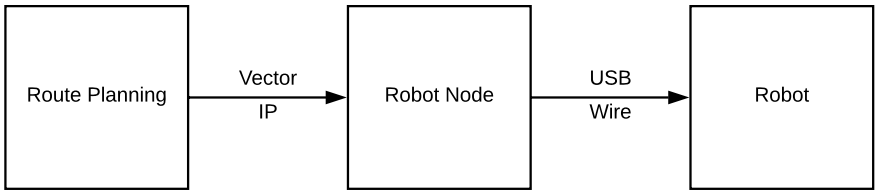


Figure 5: Route Planning Node

Route planning takes input from the other three custom nodes and determines a route that should carry it toward the target. The current proposed solution is currently to create a potential field where positive vectors that repel the robot are attached to all obstacles while a large negative vector is attached to the target location.

As the route node receives more information about the robot moving through the environment should apply transformations to all of the contained vectors to keep them up to date with where they would be as the robot changes position in the environment.

This will also require having a method to remove un-necessary or repetitive vectors. This will be easy to implement with the target vector as the targets will have identification values meaning that the current target can be verified. With obstacles this will be more difficult, and initially vectors will probably simply expire after a certain amount of time.

The route planning software should provide a direction vector to the robot controlling node 10 times every second. There is no need for the route to be the optimal route given that the environment is unknown.

2.7 Node Diagram

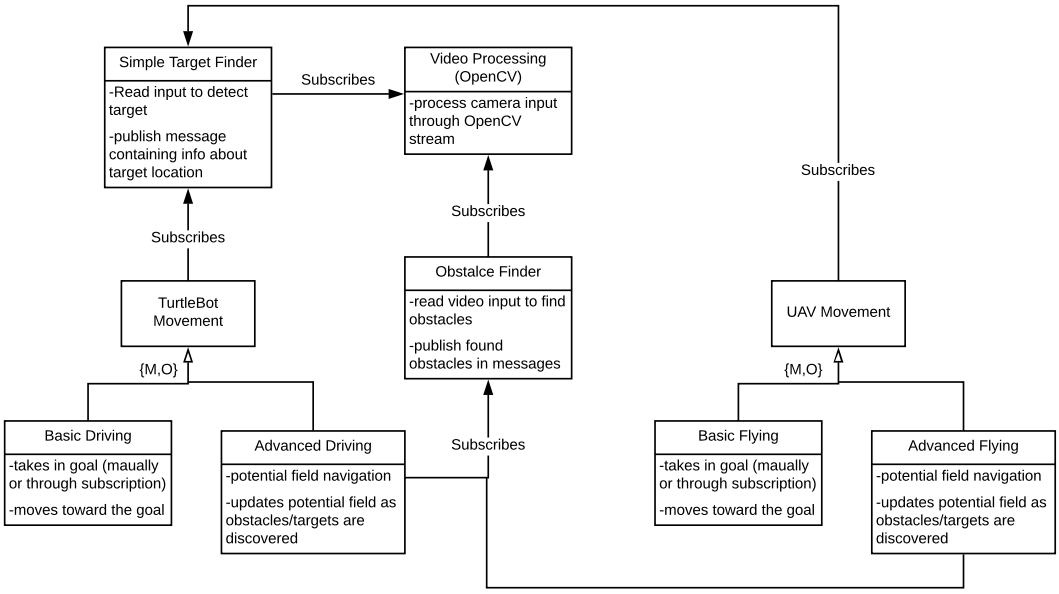


Figure 6: Project Node Diagram

Through Figure 6 it can been shown what nodes will be created or used for this project. The diagram displays what node will subscribe to the other nodes, such as the simple target finder node will subscribe to the video processor node to get messages about the target's potential location. Both the TurtleBot and the Flying Robot have two movement nodes, this is to first build a simple movement node on the platform for testing purposes, then the proper and final movement node will be made and implemented on the respective robot.

2.8 Mathematical Modeling

2.8.1 Flying Robot

Given that a quad-copter has six degrees of freedom, modeling them is a very difficult task. The AscTec Pelican already has many existing models intended for use with ROS. There is also a large collection of hardware agnostic tools for operating a quad-copter.

2.8.2 Robot in 3D Space

The robot itself will need to remember where it is located in its environment. Remembering where targets are located relative to the robot is necessary, and presumably remembering the locations of obstacles will be used as well. Using quaternion vectors relative to the robot will allow for simple transformations on these vectors while using very little computational power [10]. ROS contains methods and messages to easily communicate, convert, and apply transformations to quaternions.

2.8.3 Pose Estimation

Initially feedback from inertial sensors will be used for pose estimation, but ideally a version of Optical Flow would be used to estimate the position of the robot in 3D space. There are some libraries in OpenCV that have addressed generalized pose estimation in 6 degrees of freedom, but it requires a significant amount of computational power, a higher frame rate than our goal of 10Hz to be used for real time performance, and hundreds of visual odometry measurements must be made in every frame requiring a visually busy background.

2.9 Interfacing

The interfacing of almost all sensors with the on-board computer will be done over USB. Communication between different nodes on the same computer, and any computers off-board the robot, will be done exclusively with ROS messages sent over either internal loops or Wi-Fi.

2.10 Verification and Validation

2.10.1 Unit Tests

When using ROS it is typical to create individual nodes that all execute independent of one another. Using this modular design, it is easy to write an entire node and verify it individually before executing it with the entire system. Verification and validation of the project's nodes will be done prior to implementation in the Gazebo simulation environment.

2.10.2 Gazebo

Gazebo is a simulation environment that runs nodes that give similar input and output to nodes being executed on a real robot moving around in a real environment. This includes camera feeds, and allows for the insertion of obstacles and ArUco shapes. Testing of custom written nodes will be done in Gazebo congruently with unit-tests to ensure that nodes which rely on subscribing to other nodes are behaving as expected before executing code on an actual robot. Gazebo scheduled deadlines can be seen in Figure 7.

2.10.3 Camera Tests

Given that most of our custom nodes revolve around transforming camera data to extract information, the ArUco shapes for targets, parallax shift for obstacles, and optical flow for odometry, tests performed with only the camera and OpenCV outside of a node can be created and and executed without being contained within a ROS node. This will be useful as we can use OpenCV's feature highlighting capabilities to ensure that targets and obstacles are being identified while moving a camera through an environment.

2.10.4 Ground Based Robot

Testing the code on a ground based robot will be necessary to ensure that the logic in the movement nodes is sound. Initially simply having the robot follow moving targets or turn to face targets that are not head on will be sufficient. As obstacle avoidance gets developed ensuring that the robot does avoid the obstacles will be enough. If the project is sufficiently developed that it is possible to fine tune the parameters of the potential fields being used by the robot to avoid obstacles and move toward an obstacle it may be necessary to use motion tracking equipment already in the lab to verify that the robot is accurately executing instructions.

2.10.5 Flying Robot

Similar to the ground based robot, verifying the flying robot will primarily involve evaluating its behaviour. The behaviour we will expect is the all the functionality that was implemented on the ground

based robot previously. If more complex moves are attempted it may become necessary to test this platform using the motion capture equipment that exists in the lab.

3 Equipment

3.1 Equipment Table

A large number of items are being used in this project, most are arbitrary and may be changed in future iterations. Part way through our project we intend to re-implement the entire solution onto a different platform. A naming convention has been developed for all of our components using a three character code.

The first letter, ascending from A for hardware and descending from Z for software, represents the subsystem or general classification the equipment falls under. The second character represents a generic piece of equipment, and the third is for a particular example that we are using for our project. Any part with the same two first letters should be able to be substituted freely, an example may be two cameras. One camera (CAA) may be used on the Turtlebot as it is built in to the Turtlebot frame, while another (CAB) may be used on the flying robot as it is much lighter. The generic camera (CA-) can refer to any camera which provides the same information to the same nodes.

4 Schedule

Task Name	Duration	Start	Finish
COATIMUNDE	**144 days**	**Tue 09/10/18**	**Sun 28/04/19**
- Turtlebot	**97 days**	**Tue 09/10/18**	**Wed 20/02/19**
– Create a Basic Movement Node	12 days	Tue 09/10/18	Wed 24/10/18
– Subscribe to the Camera	8 days	Wed 24/10/18	Fri 02/11/18
– Camera and Video	**19 days**	**Tue 30/10/18**	**Fri 23/11/18**
— OpenCV able to process Video Stream	9 days	Tue 30/10/18	Fri 09/11/18
— Able to detect a Target	12 days	Tue 08/11/18	Fri 23/11/18
– Movement	**67 days**	**Tue 20/11/18**	**Wed 20/02/19**
— Move towards Target	9 days	Tue 20/11/18	Fri 30/11/18
— Avoid Obstacles and Move Towards Target	31 days	Thu 29/11/18	Thu 10/01/19
— Avoid Obstacles with Memory	32 days	Tue 08/01/19	Wed 20/02/19
- UAV	**53 days**	**Wed 13/02/19**	**Sun 28/04/19**
– Port to UAV	53 days	Wed 13/02/19	Sun 28/04/19
-Data Item Deliverables (DID)	**148 days**	**Thu 01/11/18**	**Thu 28/03/19**
– DID-4 Preliminary Design Specification	22 days	Thu 01/11/18	Thu 22/11/18
– DID-5 Design Review Presentation	15 days	Thu 15/11/18	Thu 29/11/18
– DID-6 Schedule Update	56 days	Fri 30/11/18	Thu 17/01/19
– DID-7 Detailed Design Document	63 days	Fri 18/01/19	Thu 21/03/19
– DID-8 Final Project Presentation	148 days	Thu 01/11/18	Thu 28/03/19

Table 2: Project Schedule

Purpose		Description		Equipment	
A	Ground Based Robot	A	Robot	A	Clearpath Robotics Turtlebot 2
A	Ground Based Robot	B	Robot Controller	A	Acer Aspire E-11
A	Ground Based Robot	C	Robot Base	A	Kobuki Robot Base
A	Ground Based Robot	D	Sensors	A	Stepper Motor Feedback
A	Ground Based Robot	D	Sensors	B	Orbbec Astra Pro Sensors
A	Ground Based Robot	D	Sensors	C	Gyroscope
B	Flying Robot	A	Robot	A	AscTec Pelican Quadcopter
B	Flying Robot	B	Robot Controller	A	AscTec Low Level Processor
B	Flying Robot	B	Robot Controller	B	AscTec High Level Processor
B	Flying Robot	B	Robot Controller	C	ODROID-XU4
B	Flying Robot	C	Robot Base	A	AscTec Pelican Frame
C	Camera	A	USB Camera	A	Orbbec Astra Pro Camera
C	Camera	A	USB Camera	B	oCam-1MGN-U Plus
C	Camera	B	Digital Camera	A	AscTec Option 4
D	Offboard Computer	A	Computer	A	Lenovo T520
D	Offboard Computer	A	Computer	B	Mathworks VM - ROS Gazebo v3
E	Testing Hardware	A	ArUco Symbol	0-9	Printed Single Digit ArUco Symbol
E	Testing Hardware	B	Physical Obstacle	A	Human
E	Testing Hardware	B	Physical Obstacle	B	Box
E	Testing Hardware	B	Physical Obstacle	C	Sheet
E	Testing Hardware	C	Motion Tracking	A	Reflective Dot
E	Testing Hardware	C	Motion Tracking	B	Motion Tracking Cameras
E	Testing Hardware	D	Camera Calibration	A	Printed 7cm Checker Pattern
F	Interfacing	A	Wireless Router	A	D-Link 2.4 GHz Router
S	Linux Operating System	A	Operating System	B	Ubuntu 18.04 Bionic
S	Linux Operating System	A	Operating System	K	Kubuntu (Mathworks) 14.04
S	Linux Operating System	A	Operating System	T	Ubuntu 14.04 Trusty
S	Linux Operating System	B	Bundled Program	A	Generic Loopback Device
S	Linux Operating System	C	Version Control	A	git
T	ROS Service - Custom	A	Action Services	A	Set Goal
U	ROS Service - Standard	A	Action Services	A	Trigger
V	ROS Message - Custom	A	Action Messages	A	Goal
V	ROS Message - Custom	A	Action Messages	B	Obstacle
W	ROS Message - Standard	A	Sensor Messages	A	Image
W	ROS Message - Standard	A	Sensor Messages	B	PointCloud
W	ROS Message - Standard	B	Geometry Messages	A	Twist
W	ROS Message - Standard	B	Geometry Messages	B	Quaternion
W	ROS Message - Standard	B	Geometry Messages	C	Transform
X	ROS Node - Custom	A	Vision	A	opencv_preprocessing
X	ROS Node - Custom	B	Movement	A	basic_movement
X	ROS Node - Custom	B	Movement	B	avoidance_movement
X	ROS Node - Custom	B	Movement	C	basic_flying_movement
X	ROS Node - Custom	B	Movement	D	avoidance_flying_movement
Y	ROS Node - Standard	A	Vision	A	vision_opencv
Y	ROS Node - Standard	A	Vision	B	image_pipeline
Y	ROS Node - Standard	B	Coordinates	A	tf
Y	ROS Node - Standard	C	Vision	B	image_pipeline
Z	Robot Operating System	A	Operating System	A	ROS Indigo Igloo
Z	Robot Operating System	A	Operating System	B	ROS Melodic Morenia
Z	Robot Operating System	B	Bundled Program	A	Gazebo2
Z	Robot Operating System	B	Bundled Program	B	Gazebo7
Z	Robot Operating System	B	Bundled Program	C	catkin
Z	Robot Operating System	B	Bundled Program	D	RViz
Z	Robot Operating System	B	Bundled Program	E	rosbag
Z	Robot Operating System	B	Bundled Program	F	rqt
Z	Robot Operating System	C	Client	A	roscpp
Z	Robot Operating System	C	Client	B	rospy

Table 1: Equipment

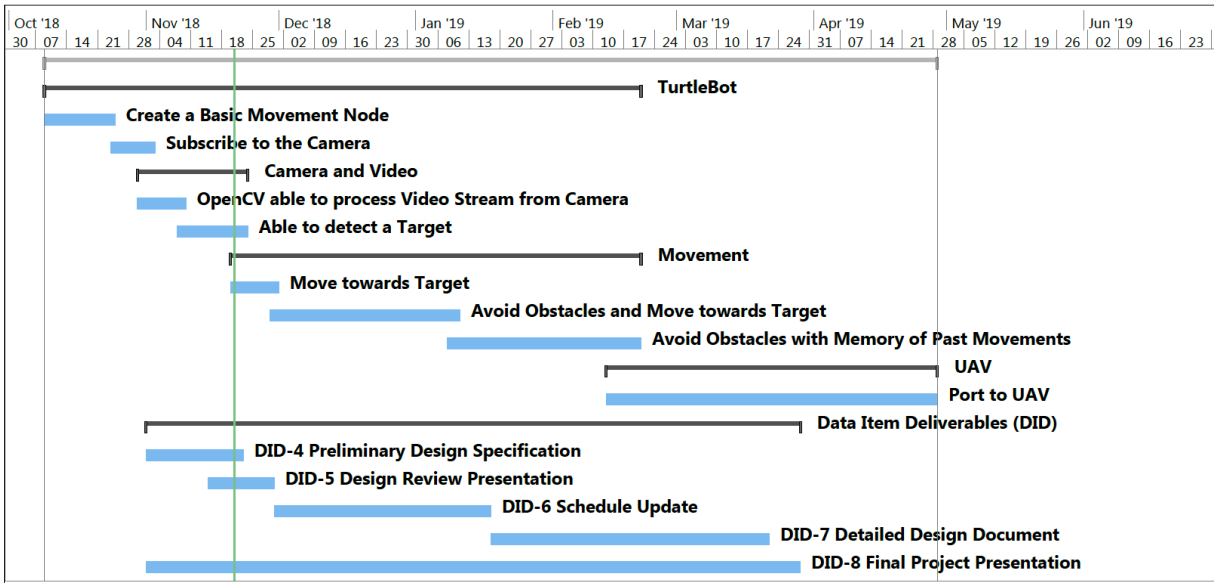


Figure 7: Project Schedule Diagram

The created schedule for the project shown in Fig 1 outlines the major milestones for the project, such as Data Item Deliverables and deadlines for completing different functionalities for the project. Porting to the UAV is listed as one task as we are not sure what problems will arise until we attempt to port the project.

5 Unresolved Issues and Risks

5.1 Limited Space

The scope of the project is limited due to only being able to test indoors. Ideally the UAV would be operating at a high speed and identifying an arbitrary target in an unpredictable environment. In our case though the testing must be completed within the confines of a relatively small robotics laboratory. Smaller obstacles and low speeds will be used indoors due to space constraints.

5.1.1 Likelihood

Guaranteed given that we are only able to test indoors without obtaining a significant number of permissions and licenses.

5.1.2 Impact

High impact, especially if using a flying robot, as the space required simply to start and stop moving must be considered meaning limited space for obstacles.

5.1.3 Process Solution

Lower speeds will be used, and most testing will be done with a ground based vehicle. Consider requesting access to a covered sports facility during an off period of time for final testing if more space is required.

5.2 Identifying Markers

The markers we intend to use, at least initially, are intended for use in Augmented Reality purposes. Tests of these shapes show that they are capable of being identified at many angles, but the reliability with which they can be identified decreases quickly as the viewing angle changes.

There is potential that markers which are reliably identified at low speeds and in simulation may not be detectable on a flying vehicle moving faster.

5.2.1 Likelihood

Both the TurtleBot and Flying Robot shall be traveling at lower speeds, therefore the likelihood of this risk is low.

5.2.2 Impact

High impact, the primary task that our robot must accomplish is moving toward a visual target. If the robot is not able to reliably identify a marker then there will be no way to verify its capability.

5.2.3 Process Solution

Alternatives to the ArUco markers can be tested, or if these prove truly unreliable then coloured areas or illuminated targets could be incorporated. These are less desirable as ideally the robot would be able to fly toward an arbitrary target.

5.3 Computer Hardware Limitations

Image processing, especially quickly and with multiple goals, is computationally expensive. While this should not be an issue on a ground vehicle moving at slower speeds, it may become more of an issue on a Flying Robot if it operates at higher speeds and is incapable of carrying as much on-board computational capability.

5.3.1 Likelihood

High likelihood due to the amount of computational power required to do image processing.

5.3.2 Impact

Medium impact, presumably we would still be able to prove our systems on the ground robot which is capable of carrying as much processing power as needed. It may also only be a limitation at higher speeds or may impose limits on what the Flying Robot is capable of identifying and tracking.

5.3.3 Process Solution

Testing many algorithms and working to streamline the algorithm used, especially for the Flying Robot, as well as selecting hardware that is complimentary to the kind of loads created by running such an algorithm can mitigate these risks. Using a Field Programmable Gate Array (FPGA) could aid with this problem.

5.4 Flight Control System Inaccuracy

Many UAV's are not able to execute arbitrary movements with high accuracy. Verifying that the flying robot is actually executing instructions may be difficult.

5.4.1 Likelihood

Medium likelihood, depending on how well developed the libraries for our particular flying robot these issues may have all been sufficiently worked out for the purposes of this project.

5.4.2 Impact

Low impact, we could still observe the Flying Robot making decisions even if it is unable to execute the instructions given properly.

5.4.3 Process Solution

Using smaller number of obstacles and moving at a slow enough speed should ensure that the robot never needs to execute extreme moves. These more extreme moves are where inaccuracies in control models will exposes themselves the most.

6 Conclusion

The preliminary design has been shown for the Flying Robot obstacle avoidance system. The background and requirement defining activities have been given to lay the groundwork and a basic understanding of the current research in this domain. This document will be used and referenced for the rest of the design and building processes for the project. The design laid out in this document will also be used in the the Preliminary Design Review and Detailed Design Document.

References

[1] *Royal canadian air force | news article | update and new name for the joint unmanned surveillance target acquisition system (justas) project*, Jul. 2018. [Online]. Available: `http://www.rcaf-arc.forces.gc.ca/en/article-template-standard.page?doc=update-and-new-name-for-the-joint-unmanned-surveillance-target-acquisition-system-justas-project/j9u7rzyf`.

[2] A. J. Barry and R. Tedrake, "Pushbroom stereo for high-speed navigation in cluttered environments", in *Robotics and automation (icra), 2015 ieee international conference on*, IEEE, 2015, pp. 3046–3052.

[3] A. J. Barry, P. R. Florence, and R. Tedrake, "High-speed autonomous obstacle avoidance with pushbroom stereo", *Journal of Field Robotics*, vol. 35, no. 1, pp. 52–68, 2018.

[4] P. R. Florence, J. Carter, J. Ware, and R. Tedrake, "Nanomap: Fast, uncertainty-aware proximity queries with lazy search over local 3d data", *arXiv preprint arXiv:1802.09076*, 2018.

[5] *About - opencv library*. [Online]. Available: `https://opencv.org/about.html`.

[6] M. Wise and T. Foote, *Rep: 119 - specification for turtlebot compatible platforms*, Dec. 2011. [Online]. Available: `http://www.ros.org/reps/rep-0119.html`.

[7] *Asctec pelican - uas for computer vision and slam*. [Online]. Available: `http://www.asctec.de/en/uav-uas-drones-rpas-roav/asctec-pelican/`.

[8] Y. K. Hwang and N. Ahuja, "A potential field approach to path planning", *IEEE Transactions on Robotics and Automation*, vol. 8, no. 1, pp. 23–32, 1992.

[9] S. A. Bortoff, "Path planning for uavs", in *American Control Conference, 2000. Proceedings of the 2000*, IEEE, vol. 1, 2000, pp. 364–368.

[10] B. Williams and I. Reid, "On combining visual slam and visual odometry", in *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, IEEE, 2010, pp. 3494–3500.