Final Report

December 15th, 2022

Team Sponsor: **Alexander (Allie) Shenkin**

Team Mentor: **Daniel Kramer**

Team Instructor: **Michael Leverington**

Document produced by

Austin Malmin, **Charles Saluski, Conrad Murphy**, **and ShanHong “Kyle” Mo**

Table of Contents

Introduction

In the world of automation, normal mundane tasks are being replaced by artificial intelligence that can perform these jobs at higher efficiency and with better accuracy. The world is starting to see a breakthrough in the field of robots and drones that perform tasks autonomously. Today, humans are currently performing tasks that can easily be automated with self-guided drones. Search and rescue missions can prove to be dangerous in places not easily navigated by humans, such as collapsed caves, house fires, war zones, etc., but a drone could make these missions safer and more efficient. Amazon has already taken use of autonomous drones in their delivery of goods throughout the United States and the United Kingdom. Drones can dust crops with greater precision and cheaper cost than planes piloted by humans. Our focus is the use of drones to aid in the study of rainforest ecosystems.

Mapping the upper and lower forest canopy currently requires a researcher to manually move a ground-based sensor through the forest which is time-consuming and difficult. The sensors used are Light Detection and Ranging (LiDAR). LiDAR sensors measure the time for a laser beam to return once fired to determine the distance of the object from the sensor and create a 3D model of the environment, whether on land or on the seas. Ecologists perform research focused on how forests around the world respond to climate change and simulate the effects certain changes in the climate would have on the environment. Team Mockingbird is working with Dr. Allie Shenkin to build an autonomous drone navigation system.

Our vision for Project Glasswing is a system with the end goal of guiding a drone through the rough forest environment quickly. We will not work with a drone ourselves, but future work may be done to mount the system to a drone. The system will use a LiDAR sensor for mapping its surroundings and saving it to permanent storage for future study, while other sensors will serve the role of object detection and avoidance to avoid crashing into branches and vines. A successful project will drastically improve Dr. Shenkin’s research on the effects of climate change on our rainforests.

Process Overview

The team followed the AGILE progress of development where we have a sprint bi-weekly where we worked on the project before showing our progress to the sponsor to discuss if the product is inline with what the sponsor desires.

This project involves assembling a functional system out of both hardware and software. Below we provide a brief description of useful hardware components and open-source software. This project will not have a physical drone available for the team to use as the team is doing the navigation system for the drone.

Hardware Components

Raspberry Pi: A single-board, Linux-based computer. It contains general-purpose I/O pins for wiring electrical components, as well as USB ports for plugging in USB devices. In the future, the Raspberry Pi can interface with the drone's onboard computer (likely an Arduino), but this is outside the scope of this project.

RPLIDAR: A brand of LiDAR sensor. LiDARs capture data using light. As for the specifics, the LiDAR will continuously send out thousands of laser points outwards in 360 degrees by rotating. The laser point will then hit an object and reflect to the LiDAR which will be captured and will be stored in the Rplidar application in the form of a black dot in a XYZ plane (X is front/back, Y is the sides and Z is the up).

Intel RealSense Depth Camera: The depth camera is a type of close-range camera that uses multiple cameras to capture the output such as Infrared that helps users to see in the dark and RBG (Red, Blue and Red) camera that captures visible light. The output will be a color spectrum that represents distance, the further it is, the "hotter" the color, blue is when it is close and red when it is far. The camera measures around 6 meters and the team will use this camera for obstacle avoidance for any object that is 6 meters or less.

Software Libraries

ROS (Robot Operating System): Despite the name, ROS is not an operating system which runs a computer. Instead, it is a framework which supports various software libraries and tools for the purpose of building robot programs. These tools include drivers, which allow the user to interface with hardware components, and libraries, which make data processing more accessible to the user.

RPLIDAR: A library, supported by ROS, which interfaces with the physical RPLIDAR sensor. The software retrieves the point data from the sensor and translates it into usable data (tuples of XYZ coordinates in 3D space). The points can then be plotted onto a map.

HectorSLAM: SLAM (Simultaneous Localization and Mapping) is a process which procedurally builds a map given the position of the observing robot and the surrounding points observed. HectorSLAM is one library which performs this process. We will use a SLAM library to produce the map of the surrounding rainforest.

Requirements

The system's architecture is the main focus of our design. Our architecture is largely determined by the physical constraints of our project, given that our system will be designed to be installed on a drone. Features of the architecture include exterior sensors, software which receives data from the sensors, mapping and object detection libraries, and software which writes to external storage.

Key Responsibilities and Features

Our key responsibilities include minimum requirements, which are those needed to make a minimum viable product, and stretch goals, which are not specified as requirements but would be useful for our sponsor.

Minimum Requirements

In this section, we list three basic functions that the drone must be able to perform, then break these functions down into specific instructions and algorithms within the software. The minimum requirements include: (1) automatic detection of objects, (2) move towards a destination, (3) output a direction in which to move.

Automatic detection of objects: This is the most important minimum requirement as this is the starting point of the project since detection of objects as a requirement will be used in other requirements.

Per our starting assumptions, the navigation system will use a LiDAR sensor. In our case, the RPLIDAR A1 has an angle of about 1 degree between each beam. Even though the degree difference between beams seems small, as the distance that the LiDAR needs to scan increases, the distance between each beam also increases. As a result, the LiDAR might miss small objects positioned in between pulses. That is why it has a guaranteed data accuracy for a range of 6 meters, even though it can scan up to 30 meters but with less certain accuracy as detection range increases.

When a laser pulse strikes an object, it will reflect back to a receiver within the sensor. A point in space is generated based on the time that it took for the photons to arrive back to the receiver. For example, if it took the photon 0.01 seconds to come back to the receiver while another photo took 0.02 seconds to come back, then the 0.01s point will be closer to the LiDAR and the 0.02s point will be twice as far away. Each point is stored within a point cloud in the form of its XYZ coordinates. Currently, we are using a 2D LiDAR, which means that even though it only scans the XY coordinates, it will output the XYZ coordinates with the Z value set to the default value of 0.

This computer will have a vision library which performs SLAM, called HectorSLAM. This vision library takes in data from the sensor's receiver and calculates the position of each point, accounting for how much the LiDAR sensor has moved from its starting position. For example, if the LiDAR moves north, HectorSLAM updates the LiDAR's position

based on how far north it has moved since it was powered on. If it begins moving east, the library offsets the LiDAR's position east as well. Point clouds are displayed on a map with a fixed size. In summary, all systems (sensor, vision library, and ROS) will function together to create one instance of a map. Every time the LiDAR sweeps across all 360 degrees of space, a new map is created. These map instances are all overlaid onto one main map.

The vision library and the sensor (RPLIDAR A1) mentioned above will also need to work with ROS as well as the computer that the team is using. ROS interacts with the Linux computer via the command line. There are commands to launch ROS, launch certain packages and libraries within ROS, etc. Some of the vision libraries also open GUIs which can be used to control certain display options.

Other than the details already mentioned with the interaction between ROS and LiDAR with the vision library, it will also remember the maps. The system will remember details about places visited previously, so it will register certain clusters of points as objects. This data is very crucial, as it will support automatic detection of objects and eventually play a role in creating safe routes throughout the environment.

Move towards a destination: While detecting and mapping the surrounding environment, the system must have goal-oriented movement towards a specific direction. This means the drone must be capable of making overall progress in a certain direction. The drone may take one step back to get out of dead ends as long as it can move two steps forward later on.

Using the point cloud created, there will be a method in the ROS that will automatically create a path to the destination. Since the map has a point cloud that is default colored black, it also means there are white spaces that aren't part of the point clouds that represent empty space. The goal is to stay within the white space and avoid black points.

Our system will take a greedy approach to move in the specified direction. The drone will make as much progress as it can in the preferred direction, as long as it doesn't detect obstacles that cannot be avoided without backtracking. This means that as long as it doesn't detect an obstacle, it will have permission to continue to move in that direction.

If the drone does reach an obstacle around a cm, our system will have a function that will find whether to move left or right based on point clouds generated and calculated by finding a path that has enough space for the drones to travel. And it will continue to do so until it reaches a dead-end, meaning it can't go forward or to the sides and can only go backward. Because the system remembers the path that it took previously, the drone will go back to the previous area where it made a decision to go a certain path using a maze-solving algorithm such as "right-hand rule" algorithm. For example, if previously encountered an obstacle where it can go right and left and the system chose "right", this time it will go back to that place and will choose "left". We will set that if there are two options of right and left, the system will always pick right first. If the previous section only had one choice of left and it already went left, it will go back to the previous section before this section that it made a choice, it will continue to make this decision until it reaches the destination that it was meant to go.

Output a direction in which to move: Our system must periodically output a direction in which to move. This direction will be defined as an angle relative to the drone's current facing direction. The direction can be further generalized into a simple Forward, Left, Right, and Backward based on which quadrant the angle occupies.

Drones running ROS have a pre-existing interface to fly the physical drone in a specified direction. This interface contains functions that, when called, fly the drone in a certain direction for a certain amount of time. The fine-tuning of how fast each motor should spin in order to move in any given direction will already exist on the drone. This means we do not need to invest time into developing the movement algorithms ourselves. Therefore, our directional output will be left as a simple print to the command

line. Human operators will move the platform in the direction the system wants to move in order to simulate drone flight. If the platform can be moved throughout the space according to our system's orders without crashing into obstacles, then that is an indication that our project has succeeded. In the future, when the scope of the project broadens to incorporate a real drone, our simple output can be translated into a function call to the drone's interface.

Stretch Goals

Following the completion of the core requirements, the following three stretch goals are listed in feasibility from most to least. (1) Store the 2D map onto an external SD card. (2) The drone will display a planned path to the user before actually doing it so the user can interrupt it in the case that is not what the user desired. (3) Giving a designated area to the drone, the drone has to be able to avoid crashing into obstacles and visit as much 3D space as possible to get a detailed map of the area.

Store and display the area visited: As the drone moves throughout the environment, it will be stitching together a map of the environment by plotting the point clouds given from the LiDAR. This process can be viewed in real time via HectorSLAM's graphical interface. Once the flight is complete, the map from HectorSLAM will be complete. This map will then be saved and written to the SD card.

Show a display screen of the planned path: Since the system can already detect an object and determine if it can move left or right depending on if the open space is big enough for the drone to navigate to, all we have to do is have a function that can draw all possible paths that the drones can take by measuring the open space. All this function will do is return the possible path that the drone can take to the user's screen. When this is completed, the team will add a function that can override the system to allow the user to determine the direction for the drone to move to the user's preference, which is considered a beyond stretched goal.

Visit as much of the designated 3D space as possible: After giving the drone commands, with the ability to detect objects, the drone will move throughout 3D space by avoiding objects and stopping at the point that it was designated. Due to limitations on our budget, this will only happen in a 2D plane as the Z coordinate will always be 0. If the drone detects an object, it will stop and the user has to manually give commands to make it move around it.
The way that the drone will visit as much 3D space as possible is a working progress but the idea is that based on the previous requirement of showing a display screen of the planned path, we will use the map where the black/red dots are point cloud that the drone can't access and the knowledge of the space that is big enough for the drone to go through, the system will have a flag saying true or false for if a path has already been crossed through. If it has, we will probably color the path so that the user can look and tell that drone already looks through this area and will then go and continue to go to the path that hasn't been colored red yet.

Architecture and Implementation

Main Communication Mechanisms

Communication within the system will largely involve interfacing between different libraries. The ROS framework facilitates the communication between these libraries by providing device drivers and supporting the libraries within the platform itself for easy use. In other words, ROS has built-in support for sensor data, the libraries which we use, and the communication between sensors and libraries. This means we do not need to handle the majority of communication ourselves.

Communication and Control Flows

Inside of a complex system of software and hardware there must be some logical flow of control in order for the system to work as intended. Figure 1 displays these control flows below. The system starts by receiving point clouds from our input stream. As these are being collected they are passed through ROS to be forwarded to the

computer vision libraries. The detected objects are passed back to ROS where it will instruct the drone to move to avoid obstacles and attempt to plot a long term flightplan. If there are no paths towards the desired goal the drone will stop, land and report the issue to the pilot. Our drone will be extra cautious when moving as the most important thing is that the drone does not crash. With that being said we will be adding extra padding to the size of the drone to ensure we are flying safe. All of these packages and processes will live on a UNIX based system. This enables the piping of output from one file to another.

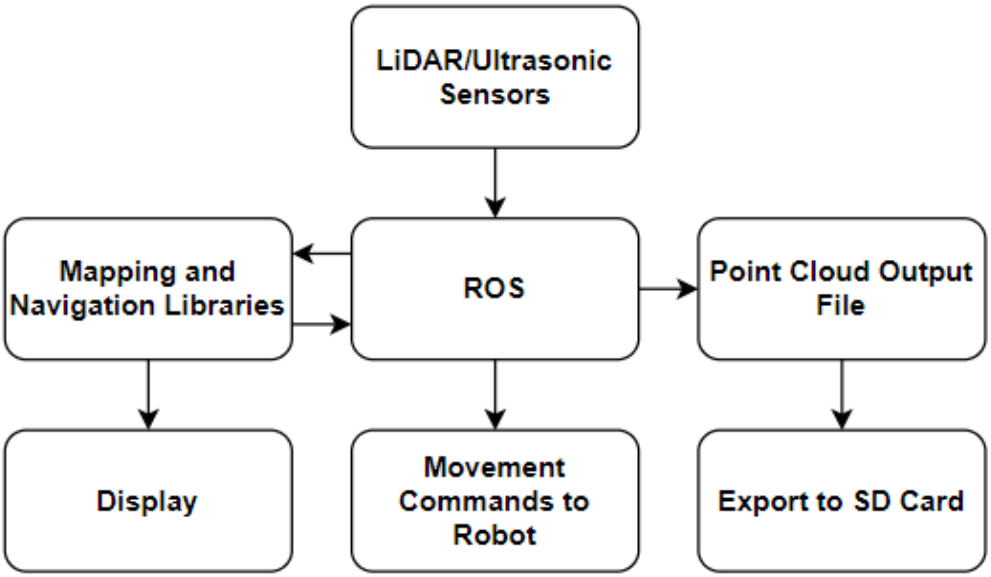

Figure 1: This diagram symbolizes how information will flow to and from our system.

Our system will have three primary modules: (1) a SLAM component which utilizes a LiDAR sensor; (2) an object detection and avoidance component; and (3) an optional algorithm which attempts to pathfind to unexplored regions in the space.

LiDAR SLAM and Map Storage

One component in our system will be LiDAR data detection with SLAM. Its purpose is to interface with the LiDAR sensor and receive point cloud data from the drone's surroundings, then plot the points on a map in 3D space. Over time, this map will be

populated with millions of data points. This map will then be written to permanent storage for future use.

Figure 1 depicts the flow of activity in our system with regards to LiDAR data collection and storage. Our LiDAR sensor collects point cloud data from the surrounding environment. This information is sent to ROS to be accessed by other libraries. A LiDAR library can retrieve the data in a usable format (tuples containing XYZ coordinates). This data can then be used in a SLAM component to plot these points in 3D space. With all the points being plotted, a visualization program can output the plotted points to a user display, allowing the user to see the map being generated in real time. At the same time, the points can also be written to the onboard SD card for permanent storage.

The user interface for this component would include a display screen of the map as it is generated in real time. Some of ROS's vision libraries, RPLidar and RViz, have built-in support for plotting LiDAR data. One of our stretch goals is to also display the path that the system wants to take, but doing so may require us to plot points on our own RViz display from scratch without help from the built-in vision libraries, which would be impractical.

Module 2: Camera Object Detection and Avoidance

Another component in our system will involve object detection and avoidance. Its purpose is to interface with an Intel RealSense depth camera, receiving data on nearby objects and hazards and giving directions on how to move around them.

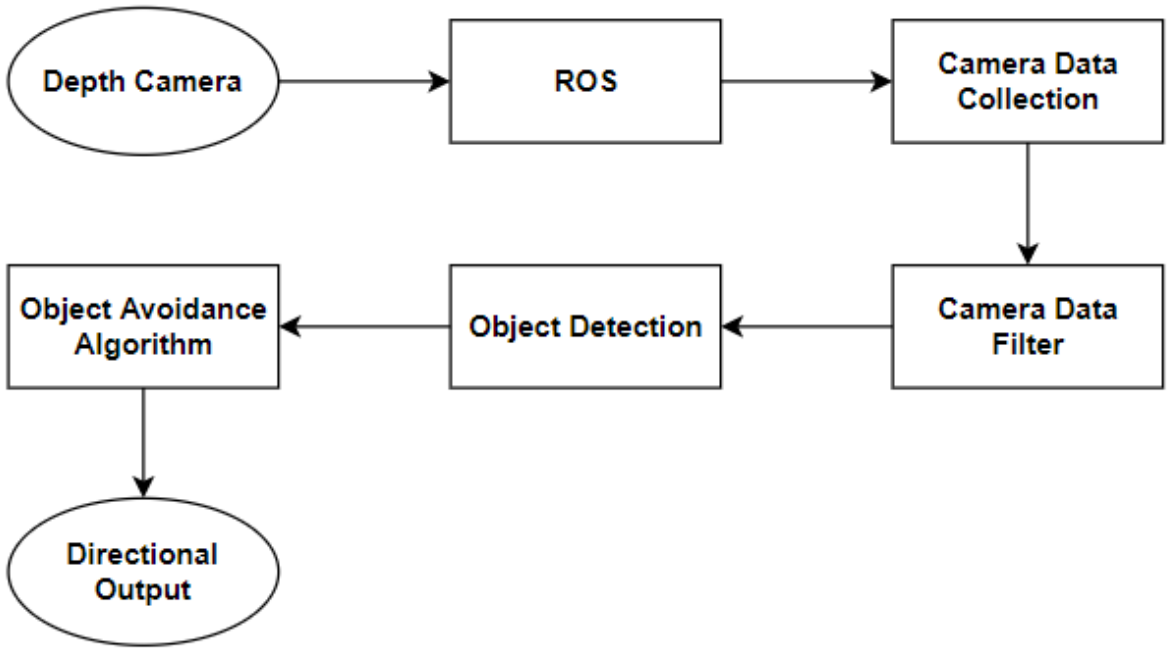

Figure 2: The flow of activity within our system, with regards to camera data collection and object avoidance.

As displayed in Figure 2, the depth camera outputs tuples of 3D coordinates to ROS. There is a massive amount of output from the camera, so the output needs to be filtered down to focus on only relevant data. This filter keeps only the points that are close enough to the drone to be a potential hazard, throwing out the points that are too far. With this filtered camera data, an object detection library can recognize objects based on clusters of data points, then make a 3D perimeter around them. Our system will then produce a directional output to dodge these objects. The movement of our drone will be a greedy algorithm which attempts to move forward as far as possible until forward movement is blocked, then backtracks from there.

Since the drone will be an arbitrary size, the user will need a way to input the size of the drone so that the system knows which gaps the drone can fit through and which ones it cannot. Otherwise, our camera module does not need to output data to a user. Intel

RealSense supports a visualization program which displays data in real time, so this is an option if the user does want it.

Module 3: Visitation of All Possible Space

One of our stretch goals is to implement an algorithm component which commands the drone to visit as much space as possible within a specified boundary. The LiDAR map is generated over time, with solid points and empty space being plotted in 3D space. This module would attempt to visit nearby areas which have not yet been plotted, then set a goal to move in a general direction towards that point.

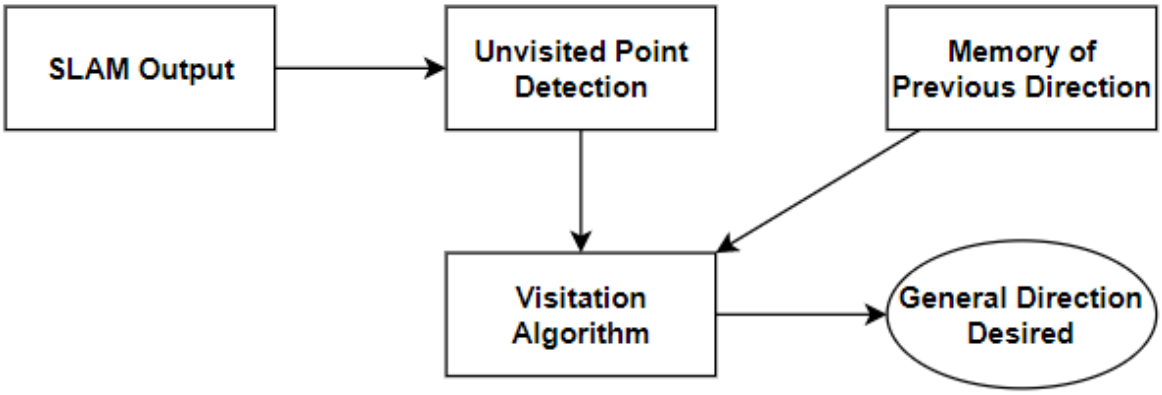

Figure 3: The flow of activity within our system, with regards to the visitation algorithm.

Figure 3 describes our visitation algorithm. Given our SLAM output, our code would find the nearest region on the map with a significant amount of unplotted space. The algorithm would then output a direction to go in order to visit the space. Our algorithm must be robust enough to avoid getting caught in a loop–for instance, hitting a dead end on the left, attempting to explore over to the right instead, then hitting another dead end and attempting to go left again. This means the algorithm will need to remember where the drone has attempted to move in order to visit the target space. With these parameters in mind, the algorithm will output a general direction in which to move; this directional output will be passed through object detection so that the drone can move there while avoiding objects.

The user will need to input the boundary of the space which needs to be visited. This will be done by inputting the coordinates of two corners on the boundary. As with the visual output of module 1, this component could also display the space which the drone is attempting to visit, as well as the overall path it intends to take; however, we are unsure if this will be feasible.

Implementation

Our team has largely completed the minimum requirements for this project. With this being said, there is still some work to do. Figure 4 provides a high level overview of these parts. Inside the figure is a line indicating current progress on the project, items to the left of the line are complete and the right are still action items left unfinished. Which is then broken down into the following sections: setting up the Raspberry Pi, designing the software, building the physical product, developing proper tests to test the product, and showcasing our final products and results of our testing.

Setting Up Raspberry Pi

Our project assumes a Linux environment to ensure compatibility with the drone. In previous iterations of this project, we utilized a virtual machine running ROS to simulate

a Linux-based system. From here on out, our team will be utilizing a Raspberry Pi to make this system more portable.

Software Design

With our environment set up and ready to go, our next step will be to utilize it to its full potential. We will be designing a system in which the hardware can communicate with the software seamlessly in this step. In order to accomplish this, we will design a plan for our code base to live on the Raspberry Pi. As this project will be picked up by another team at a later point, it is crucial that we maintain excellent documentation of how our system will function.

Building the Product

As stated above this project is not just software but also hardware. This means we will need to physically build our system as well as develop the software. The hardware components of the project will be mounted to a surface similar to a drone. This is also the stage where the code will be created and compiled into a surface which can be easily utilized for whoever comes next to interact with this project.

Planned Product vs. Built Product

In terms of planned product, it was changed many times throughout the project. At first, we thought we were building software for drones so the team planned to have a large budget to have 3D LiDAR and to use softwares that is compatible with it. After learning that we have limited budget to the point where we can only use 2D LiDAR, we decided to use ROS1 where it works best with the 2D LiDAR. After realizing that we are not using a drone, we started to tackle and test out virtual environments with ROS2 with a virtual robot as the team no longer needed to focus on the hardware and more on the software. So overall, the system we build from what the team planned is the same in terms of software but originally the final product is supposed to show that the software is working on a proper drone and a 3D LiDAR instead of a virtual environment.

Testing

Unlike other teams, our product involves both hardware and software, meaning that we have to test both. For hardware testing, you just need to make sure that it works as intended such as powering on/off as well as making sure nothing is broken. As for hardware testing with software, such as downloading data into it or testing to see if it receives the data, this requires the team to manually test every software with the hardware, through this we are able to deduce that some hardware is fake or it lacks the capacity to handle the data coming in. This can be seen from the fake memory chip that we received when we try to put data or memory into it. Also using this method, we found that any saved maps or simulations require memory beyond what we bought. As for software testing, we have to concatenate multiple software together, meaning that the team has to test one at a time to see if the software works as intended, some example that this method helped was outdated software as it contains libraries that are unusable to the current software we are using. This is why we moved from ROS1 to ROS2 more than halfway through the second semester.

Project Timeline

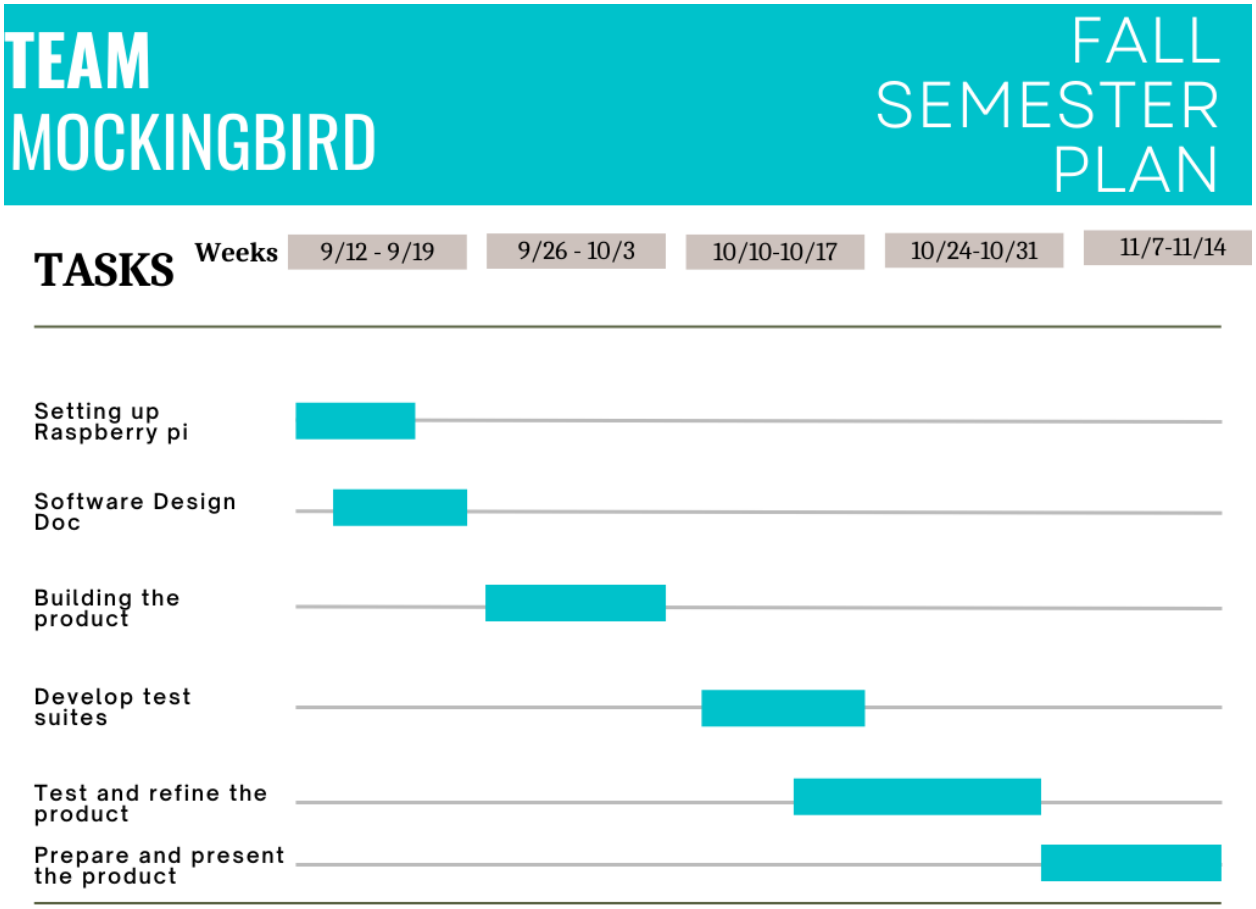

Figure 4: A Gantt chart detailing the plan for implementing the project in its final stages.

Future Work

Like all great projects, it should be able to be expanded upon by future works. For Project Glasswing, it is split into two sections – the software-focused component and the hardwar- focused component, where one team is designated to focus on one of them. Our team is the software focused team, our project is needed in order for the second team to succeed, meaning that our project should be easy to understand and easy to make changes to. With our project, the hardware-focused team will be able to expand upon what we have done and make a drone be able to fly within a proper environment.

Conclusion

Autonomous drones are a major advancement in many different fields of study and service, including the study of intricate details in the rainforests. Dr. Alexander Shenkin is a researcher and climatologist who is attempting to create detailed maps of plots of forested land. His current workflow is slow and tedious, requiring large investments in time and manpower in order to create a map which omits crucial data. This approach is unwieldy and unsatisfactory, and a better solution is needed.

Our mapping system will greatly improve Dr. Shenkin's workflow in his study of the rainforest, allowing for faster and more efficient data collection from our ecosystems. This data will help us understand our environment, revealing important conclusions such as the effects of climate change and the mitigation of carbon in the atmosphere.

Once our system can automatically detect objects, move towards a destination, and output a direction to move next, we will continue to test available hardware. If we fulfill our goals, we will have a powerful module for automated navigation and mapping which could serve as the foundation for other data analysis tools. Dr. Shenkin could mount the drone with other types of sensors that measure light, humidity, heat, and other environmental factors, allowing for a more complete understanding of the environment's structure and function. Thanks to the drone's maneuverability, this comprehensive data would be quickly and easily obtainable. It would eliminate the trouble of sending a workforce of researchers to map by hand, saving hundreds of hours of labor.

Appendices

Hardware

Raspberry Pi: A single-board, Linux-based computer. It contains general-purpose I/O pins for wiring electrical components, as well as USB ports for plugging in USB devices. In the future, the Raspberry Pi can interface with the drone's onboard computer (likely an Arduino), but this is outside the scope of this project.

RPLIDAR: A brand of LiDAR sensor. LiDARs capture data using light. As for the specifics, the LiDAR will continuously send out thousands of laser points outwards in 360 degrees by rotating. The laser point will then hit an object and reflect to the LiDAR which will be captured and will be stored in the Rplidar application in the form of a black dot in a XYZ plane (X is front/back, Y is the sides and Z is the up).

Intel RealSense Depth Camera: The depth camera is a type of close-range camera that uses multiple cameras to capture the output such as Infrared that helps users to see in the dark and RBG (Red, Blue and Red) camera that captures visible light. The output will be a color spectrum that represents distance, the further it is, the "hotter" the color, blue is when it is close and red when it is far. The camera measures around 6 meters and the team will use this camera for obstacle avoidance for any object that is 6 meters or less.

Toolchain

ROS (Robot Operating System): Despite the name, ROS is not an operating system which runs a computer. Instead, it is a framework which supports various software libraries and tools for the purpose of building robot programs. These tools include drivers, which allow the user to interface with hardware components, and libraries, which make data processing more accessible to the user.

RPLIDAR: A library, supported by ROS, which interfaces with the physical RPLIDAR sensor. The software retrieves the point data from the sensor and translates it into usable data (tuples of XYZ coordinates in 3D space). The points can then be plotted onto a map.

HectorSLAM: SLAM (Simultaneous Localization and Mapping) is a process which procedurally builds a map given the position of the observing robot and the surrounding points observed. HectorSLAM is one library which performs this process. We will use a SLAM library to produce the map of the surrounding rainforest.

Setup

ROS2 Navigation2 Instructions

1. Start with a fresh Ubuntu 20.04 installation.
2. Open a terminal to run the following commands in.

```
wget
https://raw.githubusercontent.com/ROBOTIS-GIT/robotis_tools/master/in
stall_ros2_foxy.sh
sudo chmod 755 ./install_ros2_foxy.sh
bash ./install_ros2_foxy.sh
sudo apt install ros-foxy-gazebo-*
sudo apt install ros-foxy-cartographer ros-foxy-cartographer-ros
ros-foxy-navigation2 ros-foxy-nav2-bringup
source ~/.bashrc
sudo apt install ros-foxy-dynamixel-sdk ros-foxy-turtlebot3-msgs
ros-foxy-turtlebot3
echo 'export ROS_DOMAIN_ID=30 #TURTLEBOT3' >> ~/.bashrc
source ~/.bashrc
mkdir ~/colcon_ws
mkdir ~/colcon_ws/src
```

```
cd ~/colcon_ws/src
git clone -b foxy-devel
https://github.com/ROBOTIS-GIT/turtlebot3_simulations.git
cd ~/colcon_ws && colcon build --symlink-install
echo "export TURTLEBOT3_MODEL=burger" >> ~/.bashrc
```

3. Once install has been completed successfully run the following commands in separate terminals

```
cd ~/colcon_ws
ros2 launch turtlebot3_gazebo turtlebot3_world.launch.py
ros2 launch turtlebot3_cartographer cartographer.launch.py
use_sim_time:=True
ros2 launch turtlebot3_navigation2 navigation2.launch.py
use_sim_time:=True
```

ROS2 instructions

To install ROS2 on Ubuntu 20.04

- https://docs.ros.org/en/foxy/Installation/Ubuntu-Install-Debians.html

- Installing RPLIDAR
  - Sudo apt install ros-foxy-rplidar-ros
  - https://index.ros.org/r/rplidar_ros/
- Installing Realsense
  - https://github.com/IntelRealSense/librealsense/blob/development/doc/distribution_linux.md
- Looking for ROS integration
  - https://github.com/IntelRealSense/realsense-ros/tree/ros2-511
    - Error: https://github.com/IntelRealSense/realsense-ros/issues/1408
    - To fix: https://github.com/IntelRealSense/realsense-ros/issues/1408#issuecomment-698128999

Guide to run the system:

Step 1: Open the first terminal and type in:

```
cd /opt/ros/foxy
```

Step 2: Open second terminal and type in:

```
source /opt/ros/foxy/setup.bash
```

Step 3: In the second terminal after step 2 type in:

```
ros2 launch rplidar_ros view_rplidar.launch.py
```

ROS1 Noetic with HectorSlam

Watch this video and follow instructions step by step:

https://www.youtube.com/watch?v=Qrtz0a7HaQ4

There are two methods of running this program. First method is a bash script made so you can run in 1 line. If this method doesn't work, proceed to method 2.

Method 1: Bash Script

```
cd catkin_ws
sudo bash run_slam.bash
```

Method 2: Manual

This method requires multiple terminals open, just follow the instructions step by step.

Open terminal one:

```
cd catkin_ws
source devel/setup.bash
sudo chmod 666 /dev/ttyUSB0
roslaunch rplidar_run rplidar.launch
```

Open terminal two:

```
cd catkin_ws/src
```

```
source devel/setup.bash
sudo chmod 666 /dev/ttyUSB0
roslaunch hector_slam_launch tutorial.launch
```

Install Intel RealSense SLAM

- Install Realsense SDK: https://nu-msr.github.io/me495_site/realsense.html
- Install DDynamic Reconfigure: https://github.com/pal-robotics/ddynamic_reconfigure
    - Clone to catkin-ws/src
- Download realsense ROS package
    - Clone from the legacy branch: https://github.com/IntelRealSense/realsense-ros/tree/ros1-legacy
    - Clone to catkin-ws/src
- Download Octomap: https://github.com/OctoMap/octomap_rviz_plugins.git
    - Clone to catkin-ws/src

```
sudo apt-get install ros-noetic-imu-filter-madgwick
sudo apt-get install ros-noetic-rtabmap-ros
sudo apt-get install ros-noetic-robot-localization
catkin_make
source devel/setup.bash
```

Run Intel RealSense SLAM:

```
roslaunch realsense2_camera opensource_tracking.launch
enable_accel:=true enable_gyro:=true
```

Production Cycle

At this point, we have installed and configured the raw files from the github. In order to edit, compile, and deploy any changes to the system, one would need to edit those files

directly. For example, in order to modify the existing robot and its size, one would need to modify the Nav2 C++ file establishing this robot. Upon exiting and saving this file, the changes will be live into the system and can be tested with the simulation files provided. As this is an incomplete system that is being finished by another capstone team, there are still some necessary changes left. That team will make these changes and produce another final report instructing how it shall be used.