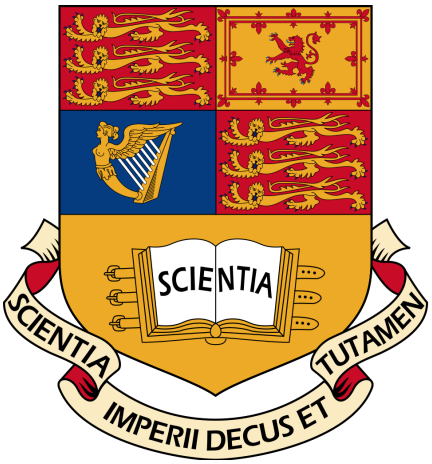
Imperial College London

Department of Electrical and Electronic Engineering

Final Year Project Report 2021



Project Title:	**Hey Drone! Map This Space**
Student:	**Daniel Gallego Rubio**
CID:	**01340223**
Course:	**MEng Electrical & Electronic Engineering**
Project Supervisor:	**Dr Christos-Savva Bouganis**
Project Repo:	**https://github.com/danigr99727/monocular_exploration**

Final Report Plagiarism Statement

I affirm that I have submitted, or will submit, electronic copies of my final year project report to both Blackboard and the EEE coursework submission system.

I affirm that I have provided explicit references for all material in my Final Report which is not authored by me and represented as my own work.

Acknowledgements

I'd like to thank my Dr Christos Bougannis and Mr Alexandros Kouris, for their continuous guidance during the course of this project.

I would also like to acknowledge the organisations responsible for developing and maintaining the open source projects that form the backbone of this project: The Computing Department at the Universidad de Zaragoza (ORB-SLAM2), the Aerial Robotics Group at the Hong Kong University of Science and Technology (FUEL), The department of Aerunautics and Astronautics at MIT (FLaME), Boston University and Adobe Research (TDNet) and Microsoft (AirSim). Source code availability was in fact the most important factor when choosing state of the art components for the system proposed in this project.

Finally, I'd like to acknowledge my friends for making these four years of university such an enjoyable time, and my family for their continuous love, encouragement, and of course, financial support.

Abstract

This project deals with the problem of enabling autonomous exploration of 3D environments using a small drone. More specifically, a system is proposed to enable autonomous exploration when a monocular camera is the only sensor available to form a map of the environment. This project also looks into the possibilities that state of the art semantic segmentation systems offer in order to allow UAVs to make better sense of the environment surrounding them, thus improving performance in the exploration task.

Contents

Chapter 1

Introduction

1.1 What this project is about

During the decade that just ended, significant advancements were made in the fields of computer vision and autonomous robotics. More specifically, small commercial drones have become popular platforms in academia; they are cheap, small and easy to maneuver, while also posing a challenge in terms of their sensor, power and computing capabilities. This project aims to design and (partially) implement a prototype where a low-cost drone is made capable of autonomously generating maps of its surrounding environment.

As the reader can probably imagine, there are many possible applications for small autonomous UAVs; Helping during search and rescue operations, mapping the interior of caves, volcanos, nuclear reactors and other inaccessible locations, etc. However, this project will treat autonomous mapping and exploration as an end on itself, rather than focusing on possible applications.

1.2 Requirements Capture

The system designed as part of this project has as its final aim to enable a small UAV to explore an arbitrary 3D environment of a predefined size. The system will be given the size of the environment it should explore, and will then autonomously navigate through the environment aiming to produce a 3D voxel map with a predefined resolution, where the resolution indicates the size of each voxel in meters.

The proposed system will not just form a geometric map of the environment, but will also aim to give it meaning, by annotating each voxel with a semantic label. For example, if a person is found in the environment, each voxel corresponding to that person should be given a “person” numeric label. This project will also look into a case study in which the use of semantically labelled voxels can enhance the navigation of the drone by selectively keeping longer distances when navigating close to certain kinds of obstacles like people.

The initial design aims to enable exploration while a map of the environment is formed from monocular images. Unfortunately, this final goal has not yet been implemented, and instead two

separate deliverables will be produced:

- Deliverable 1, the mapper: Will make use of state of the art computer vision techniques to form a 3D map of the environment from a stream of 640x480 monocular images. This task should be completed in real time, with a throughput of 30fps and a latency in the order of hundreds of milliseconds.

- Deliverable 2, the explorer: Will enable exploration of a simulation environment, while forming a 3D map of the simulation environment using ground-truth depth and odometry instead of monocular images.

1.3 High-level design: Hardware

As explained above, this project does not implement a full closed control loop on a real drone. However, design has been made with the Parrot Mambo FPV (see figure 1.1) in mind, with the hopes that with a bit of future work, the final design could be implemented on this drone. The mambo's battery can keep the drone flying for approximately 10 minutes. It has two cameras: A FPV (First person view) camera capable of recording 720p (configured to 480p for this project) video at 30fps, and a downward-looking camera used to assist the IMU (Inertial Measurement Unit) in providing stability. The FPV camera module has built-in WiFi, which is used to stream video in real time via RTSP [1].

Figure 1.1: Parrot Mambo FPV

Rather than running the computationally heavy mapping algorithms in the drone, a base-station approach is taken (see figure 1.2) the computationally expensive tasks are offloaded to a laptop running Ubuntu 18.04. The laptop has a 4-core Intel i7-6700HQ CPU, 16GB of RAM and an Nvidia GTX 1070 graphics card with 8GB of dedicated VRAM.

Figure 1.2: Hardware high-level overview of the proposed system.

1.4 High-level design: Software

More detail will be given about how these technologies work and why they have been chosen to form each deliverable. For now, a high-level overview is given. The mapper deliverable will consist of the following high-level components (see figure 1.3):

- RTSP decoder and rectificator: Receives video from the drone, decodes it, and rectifies it. The rectification step removes distortion caused by the camera lens, and is a necessary step when feeding images to many computer vision algorithms.
- Sparse SLAM: SLAM stands for Simultaneous Localization and Mapping. This module receives a stream of rectified images and outputs odometry estimations.
- Depth estimator: Receives a stream of rectified images and of odometry estimations, and outputs a stream of depth estimations.
- Voxel Mapper: Uses the odometry and depth estimations to update a 3D occupancy map.

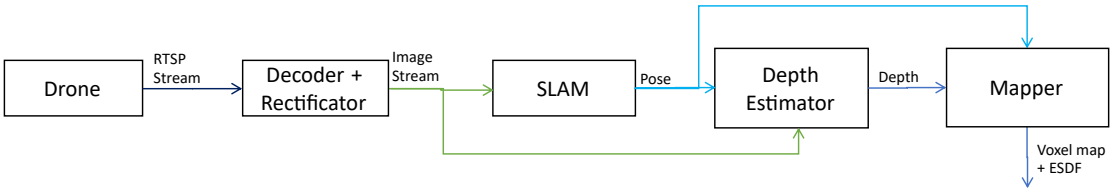


Figure 1.3: Deliverable 1: The mapper

The explorer deliverable will use a simulation known as AirSim to retrieve ground-truth depth and odometry measurements form a simulation environment. The following components are then used to form a closed loop (see figure 1.4):

- Explorer/Planner: Uses the occupancy map and ESDF and inputs to a planning algorithm that has as ultimate aim the exploration of an environment of a specified size. Its output is a stream of trajectories.
- Controller: Receives streams of odometry estimations and trajectory commands, and outputs control commands for the drone. The aim of the controller is for the actual trajectory of the drone to closely follow the desired trajectories given by the planner.
- Semantic segmentator: Recieves a stream of rectified images and outputs a stream of semantically segmentated images.

- Mapper's semantics extension: Will use the semantically labelled image stream from the segmentator to give a semantic label to every occupied voxel in the map. These semantic labels can then be used to change the drone's behaviour when navigating close to certain objects.

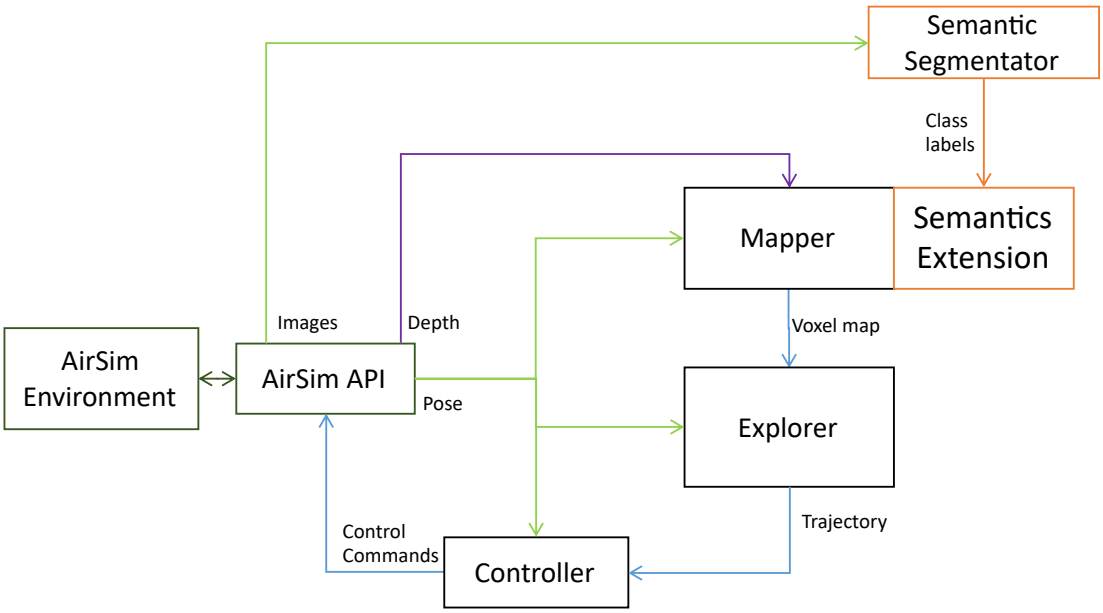


Figure 1.4: Deliverable 2: The explorer

1.5 This report's structure

- Background: Introduces previous works on UAV monocular exploration, as well as on semantically-aided navigation. It also gives an overview of the state of the art of the technologies and algorithms behind the components used for this project.
- Analysis and Design: Gives a detailed description of the design process followed to come up with the designs described above. It also explains why different third-party algorithms and libraries were chosen for each component.
- Implementation: Discusses the most important and interesting aspects on how the deliverables were put together, as well as lower-level design choices that were made.
- Testing: Explains the steps taken to verify functional correctness of the two deliverables.
- Results: Aims to show how well the deliverables work, as well as how each component's parameters affects the overall system performance.
- Evaluation and Conclusions: This final chapter critically evaluates the whole project and compares it to previous works.

Chapter 2

Background

Across the literature, the autonomous mapping problem is usually approached by researchers in the following way:

- Firstly, the position and orientation of the agent within an environment must be found. This is known as **odometry**.

- Secondly, a representation of the environment surrounding the agent must be formed. This is a particularly hard problem to solve when using monocular cameras, given that they don't provide the depth information that laser sensors, RGBD cameras or stereo cameras provide.

- Thirdly, a planning algorithm is used to make decisions on how to manoeuvre the agent at each point in time. Planning algorithms are often split into a hierarchical structure: local planning algorithms are tasked with maneuvering the agent within the vicinity of a starting point, while global planning algorithms are tasked tasked with higher level tasks such as "what part of the environment should be explored next?".

One alternative to this approach that has been explored by several researchers [2] is what is known as "Behaviour Reflex". This approach aims to train neural networks that directly map the input coming from sensors to the outputs given to actuators, without having to create an accurate representation of the environment as a middle step. This area of research is particularly interesting given its similarities with the way in which humans and animals autonomously navigate environments, but the Behaviour Reflex approach also seems to not be as popular, and no works have been found that achieve full UAV autonomy on the task of exploring arbitrary 3D environments using Behaviour Reflex.

2.1 Monocular autonomous navigation and mapping

In this section, we explore three recent papers that work on different approaches to the tasks of monocular camera-based navigation, exploration and mapping. These papers were selected because they are the newest works found on three different approaches (reinforcement learning based, sparse-SLAM based and depth-estimation + voxel map based) to monocular navigation, and therefore are considered to be a good representation of the state of the art.

2.1.1 Neural SLAM (2020)

Reinforcement learning has been attracting a lot of attention in the field of robotics in the last few years. Most papers found consider ideal scenarios where the environment is already known, or the robots can easily form a representation of the environment using expensive laser sensors. However, in Neural SLAM [3], researchers from Carnegie Mellon University propose an architecture that uses reinforcement learning methods to explore an environment using uniquely a monocular camera. Their pipeline consists of:

- A SLAM module which gives a pose estimation for the agent and builds a dense 2D map of the environment.
- A global policy, in charge of exploiting the regularities of real-world environments in order to select a point in space where the agent should try to navigate in the long term.
- A planner in charge of using the long-term goal to produce short-term goals.
- A local policy, which takes the short-term goals and the current frame and aims to maneuver the agent towards the short-term goal

The authors of Neural SLAM use the Habitat [4] simulation to train and test their algorithms. The authors explain how this modular approach to an exploration pipeline reduces the size of the search space during training, giving Neural SLAM an edge with respect to papers such as Chen et al. [5], that propose end-to-end learning-based exploration strategies. The original Neural SLAM described on this paper works exclusively in 2 dimensions, and therefore it cannot leverage the capabilities of quadcoptors to move in all three dimensions, and would not be suitable to explore environments where height is a relevant direction (e.g, a cave, a multi-storey building, etc) unless modifications are made to the original work.

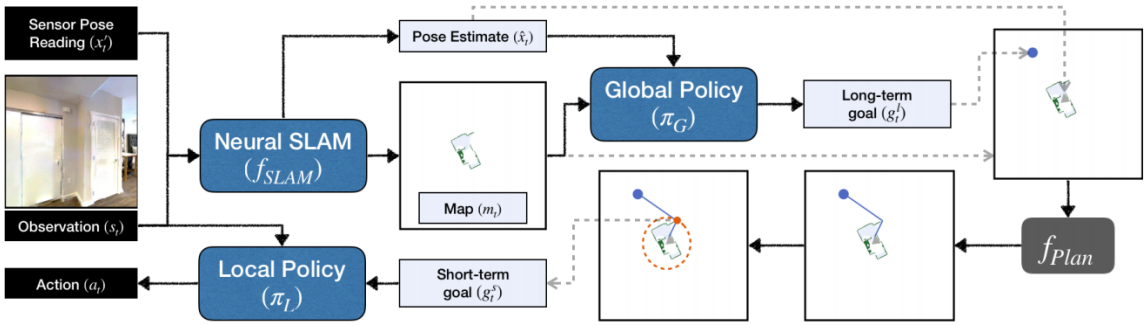


Figure 2.1: Neural SLAM pipeline.

2.1.2 LoS-Exploration (2020)

In LoS-Exploration [6], researchers propose a system that carries out exploration using ORB-SLAM [7] features as "hints" to create local occupancy maps that it can use to safely navigate the environment; A volume in space containing lots of ORB-SLAM features is considered to have a larger probability of being occupied, while volumes present in a line of sight between these features and the current position of the agent are considered to have lower probabilities of being occupied. This system is proven to be able to fully explore a simple apartment environment running on the AirSim

simulation. It avoids any collisions, and is shown to complete the exploration task independently of what the starting point is. The system is also particularly lightweight, given that ORB-SLAM2, it's most computationally expensive component, can run on real-time using a single CPU core. However, the whole exploration process is slow for a drone like the parrot mambo which can only fly for 10 minutes at a time. Furthermore, the environment where the authors carry out their experiments in seems to be somewhat tailored to the algorithm the authors propose; most of the obstacles are large walls with many paintings on them, helping to increase the amount of features detected by ORB-SLAM2.

2.1.3 Depth estimation + 3D Occupancy grid for navigation (2018)

In [8], authors at the HKUST (Hong Kong University of Science and Technology) Aerial Robotics Group use a monocular fish-eye SLAM pipeline (more specifically, it uses VINS [9]), followed by a depth estimator. The depth images are then transformed into a grid map which is used by a trajectory planner to allow for autonomous navigation.

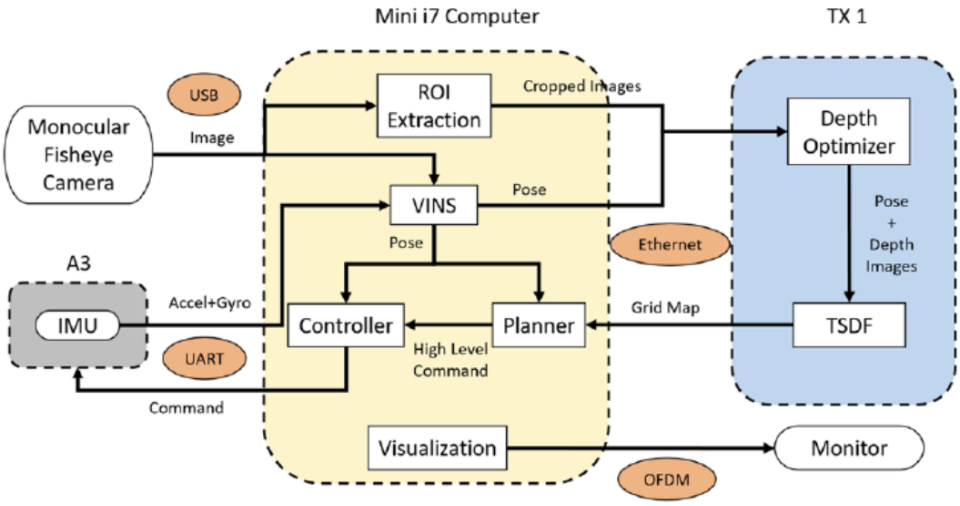


Figure 2.2: Software pipeline proposed by the HKUST researchers.

The system proposed in this paper offers a good performance-speed balance: It can run at 10Hz, with an average latency of around 230ms (slightly better than the human reaction time) on a computationally constrained platform consisting of an Intel i7 CPU and an Nvidia TX1, which only has 206 CUDA Cores, compared to the 1,920 CUDA cores of the GTX 1070 used in this project. The system offers more robust maneuverability than LoS-Exploration, probably thanks to the denser mapping of the environment is produces. It is also able to create 3D maps, as opposed to the 2D maps that Neural SLAM works with.

2.2 Semantically-aided monocular navigation

Semantic segmentation, or the ability to semantically label regions in an image is one of the many tasks that deep learning has been facilitating during the rise of deep neural networks that took place in the last few years. Many academics in the field of robotics and autonomous navigation are now

interested in finding new applications for these systems; The main idea is that robots should be able to exploit semantic information about the environment surrounding them to improve their decision making capabilities. It was found that most literature in this area focuses in the specific problem of navigating in road and urban environments, however, some recent works [10] [11] have been found that are more closely related to this project.

One particularly interesting paper, published in early 2021 by the same authors responsible for Neural SLAM, is "Object Goal Navigation Using Goal-Oriented Semantic Exploration". The system proposed in this work (see fig. 2.3) consists of a mapping module, which builds a semantic map of the environment from RGBD images and odometry readings, and a Goal-Oriented semantic policy module, which is given the task of exploring the environment until a specified object is found. The authors show how a policy trained when using semantic information greatly over-performs a policy trained on a purely geometrical map when it comes to the speed in which the agent finds the specified object. This is, the authors argue, because when the semantically trained policy is asked, for example, to find a dining table, it will have an understanding of where dining tables are more likely to be located, and will take the objects already found in the environment into account to take new decisions on where to navigate.

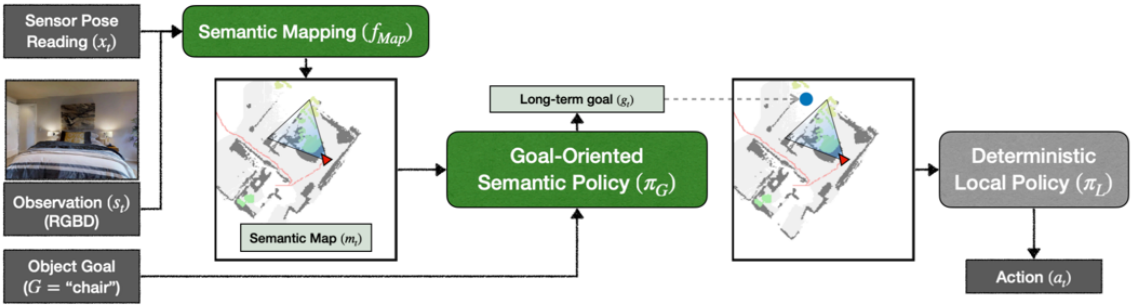


Figure 2.3: Source: [11]

2.3 Camera calibration and the pinhole camera model

A point in a 2D image will correspond to a specific set of points in the 3D world. Understanding the mathematics behind what is known as the "Pinhole Camera Model" is essential to understanding the functioning of some of the computer vision tools mentioned in this chapter, as well as to learn how to calibrate them to work with specific cameras. For a pinhole camera with no distortion, the following relation holds between the intrinsic parameters (first matrix), the extrinsic parameters (second matrix, indicates the rotation and translation of the camera), 3D world coordinates and 2D image coordinates:

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{2.1}$$

2.4 Odometry: Localizing an agent within an environment

This is a fundamental problem to consider when programming robots capable of autonomous navigation and exploration: We require the robot to be aware of its orientation and position within the environment. The problem can be solved with a variety of sensors (GPS, sonar, laser, IMU, wheel speed sensor, etc), but in this project we focus on what is perhaps the most popular odometry technique when GPS is not available or not accurate enough: Visual Odometry (VO). More specifically, we will focus in Monocular VO, which uses monocular cameras as its main sensor. As explained in [12], a basic VO pipeline has the following stages:

- Feature detection: A set of distinct points in the image will be selected.
- Feature matching: Similarity between the features detected in two frames is calculated in order to determine correspondence between features.
- Motion estimation: Matching features are used to compute the transform between the two frames.

Figure 2.4: Feature matching between two images. Source: OpenCV [13]

Visual odometry makes several assumptions about the environment:

- The environment must be sufficiently illuminated.
- The environment must have enough texture. As a thought experiment: If a human is put in the middle of a large white room, he won't be able to tell easily how far he is from the walls if he only uses his eyes to asses his position.
- Most objects in the camera's field of view must be static. The presence of a few dynamic objects in the environment is not an issue for state-of-the-art VO solutions, as they use statistical techniques to discard outlier features that do not move in the same direction as the rest of features in the field of view.
- Consecutive frames must be similar enough to contain matching features: If the robot happens to move or rotate too fast, then two consecutive frames will point to different regions of the environment, and it wont be possible to find the transform between the two frames. This issue

can be mitigated using systems capable of running VO algorithms at high frame rates, and is one of the reasons why computational load when forming maps is given such importance in this project.

2.5 SLAM: Simultaneous Localization and Mapping

As explained in [14], because VO only aims at optimizing the local consistency of a trajectory, it often suffers from a drift problem; error introduced with each new frame accumulates over time. SLAM, on the other hand, expands VO by adding pipeline stages that aim at optimizing global consistency by remembering the features detected and position estimated in previous frames. Modern SLAM pipelines look for loops in the trajectory of the agent to help to correct drift. Figures 2.5 and 2.6 illustrate the dramatic effect loop closure algorithms can have in correcting the shape of a trajectory.

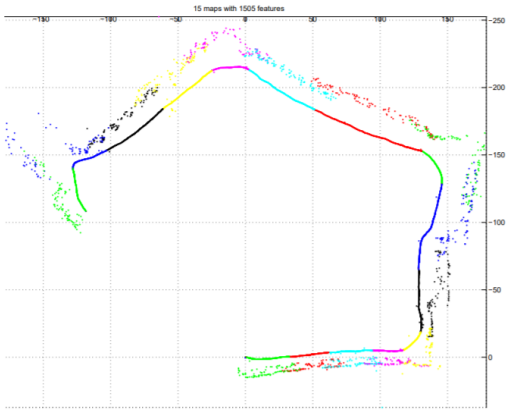


Figure 2.5: SLAM before loop closure. Source: [15]

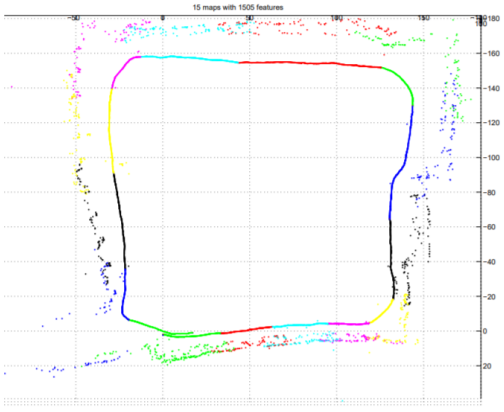


Figure 2.6: SLAM after loop closure. Source: [15]

One popular Visual SLAM C++ library, which is well documented and has a large community behind is ORB-SLAM2 [7]. ORB-SLAM2 can easily be adapted to many different camera types, and comes with several interesting features such as storing and loading of maps.

2.5.1 Learning-based visual SLAM methods

Like many other tasks in the field of computer vision, numerous attempts have been made in the last few years to outperform "classical" SLAM algorithms using learning-based techniques. One state-of-the-art learning-based visual SLAM work is Deep-VO. In Deep-VO [16], researches propose a method where a live video is fed first into a series of convolutional layers, followed by a series of recursive layers (see figure 2.7). According to the authors, Deep-VO gives results comparable to state of the art classical VO methods. However, they don't make any mentions on whether their model can compete with geometry-based methods such as ORB-SLAM2 when it comes to computational requirements. ORB-SLAM2 can easily run on a single CPU core, while machine learning models models relying in convolutional neural networks (as it is the case with Deep-VO) often require powerful GPUs to run in real-time. Comparing performance of these two VO approaches could be an interesting future work task as an extension for this project.

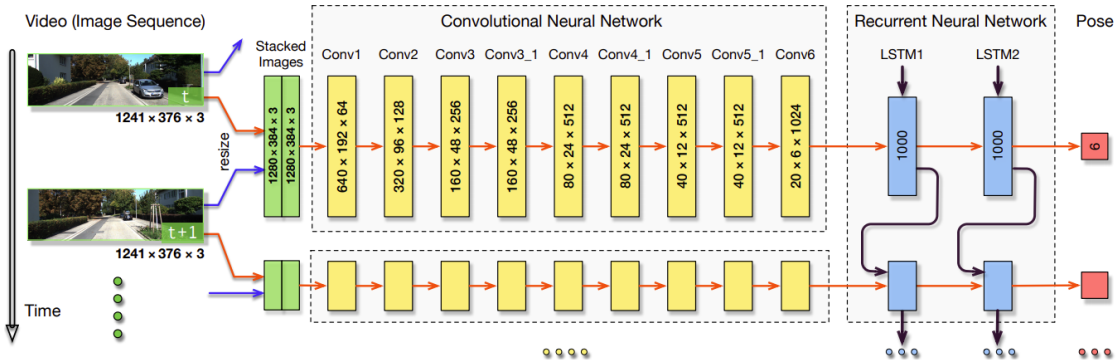


Figure 2.7: DeepVO's neural network architecture overview.

As explained in [3] the main advantages of learning-based SLAM/VO approaches are:

- Learning provides flexibility of choice of input modalities: classical systems rely on observing geometry through the use of carefully calibrated sensors, while learning systems can infer geometry directly from RGB images.
- The use of learning can improve robustness to non-ideal circumstances such as poor illumination or difficult weather conditions.
- Learning can effectively leverage structural regularities of the real world, leading to more efficient behavior in previously unseen environments

2.5.2 Boosting Visual SLAM with object detection

Object-SLAM [17] [18] [19] [20] [21] [22] aims to combine traditional SLAM methods with object detection in order to a) produce more meaningful maps by adding semantic information, and b) leverage object data to improve accuracy and robustness of SLAM systems. In CubeSLAM [20], researchers explain how objects can provide geometric and scale constraints, reducing drift in monocular SLAM. In DS-SLAM [17], semantic segmentation is incorporated into a SLAM pipeline, which is shown to reduce the impact of dynamic objects in the absolute trajectory accuracy with respect to ORB-SLAM [7].

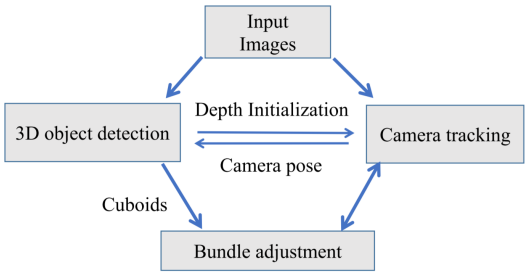


Figure 2.8: CubeSLAM pipeline.

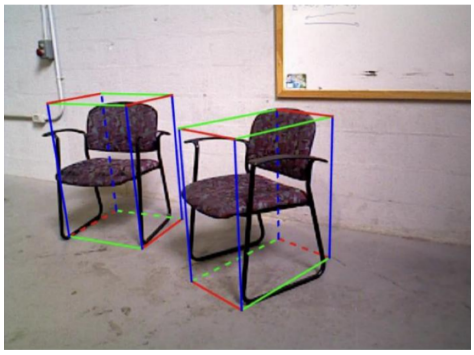


Figure 2.9: CubeSLAM's cuboid proposals.

Another interesting Object-SLAM proposal is SLAM++ [23], by researchers at Imperial College. This paper shows how object detection can accelerate dense reconstruction of 3D environments by

leveraging repetitive, domain-specific objects and structures, thus avoiding unnecessary computational effort.

2.5.3 Boosting Visual SLAM with inertial measurements

Accross the literature, one common way of improving the performance of visual SLAM algorithms, is to combine visual data with that of an IMU (Inertial Measurement Unit). These are cheap sensors which always come embedded within UAVs, as they are essential to maintain stability. One examples of Visual + IMU SLAM is VINS-Mono [9]. The authors of this paper explain that an issue with traditional monocular SLAM is how these systems are incapable of recovering scale; they may be able to tell that "the distance from A to B is twice the distance from B to C", but won't be able to tell whether the distance form A to B is 1m or 10cm. In addition, an IMU has the potential to bridge any gaps if the visual features monocular SLAM relies on are lost for a few frames, thus increasing robustness of the SLAM system. The authors also mention some issues and challenges that arise when trying to fuse information from the two sensors, but ultimately, they claim to obtain better quantitative results when evaluating VINS-Mono against previous visual-inertial SLAM solutions.

2.6 Depth estimation from cameras

Two main approaches for depth estimation have been found in the recent literature; geometry-based and learning-based. Geometry-based approaches aim to build a 3D representation of the environment using epipolar geometry, a set of mathematical techniques used to relate the position of a point in two images to the position of that point in the real world when the intrinsic and extrinsic properties of each frame are known. These approaches can be computationally demanding, but lately a few works like [24] and [25] have been shown to make depth estimations on CPUs at very high framerates.

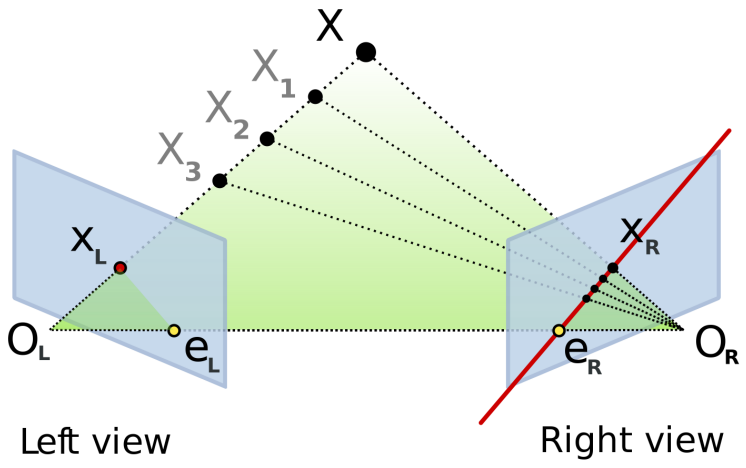


Figure 2.10: Epipolar geometry. Source: Wikipedia

On the other hand, learning-based approaches using Convolutional Neural Networks [26], Recurrent Neural Networks [27], and Generative Adversarial Networks [28] have all been successfully used for depth estimation by numerous researchers. However, a lot of these architecture solely consider

depth estimation from a single image, which fails to leverage the valuable geometric information that can be obtained when using consecutive frames in a video stream. In MVDepthNet [29], researchers propose a neural network architecture that does leverage video streams of related frames, while being able to run real-time in an Nvidia Titan-Xp GPU.
Another area of research being explored recently are neural networks combining two or more tasks into a single architecture. For example, in SimVODIS [30], the authors show an architecture that takes a sequence of images as an input, and performs instance segmentation, visual odometry and depth estimation at the same time. One benefit of systems like SimVODIS is the computational power they save when compared with a system that runs three separate networks. The other benefit is that these tasks can be helpful to each other, in a similar manner to how Visual Odometry and Object detection were found to be mutually beneficial in Cube-SLAM.

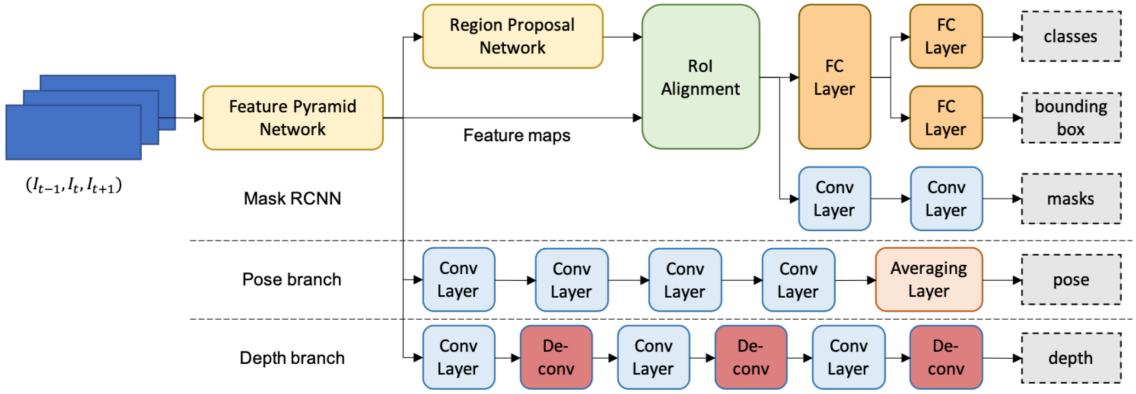


Figure 2.11: Overview of the SimVODIS neural network architecture.

2.7 Creating occupancy grids from depth images

In this section we explore systems capable of fusing sequences of depth images and odometry data to build a 3D occupancy map of the environment. Some examples are FIESTA [31] and VoxBlox [32]. The way these systems work is often as follows:

1. The depth image is transformed into a depth cloud by projecting every pixel into the real world using the depth camera's intrinsics (see equation 2.2). The camera's extrinsics provided from the odometry measurements are also taken into account to project the point cloud into the world coordinates.

2. A data structure is used to represent a 3D box made up of voxels where every voxel is labelled as "occupied" or "unocupied". Every time a new point cloud is received, calculations are performed on what the new occupancy probability is for the relevant voxels, and the data structure is updated accordingly.

$$\begin{bmatrix} p_x(i,j) \\ p_y(i,j) \\ p_z(i,j) \end{bmatrix} = \begin{bmatrix} (i - c_x) * \texttt{depth}(i,j)/f_x \\ (j - c_y) * \texttt{depth}(i,j)/f_y \\ \texttt{depth}(i,j) \end{bmatrix} \tag{2.2}$$

2.8 Exploring and Path Planning

Once the system has reconstructed a good representation of the environment, the next step is to make decisions on what to tell the agent to do. In systems that aim to autonomously explore the environment a distinction is often made between an algorithm in charge of selecting where the agent should explore next, and an algorithm in charge of generating trajectories. In order words, a hierarchical approach is followed where separate algorithms are responsible for making local and global plans.

In the global side, one popular family of exploring algorithms are frontier-based approaches, first introduced by Yamauchi [33] in 1997. Yamauchi's algorithm works by selecting the closest frontier in space as the next target, and since then several researchers have proposed improvements to his work. Another family of exploration algorithms are sampling-based approaches, which aim to sample the environment in search for the best possible next view. For example [34] uses a data structure that facilitates finding the region of the environment with the highest estimated information gain.

In the local side, algorithms in charge of generating trajectories that facilitate navigation to local goals often have to be mindful of the dynamic properties of the agent being controlled. Several state of the art works [35] [36] [37] were found that combine the exploration techniques mentioned above with quadcoptor-specific trajectory planning algorithms.

2.9 Semantic segmentation

Semantic segmentation is the task of giving every pixel in an input image a semantic label corresponding to the object that pixel is a part of. As already mentioned, state of the art performance in semantic segmentation has dramatically surged in the last few years thanks to new deep learning techniques. When it comes to robotics-related applications, the main direction of recent research has been in what is known as "video semantic segmentation". The conventional approach when using semantic segmentation in a video feed consists of processing frames one by one as if no correlation existed between them. This can be considered sub-optimal, and is a problem that video semantic segmentators aim to solve.

One family of video semantic segmentation models work by running a conventional segmentation model for all frames, and then adding additional layers that take advantage of the temporal dimension to extract better features [38]. Other methods aim to tackle the problem of redundant computation by, for example, using LSTMs [39] or selecting keyframes to fully segment, while only updating certain layers of their neural network on non-keyframes [40].

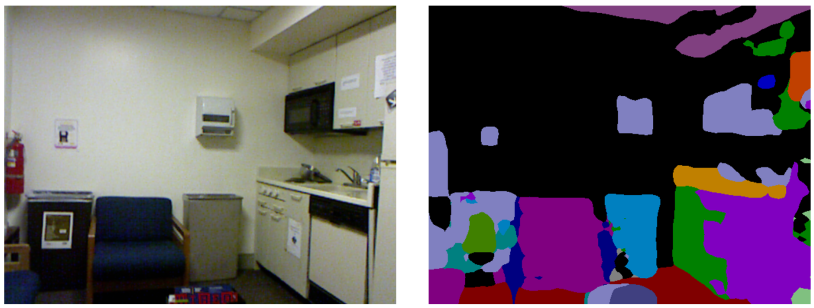


Figure 2.12: Running a semantic segmentation model (TDNet) on an image from the NYUD v2 dataset.

Chapter 3

Analysis and Design

3.1 Defining the system's architecture

During the earlier stages of this project, attempts were made at designing a system capable of navigating an environment while building a representation of the environment that is as sparse as possible. A sparse representation should require fewer computational resources to construct and to take decisions with, and is therefore more desirable than a dense one when the objective is just to explore the environment, rather than to explore **and** build a detailed model of the environment. Some recent works that use sparse representations to allow for autonomous navigation like the one by Teso et al. [41] were immediately discarded, as they only allow for autonomous navigation in simple, specific environments (a corridor with a few obstacles in the case of Teso's work), and place no emphasis on generalizing to a more diverse set of environments. One work that was considered is LoS-Exploration [6] (see background chapter for a more detailed explanation on LoS-Exploration). Attempts were made to test the code released by the authors in GitHub. LoS exploration uses AirSim as its demo simulation, but the authors did not provide the environment they use to show their paper's results. Instead, the indoor environments provided by PEDRA [42] were used. The results obtained were poor; exploration often stopped after only a small portion of the environment had been explored, and as hypothesized in the background section, LoS-Exploration wasn't found to be able to generalize well in environments different to the one used in the paper.

The next step was to consider a denser environment representation, and a system similar to that proposed by HKUST researchers in [8] was considered. This system uses depth and odometry estimations to construct a voxel occupancy map. 3D space is divided into cubes of a specified size, and then each cube is labelled as occupied or unoccupied. Depending on the application, larger or smaller voxels can be chosen, in a trade-off between computational load and level of realism of the produced map. The final system designed for this project is partially based in the HKUST design (see figure 3.1), but some improvements have been made:

- According to the authors of the depth estimator used in this project (FLaME [24]), their work is both less computationally demanding than previous works (FLaME doesn't need a GPU as the solution used by HKUST does), and achieves better accuracy than previous works.
- FIESTA [31] (2019), the mapper module chosen for this project, postdates the work by HKUST. FIESTA claims to outperform all previous state of the art and unlike the mapper used in HKUST's

work, it can run exclusively on a single CPU core. When using a CPU+GPU laptop as a base station, as it is the case in the project, having tasks that can efficiently run in the CPU can be beneficial, given that other tasks such as image segmentation are likely to heavily load the GPU.

- The HKUST work focuses exclusively on navigating from point to point, while this project focuses on the whole exploration task.

- This project also adds a "semantic extension" which aims to use semantically segmentated image to enrich the produced map with semantic data, as well as improving the way the navigation algorithm works.

As already explained in the introduction chapter, the final design of this project shown in figure 3.1 has not been implemented, and instead, the system has been split into two deliverables (look back at figures 1.3 and 1.4).

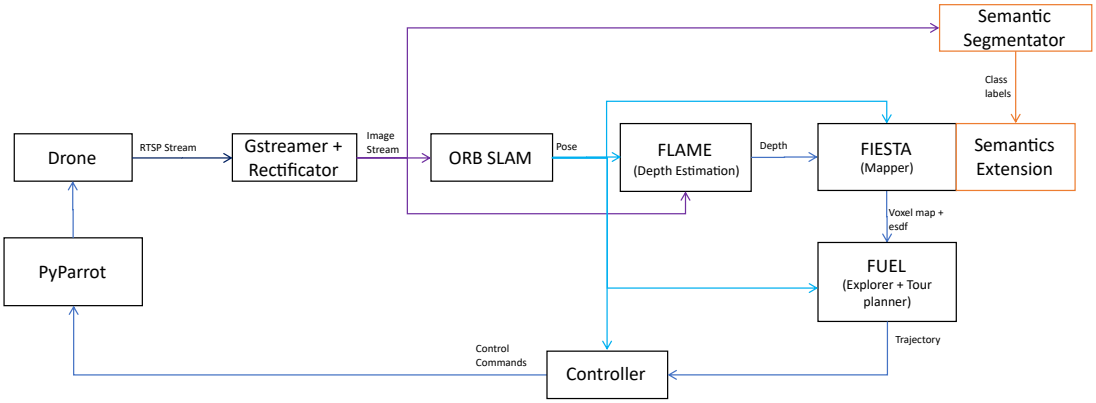


Figure 3.1: Overview of the final proposed system (not implemented as of the time of writing this report)

3.2 The RTSP decoder module

Several software tools are available to recieve real-time video streams via RTSP. FFMPEG, VLC and Gstreamer were all tested with the Parrot mambo. Out of them, Gstreamer was selected for this project as it was seen to provide more robust video outputs and also because of the powerful command-line tool it offers to quickly try different video decoding pipelines. Another advantage of Gstreamer is are the abundance of plugins produced by third-parties like Nvidia. According to Nvidia [43] many of their GPUs, including the GTX 1070 used in this project, contain one or more dedicated hardware-based decoders and encoders which provide video decoding and encoding for several popular codecs, among them H264, the one used by the Parrot Mambo. If Nvidia's decoding plugging for Gstreamer is used, then the decoding task will be completely offloaded to the NVDEC hardware component, leaving both the CPU and the GPU's main cores (or CUDA cores) available for other tasks.

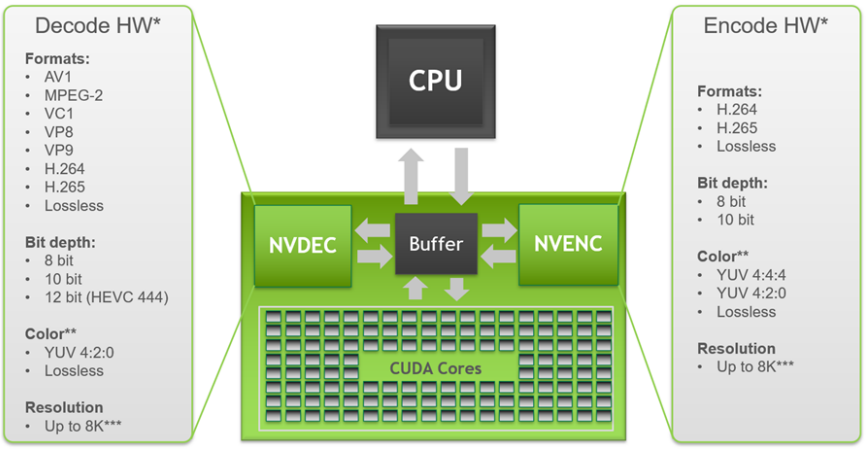


Figure 3.2: Nvdec and Nvenc within an Nvidia GPU. Image courtesy of Nvidia

3.3 Choosing a SLAM module

A recent review from February of 2021 [44] bench-marked several state of the art visual and visual-inertial SLAM algorithms. Despite of being a few years old (2017), ORB-SLAM2 was concluded to be the best performer overall. It was able to even slightly over-perform visual-inertial frameworks such as VINS-Mono [9]. The area where ORB-SLAM2 was weakest, however, was scale estimation. VINS-Mono was briefly tried and tested together with AirSim, but it was soon realized that integrating AirSim's IMU data with VINS-Mono was a challenge there was no time for during this project, and therefore ORB-SLAM2 was the final choice for a SLAM module.

3.4 Choosing a depth prediction module

The authors of FLaME show how their algorithm is capable of running on a single Intel Core i7 3.4GHz CPU core while generating mesh reconstructions at up to 230 Hz. This frequency is dependent on several parameters, but in any case, flame is shown to beat all previous alternatives when it comes to speed and estimation accuracy. The authors also show how FLaME can be used for autonomous drone 2D navigation by creating a 2D occupancy grid updated using slices from the FLaME meshes.

The philosophy behind FLaME's groundbreaking speed, the authors argue, is that that many conventional depth estimation techniques **oversample** the images they receive; most real-world scenes have a limited geometric complexity, and for many real-world applications of depth prediction, a fine, pixel-by-pixel estimation of depth is not needed. As illustrated in figure 3.3, FLaME works by:

- Selecting a number of features.
- Matching these features to the previous frame and estimating the depth of each feature using epipolar geometry.
- Placing all features in 3D space, and connecting them to form a 3D mesh.
- Regularize the 3D mesh by smoothing noise away.

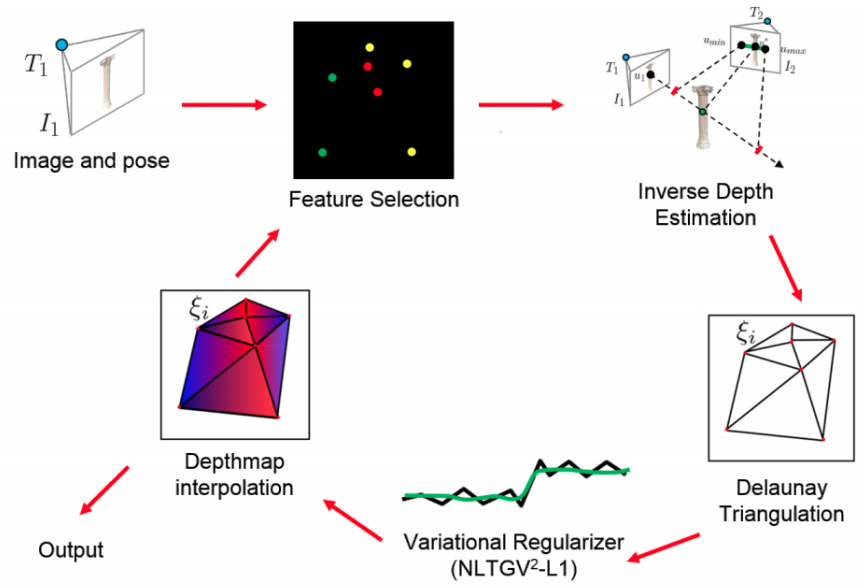


Figure 3.3: Overview of how FLaME works. Source: FLaME Paper [24].

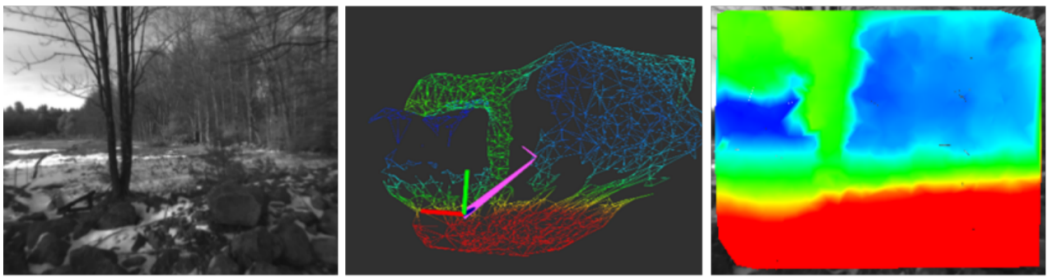


Figure 3.4: Mesh and Depth Estimation generated with FLaME [24]

3.5 Choosing the voxel mapper module

Several open-source solutions were found that perform depth-to-voxel mapping, including VoxBlox [32], RTAB-MAP [45] and FIESTA [31]. FIESTA, which is the most recent work (Jul 2019), claims to outperform VoxBlox in accuracy and performance by an order of magnitude. In addition, FIESTA also produces an ESDF (Eucledian Signed Distance Field) map, which some planners such as CHOMP [46] and FUEL [35] make use of. ESDF, which provides information on how far points in the free space are from obstacles in the environment, is particularly useful for safe quadrotor navigation. For these reasons, FIESTA has been chosen as the volumetric mapping module for this project.

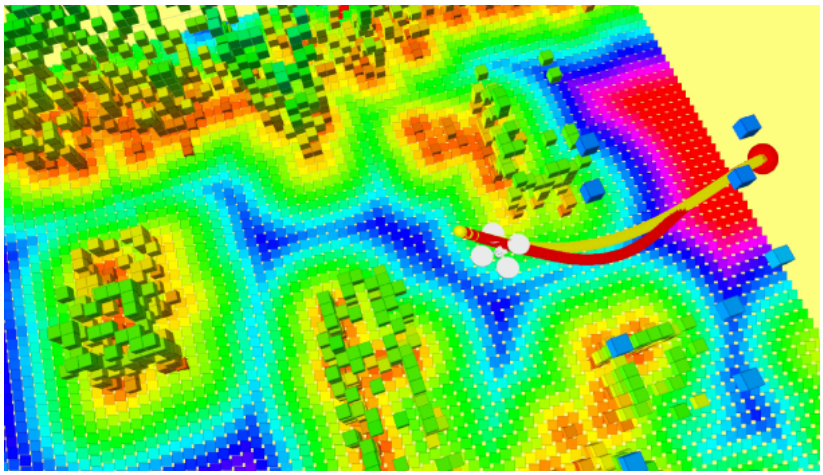


Figure 3.5: Map created with FIESTA

3.6 Choosing the explorer module

When creating environment exploration algorithms, trade-offs often have to be made between speed and mapping accuracy. FUEL (Fast UAV Exploration using Incremental Frontier Structure and Hierarchical Planning) [35] is a recent paper (December 2020) proposing a system that performs frontier-based exploration faster than previous approaches, while taking into account the dynamics of quadrotors. Some alternatives exist of similar works that have also published their source code online, for example, [36] and [37]. However, FUEL was finally chosen because its original codebase already includes FIESTA as a submodule, making the integration task between the two components easier.

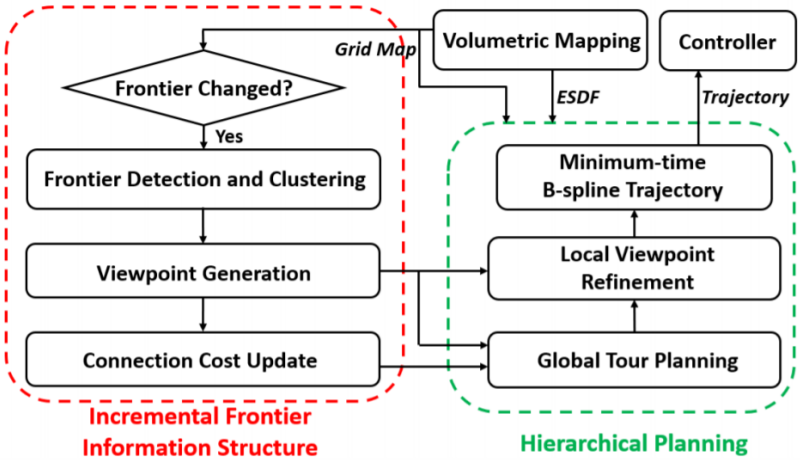


Figure 3.6: FUEL system overview [35].

3.7 Choosing a semantic segmentator

TDNet [47], a work from April 2020, was chosen because of its excellent speed and accuracy results and because of the availability of pre-trained models provided by the authors. TDNet, as

explained in its paper, was developed to address some of the challenges that come with keyframe-based video semantic segmentation systems. In a conventional image segmentation model, such as PSPNet, frames are first encoded to extract what in machine learning is usually known as "features" (not to be confused with the concept of feature when talking about feature matching in ORB-SLAM or FLaME). These features are then fed to consecutive layers that form the output image. In TDNet, however, consecutive frames are passed through separate "shallower" feature extractors that have fewer layers and are therefore faster to compute. The features extracted from each shallow network are then combined through what is known as an Attention Propagation Module before being passed through the rest of the network (see fig. 3.7).

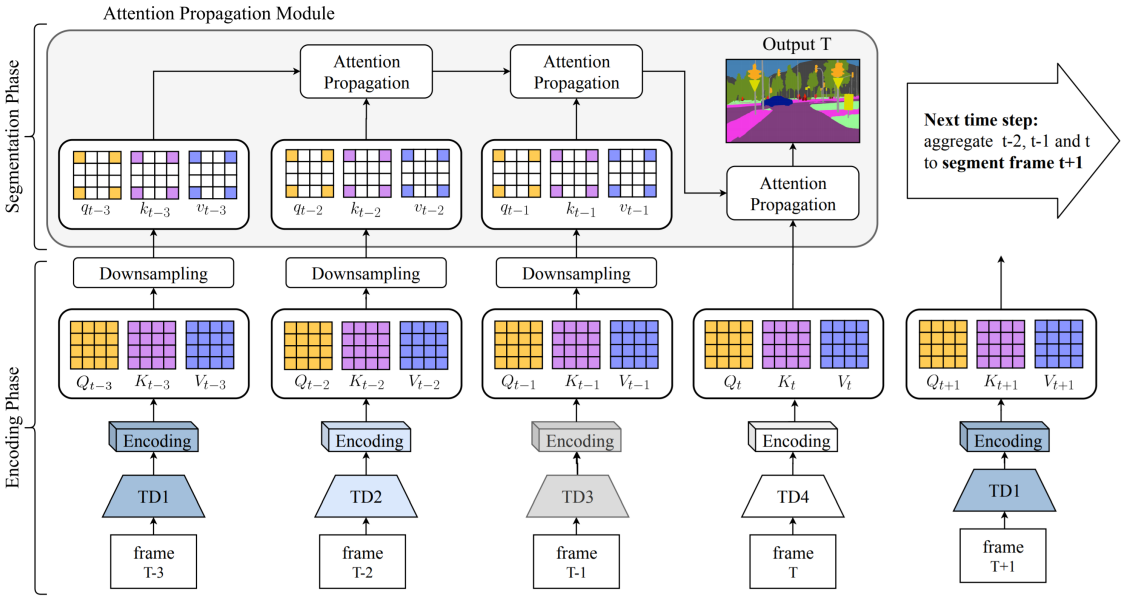


Figure 3.7: Overview of the TDNet architecture. Source: [47]

The authors of TDNet show performance results for the Cityscapes dataset (see fig. 3.8). Both their BiseNet and PSPNet based models show significant latency reductions, while maintaining similar mIoU (mean intersection over union) performance. For this project, the TD^2-PSP50 model was chosen. More specifically, a version of the model trained on a dataset featuring video sequences of indoor environments known as NYUD v2 [48] is used.

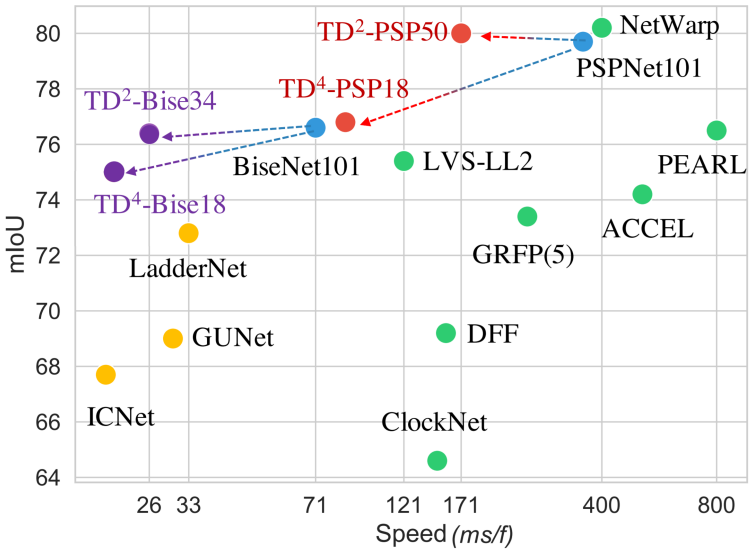


Figure 3.8: TDNet measured against the state of the art. Source: [47]

3.8 Defining the semantics extension.

As already discussed in the background chapter of this report, the addition of semantic information into a map can help a navigation/exploration algorithm achieve superior performance. This project aims to expand the existing codebase of FUEL to implement the case study mentioned in chapter 1.2 so that:

1. FUEL shall produce a semantically-labelled map, that is, every voxel within the map that FUEL already produces will be labelled with a semantic tag corresponding to the object that voxel is a part of.

2. It is likely that all of the components in FUEL's hierarchy could be benefited from the use of semantic labels to improve different aspects of FUEL's performance. For the case study in this project, it is proposed that FIESTA is modified in such a way that the ESDF can be modified around obstacles depending on their semantic label. This could serve as a starting point for the implementation of more sophisticated algorithms, and some ideas will be given in the final chapter of this report.

3.9 Introducing a Simulated Environment: AirSim

A simulation can be a great tool to facilitate development and testing in robotics. Developing a closed loop system like the one proposed for this project using directly a real world drone would not be efficient, and would likely involve many hours of trial and error, while having to pay extra care to ensure safety and maintain the integrity of both the drone and its surrounding environment. For these reasons, a simulation is introduced as a development tool that would allow the proposed system to be tested and evaluated even before having to function in the real world.

Airsim is an open-source simulation created by Microsoft for researchers working with autonomous vehicles. It provides support for autonomous cars and autonomous drones., and comes with powerful APIs for Python and C++ that allow the user to retrieve information from and send information to the simulated drone. AirSim is also shipped with a ROS wrapper around its C++ API. AirSim's simulation environments, which run in Unreal Engine, communicate with the AirSim API over TCP/IP. This means that the computationally heavy task of running the simulation environment can be offloaded to a different computer connected to the same network as the "base station" computer. The AirSim simulation was chosen because of the large size of its online community, and for the high level of realism that its 3D environments can offer, which is not the case for other options such as Gazebo. Figure 3.10 shows how a real-world drone can be replaced by a simulation within the system in figure 3.1.

Figure 3.9: The AirSim simulation.

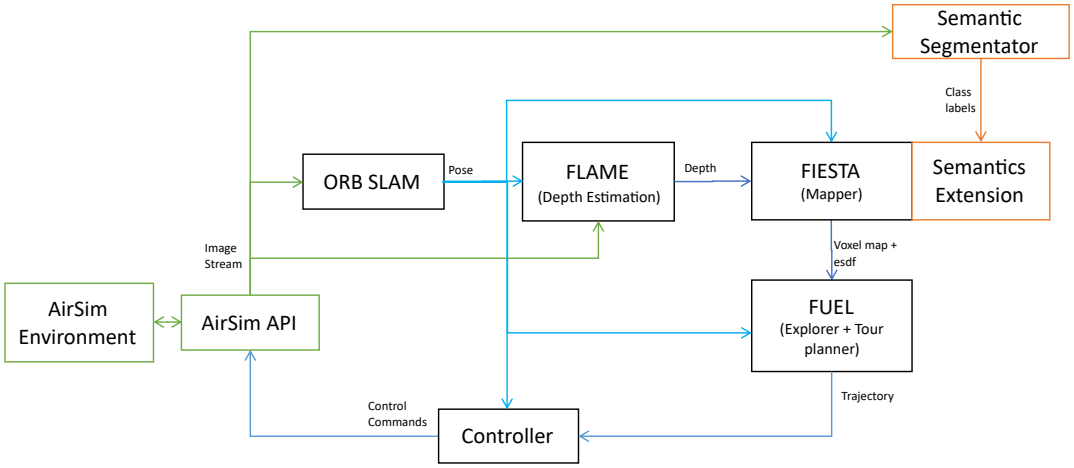


Figure 3.10: Overview of the proposed system (with simulation instead of physical drone).

3.10 The Controller Module

The controller's job is to use trajectory commands from the planner and odometry measurements to ensure that the drone follows the desired trajectory as closely as possible. In the case of the FUEL explorer, trajectories are published by an internal component known as `traj_server`. Each trajectory consists of three vectors representing the desired position (T_x), velocity (T_v) and acceleration (T_a) of the drone, as well as two scalars representing the desired yaw (T_ψ) and yaw rate ($T_{\dot{\psi}}$) of the drone.

Figure 3.11 shows the controller used by the demo that comes with FUEL. This controller, which is described in more detail on a separate paper [49], uses two control loops to control the position and the orientation of the drone, and outputs commands in the form of a force vector and an orientation quaternion.

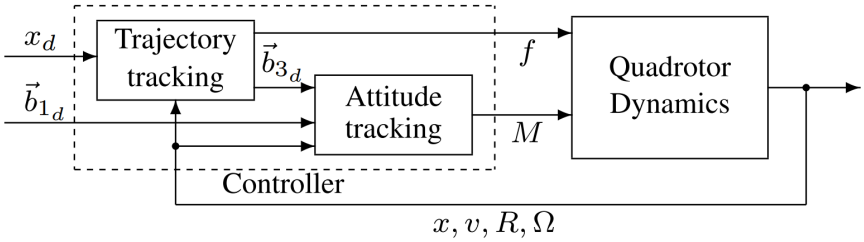


Figure 3.11: Controller used by FUEL's demo [49]

AirSim's ROS wrapper only takes commands in the form of a `geometry_msgs::Twist` message, which has two fields used to indicate the desired linear and angular velocity of the drone. If trajectory commands for the desired velocity are passed directly into the drone, small errors over time caused by simulated external disturbances will create a large drift between the actual and desired positions. Therefore, a controller had to be designed that can make corrections to the drone's trajectory while outputting commands in the `geometry_msgs::Twist` format. Designing control systems for quadrotors can be a big endeavour on its own, therefore, a simple linear controller with two separate control loops was designed to avoid spending too much time on this task.

The first control loop produces a linear velocity vector output "A_v". As seen in figure 3.12, the desired velocity and acceleration vectors, plus two terms whose size is proportional to the position and velocity errors are combined to give a final velocity command:

$$A_v = T_v + T_a + K_x(T_x - O_x) + K_v(T_v - O_v) \tag{3.1}$$

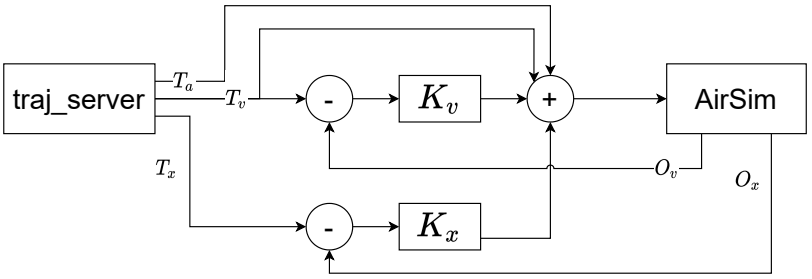


Figure 3.12: The trajectory to velocity command control loop.

The second control loop deals exclusively with the yaw rate scalar "$A_{\dot{\psi}}$" output. The desired yaw rate is added up with two terms whose size is proportional to the yaw and yaw rate errors. Calculating the yaw error is, however, not as trivial as simply subtracting two angles. Algorithm 1 shows how it was calculated instead.

$$A_{\dot{\psi}} = T_{\dot{\psi}} + K_{\dot{\psi}}(T_{\dot{\psi}} - O_{\dot{\psi}}) + K_{\psi} \times \texttt{AngleDiff}(T_{\psi}, O_{\psi}) \tag{3.2}$$

Algorithm 1 Calculating error between two angles

procedure AngleDiff(α, β)
  $\theta \Leftarrow (\alpha - \beta) \mod (2\pi)$
  if $\theta < \pi$ **then**
    return θ;
  else
    return $\theta - 2\pi$
  end if
end procedure

Chapter 4

Implementation

4.1 Using ROS

As explained in `ros.org`, "The Robot Operating System (ROS) is a flexible framework for writing robot software. It is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms" For the purposes of this project, the following features of ROS were the most useful:

- ROS supports several programming languages, such as C++ and Python. In this project there are Python-based components such as TDNet, and C++-based components such as ORB-SLAM2 and FLaME. ROS provides a way to set up communication between these components that is easier to implement than alternatives such as python bindings or protobuf.

- The communication paradigm offered by ROS used most often in this project is the Publisher/-Subscriber model. In this model, a ROS "node" creates a `ros::Publisher` class, which publishes messages of a certain type in a certain "topic". Then, any other node running in the same system can subscribe to that topic by creating a `ros::Subscriber` class and passing a callback function that will be executed every time a new message is received from the publisher. This way, ROS takes care of all concurrency issues that the programmer would have to deal with when writing a conventional C++/Python program.

- Roslaunch can be used to initialize several nodes with specific configurations for their topic names and parameter values. This is very useful, for example when wanting to flexibly plug and unplug different components that advertise topics of the same type: It makes it easy, for example, to connect airsim's groundtruth depth to FIESTA, and then, by changing a few lines of code in the .launch file, swap the groundtruth depth by a depth produced by FLaME.

- Rosbag provides functionality to record, store and replay streams of data coming from any ROS topic. This can be very useful when creating repeatable tests.

- ROS comes with RVIZ, a powerful GUI app with a large set of features to visualize most built-in ROS message types. RVIZ can also be expanded with custom C++ plugins. For this project, two RVIZ plugins are used: The `flame_RVIZ_plugin`, that helps visualizing FLaME's meshes, and the `graph_RVIZ_plugin` that helps plotting graphs in real time from numerical streams of data.

ROS receives frequent criticisms, such as the fact that it uses non-user-friendly xml syntax on its .launch files, or that it tends to rely on outdated versions of tools like Python and OpenCV, giving rise to many dependency issues when trying to work with code that relies in newer versions of these tools, something that happened several times during the course of this project. In fact, it has been found that ROS is rarely used by professionals in industry. Some of ROS's criticisms have been addresses in ROS 2, it's newer version. However, for this project ROS 1 is used simply because the original code bases of the AirSim ROS wrapper, ORB-SLAM2, FLaME and FUEL all rely in ROS 1.

4.2 Implementing the mapper deliverable

4.2.1 Implementing the decoder and rectifier

Both of these modules were implemented using built-in ROS packages. The first one, `gscam`, takes a Gstreamer pipeline description, as well as a camera calibration file as input parameter, then it calls Gstreamer's C++ API to launch the pipeline and publishes a stream of images and of `sensor_msgs/CameraInfo` messages on the relevant topic. The second one, `image_proc`, subscribes to image and CameraInfo topics, and publishes a new stream of images rectified according to the distortion data in CameraInfo.

4.2.2 Feeding a stream of images into ORB-SLAM2 and FLaME

Connecting these two components to an image stream coming from AirSim, a rosbag replay or Gstreamer is not as straight forward as simply publishing the images in one end and subscribing to them in the other. This is because both the intrinsic parameters and distortion coefficients of the images being received must be known and given as parameters where appropriate. Fortunately, ROS comes with a camera calibration tool (see figure 4.1). The calibrator subscribes to a topic carrying images and asks the user to point the camera towards a checkerboard. Then it asks the user to move the camera to produce frames where the checkerboard appears from different points of view. When the algorithm is confident about its parameter estimation, it allows the user to produce a .ost file where all relevant data is printed out.

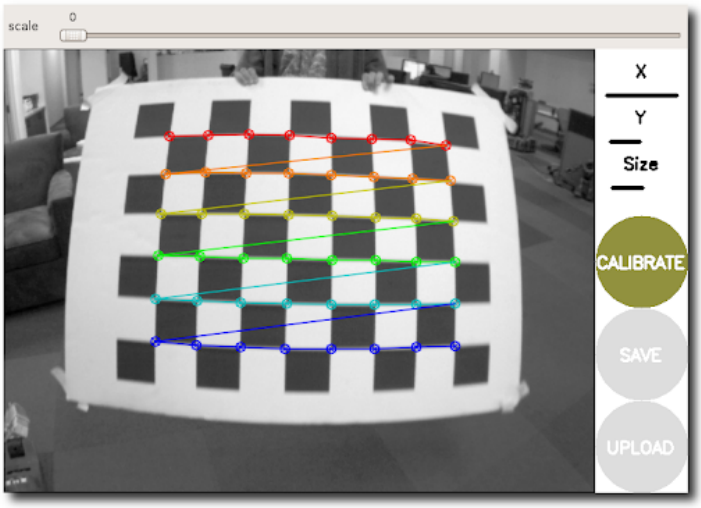


Figure 4.1: ROS camera calibrator.

4.2.3 Connecting ORB-SLAM2 to FLaME

A "FLaME nodelet" was provided with the original FLaME code. Most of the FLaME nodelet has been reused for this project, except for the `TrackedImageStream` class. This class' job is to subscribe to the corresponding transform, image and CamerInfo topics, and provide the FLaME nodelet with camera intrinsics information, as well as a thread-safe queue with instances of a `Frame` struct containing each of the received images together with its corresponding position, orientation and timestamp.

The original version of this class assumes that when an image is received, the corresponding transform for that image is already available within ROS's tf [50] tree. However, for this project, transforms do not become available until ORB-SLAM2 processes the new image. An initial attempt was made to solve this problem by splitting the original thread-safe queue into two thread-safe queues containing timestamped images and timestamped pose + orientation respectively. Code was then added to the main loop of the flame nodelet that would draw items from the two queues when their timestamps match. This solution was verified to work well, however, it was found later that ROS provides a more elegant solution to this problem, the `message_filters::Synchronizer`. A synchronizer object takes an arbitrary number of ROS subscribers as constructing arguments. Then, a callback can be registered to be called when messages whose timestamps fit a preestablished synchronizing policy are received.

4.2.4 Connecting ORB-SLAM2 and FLaME to FIESTA

As already mentioned, although FIESTA it has its own repository, it was actually implemented within a package named `plan_env` which came with the original FUEL repository. To use this package standalone without needing to launch the entirety of FUEL, a new executable named `test_sdf_map` was added to `plan_env`.

One challenge faced when putting these modules together was related to the different axis conventions used by different components. ORB-SLAM2, FLaME and FIESTA use optical, or RDF, coordinates (x=Right, y=Down, z=Forward). However, in order to follow the same convention as AirSim, the chosen world coordinate convention for maps in this project was RFU (x=Right, y=Forward, z=Up). The transformation matrix between these two axis conventions is expressed in equation 4.1. A pose + orientation message in one axis convention can be transformed into the other by pre-multiplying by this matrix.

$$T_{RDF \to RFU} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.1}$$

4.3 Optimizing the mapper's computational requirements

When simulating an environment, computational load is not necessarily a critical issue, as time itself can be simulated at different speeds. However, this project's final (although unfinished) goal would be for a Parrot Mambo to explore a real-world environment in a safe and agile manner. Optimizing the latency in which each individual frame can be processed, as well as the number of frames

per second that can be processed is critical for the following reasons:

- It will improve the robustness of the system: Both ORB-SLAM2 and FLaME are prone to failure (aka, loss of tracking) at low frame rates. This is because they rely on features within consecutive frames staying within the camera's field of view. Autonomously recovering from a loss of tracking event that deprives the system from odometry and/or depth data would be hard if not impossible to implement.
- It will improve the agility of the drone: When the time for a signal to go all the way around the closed loop pipeline is small, navigation can happen faster and reaction time to unexpected events will be shorter. This would be particularly helpful when navigating in environments with dynamic obstacles.
- It reduces hardware requirements of the system, potentially opening up opportunities for future work in which a base-station is no longer needed, and all computations can be performed onboard the drone.

4.3.1 CUDA-Accelerated ORB-SLAM2

ORB-SLAM2 yields, as previously mentioned, excellent results. However, the original code published by its authors, which runs completely in the CPU has several execution bottlenecks and inefficiencies. Thankfully, this is also a very popular and highly studied algorithm and several separate attempts have been made to accelerate it [51] [52] [53], mostly by parallelizing its most costly hotspots (namely feature extraction) using CUDA-enabled GPUs. For this project, a fork of the work by Yunchih Chen at Nottingham Trent University [53] is used. Yunchih works shows how CUDA-accelerated ORB-SLAM2 roughly doubles the framerate of the original ORB-SLAM2 when running on an Nvidia TX1.
One particular challenge faced that is worth mentioning, as it represents what was perhaps one of the technically hardest aspects of this project was the compilation of ORB-SLAM2-CUDA. The original code of this module was written to work with CUDA-enabled OpenCV 3. However, for this project Ubuntu 18 is used, and the CUDA toolkit version needed to compile CUDA-enabled OpenCV 3 is not easily available for Ubuntu 18. To go around this issue, CUDA-enabled OpenCV 4 was installed and the code of ORB-SLAM2-CUDA had to be modified to work with it. This, however, created a new issue, where `cv_bridge`, the package used by ROS to transform ROS image messages into OpenCV data structures and vice-versa, was only officially available for OpenCV 3. To solve this, a new package was created, "`cv_bridge_4`" that modifies the original `cv_bridge` code to work with OpenCV4.

4.3.2 ROS Nodelets

ROS offers a very convenient way to plug different modules together. However, one mayor disadvantage of the traditional ROS model is the way that communication happens between nodes via TCP. This works well for many purposes, but when it comes to messages containing large amounts of data such as raw images or point clouds, the process of packing, sending, receiving and unpacking a message can become a computationally intensive task. Sending the 640x480 frames used in this project between two nodes would involve transmitting 307,200 pixels via TCP. When two modules such as FLaME and FUEL are running in the same computer, then it would be much less wasteful if, for example, FUEL could access the depth images served by FLaME directly from the RAM. ROS

Nodelets facilitate this by allowing communication to happen via C++ shared pointers rather than TCP.

FLaME is the only component of this project that was written as a ROS nodelet by the original authors. ORB-SLAM2-CUDA, the AirSim ROS wrapper and FUEL were provided as conventional ROS nodes, but nodelets have been written for each of them. The steps to convert a node into a nodelet are usually as follows:

- The nodelet package must be indicated as a dependency both in the package manifest "package.xml" and in CMakeLists.txt
- The necessary "#includes" for `nodelet.h` and `pluginlib/class_list_macros.h` must be added.
- ROS nodes have a `int main()` function, like any other C++ executable program, to indicate the compiler where process execution must start. However, nodelets must instead define a new class inheriting from `nodelet::Nodelet` which overrides its `OnInit()` function. This will be used as the nodelet's entry point (see listing 4.1).
- ROS nodes run as full processes, and are therefore indicated as executable targets to CMake, using `add_executable()`. Nodelets are library targets instead, and therefore `add_library()` must be used in CMakeLists.txt.
- It must be ensured that subscribers set up their callbacks with `ConstPtr` arguments.
- It must be ensured that publishers do not modify a message after calling `ros::Publisher::pubilsh()`. The best way to achieve this is to store messages and all their fields in the heap, using a `shared_ptr`, then calling `publish()`, and never modifying the message's content after that. Listings 4.2 and 4.3 show an example of how the process of publishing a point cloud had to be modified within FUEL's codebase to prevent runtime errors when using the FUEL nodelet.

Finally, the TDNet node has not yet been implemented as a nodelet. The TDNet ROS wrapper was written in Python, as this was the quickest way to do so within the time frames of this project. Python is a high level language that does not offer the low-level features that nodelets make use of, and therefore the only way to optimize communication between TDNet and the other components of the system would be to rewrite the TDNet ROS wrapper in C++. This would come with a few difficulties, like having to use switch between conventional PyTorch and PyTorch's C++ API.

Listing 4.1: The FUEL nodelet.

```
1  #include <ros/ros.h>
2  #include <nodelet/nodelet.h>
3  #include <exploration_manager/fast_exploration_fsm.h>
4  #include <pluginlib/class_list_macros.h>
5
6  class ExplorationNodelet : public nodelet::Nodelet {
7  public:
8      ExplorationNodelet()=default;
9      virtual void onInit() {
10         ros::NodeHandle& nh = getPrivateNodeHandle();
11         expl_fsm.init(nh);
```

```
12      }
13      FastExplorationFSM expl_fsm;
14  };
15  PLUGINLIB_EXPORT_CLASS(ExplorationNodelet, nodelet::Nodelet)
```

Listing 4.2: Publishing a point cloud within the original FUEL code.

```
1  pcl::PointCloud<pcl::PointXYZ> cloud;
2  for (int i = 0; i < n; ++i)
3  {
4  cloud.push_back(point_cloud_.points[i]);
5  }
6  sensor_msgs::PointCloud2 cloud_msg;
7  pcl::toROSMsg(cloud, cloud_msg);
8  depth_pub_.publish(cloud_msg);
```

Listing 4.3: Code modified for publishing pointcloud within FUEL's nodelet. The previous code would raise a runtime error when running as part of the FUEL nodelet.

```
1  std::shared_ptr<pcl::PointCloud<pcl::PointXYZ>>
2      cloud(new pcl::PointCloud<pcl::PointXYZ>);
3  for (int i = 0; i < n; ++i)
4  {
5  cloud->push_back(point_cloud_.points[i]);
6  }
7  sensor_msgs::PointCloud2Ptr cloud_msg(new sensor_msgs::PointCloud2);
8  pcl::toROSMsg(*cloud, *cloud_msg);
9  depth_pub_.publish(cloud_msg);
```

4.4 Implementing the explorer deliverable

4.4.1 Connecting the AirSim simulation to FUEL

AirSim environments tend to be highly realistic and therefore computationally heavy. Furthermore, AirSim was developed by Microsoft, and therefore several facilities exist when using windows to create and run AirSim environments. For these two reasons, it was decided to run AirSim environments in a separate Windows laptop. Establishing communications between the two computers was as simple as specifying the IP address of the computer running the AirSim environment when launching the AirSim ROS wrapper node. This node wraps around the AirSim C++ API by publishing, topics with the drone's odometry and camera image streams, as well as recieving subscriptions for drone velocity commands.

Finally, the loop between FUEL and AirSim was closed by making use of the controller: The controller was implemented as a ROS node that subscribes to trajectory messages from FUEL and odometry messages from AirSim, and publishes velocity commands back to the AirSim ROS wrapper

node. Rather than writing this node from scratch, the controller used by the demo that comes from FUEL was copied and modified according to the design mentioned in the previous chapter.

4.4.2 The TDNet ROS wrapper

The TDNet authors provide pre-trained weights and sample code to perform inference using a folder of image files. For TDNet to work within this project, a Python ROS wrapper was written. The wrapper subscribes to images, which need to be resized, normalized and turned into numpy arrays, to then be fed into TDNet. The result is published in two different topics. The topic `/semantic` publishes the raw output of TDNet as monochromatic 8-bit images that can be used by FIESTA to semantically label its map's voxels. The topic `semantic_color` feeds each 8-bit pixel of TDNet's output into a function that maps each gray level into an RGB level. This topic can then be displayed in RVIZ for visualization purposes.

The TDNet wrapper needs Python 3 to run. However, ROS only officially supports Python 2, and packages like `cv_bridge` that come built in with ROS only work with Python 2. To overcome this, cv_bridge had to be built from source, specifying the appropriate Python interpreter in the CMake options.

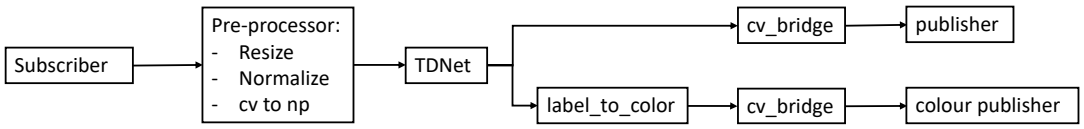


Figure 4.2: Overview of the TDNet ROS wrapper.

4.4.3 Adding semantic labels to the voxel map

It is worth noting that an option is given to the user on whether to use or not the semantics extension. A new boolean ROS parameter `map_ros/do_semantics` has been introduced to indicate whether the original or the extended version of FUEL should be executed.

The following steps were followed to implement semantically-labelled voxels:

1. New subscribers and synchonizers were added to the `MapROS` class. The new synchronizers will listen for pose, depth images and segmentation images at the same time, and call the corresponding callback when messages with similar timestamps are received for all three topics.

2. The function used to create a point cloud from depth images (look back at equation 2.2 for reference) was modified to create a labelled point cloud of type `pcl::PointCloud<pcl::PointXYZL>` where the label field is filled with the value of each pixel in the semantically segmented image.

3. The semantically labelled point cloud is then fed to a function that uses it to update the occupancy map. This function has been modified to recalculate the label of voxels set as occupied every time a new cloud is received by calculating the mode of all the points from the pointcloud that mapped to that voxel.

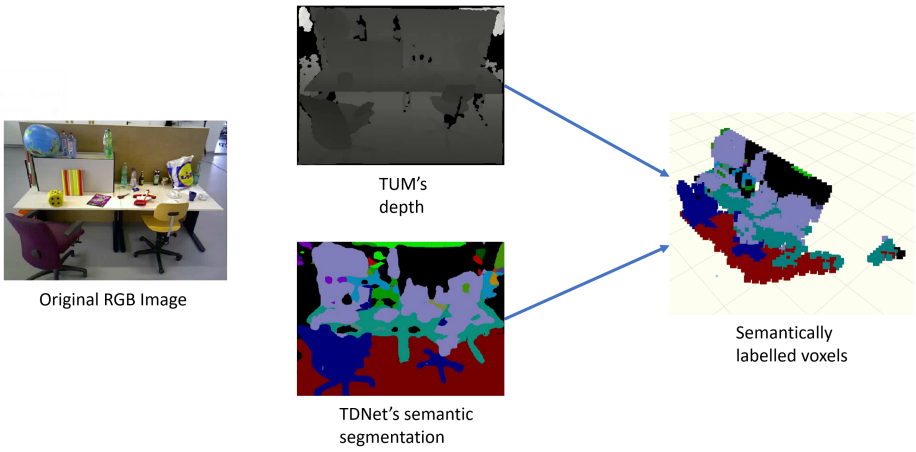


Figure 4.3: Creating a semantically labelled voxel map using the TUM datset's depth images and TDNet's semantic segmentation.

4.4.4 Using semantic voxel labels to tweak obstacle inflation

The original code in `sdf_map.cpp` takes a ROS parameter called `obstacle_inflation`. When a voxel is labelled as occupied, the function `SDFMap::clearAndInflateLocalMap()` will, among other things, label every voxel within `obstacle_inflation` distance of an occupied voxel as occupied. This function has been modified so that a new parameter, called `person_obstacle_inflation` is instead applied when the voxel's semantic label corresponds to that of a person, while the original `obstacle_inflation` is applied to every other voxel.

Chapter 5

Testing

5.1 Testing the mapper deliverable

When putting together the different components that form the mapper, functional correctness was verified by using the EuRoC and TUM datasets that the original authors of ORB-SLAM2 and FLaME already made use of on the demos that come with their original code. For instance, samples of the EuRoC dataset were fed first to the FLaME demo and then to the combined ORB-SLAM2 and FLaME to verify that the versions of ORB-SLAM2 and FLaME modified as explained in the previous chapter were working well when put together. The mapper was also tested by feeding it both AirSim and real-world video. The results will be discussed in the next chapter.

5.1.1 The profiler

Intel's VTune profiler is a powerful tool that can be used to look for any bottlenecks and assess how much CPU usage every function is taking. It also produces data on how efficiently a program is using the CPU; programs can often improve their computational efficiency by, for example, scheduling its operations in a way that it keeps the Fetch-Decode-Execute stages of the CPU always as busy as possible, or making use of advanced micro-architectural features such as vector operations. Profiling whole ROS systems is a challenge, because ROS uses several separate processes. Usually VTune is used to profile each process independently, but instead, a system-wide profile, which profiles every process in the operating system was used. System-wide profiles have a few limitations, as they use "Hardware-event based sampling" rather than "User-Mode Sampling". In practice this means that relations between callers and callees cannot be made, and therefore less information is available on the relations between different functions.

5.1.2 The Performance Meter Node

A C++ ROS node has been written from scratch to measure latency at each stage of the mapper. Each node (gscam, ORB-SLAM2, FLaME, FIESTA, TDNET) sends a `std_msgs::Header` containing the stamp that identifies each frame every time they recieve or finish processing a frame. The performance meter node listens to these header messages, makes all relevant computations and publishes the fps and latency for each component in the form of `std_msgs::UInt8` and `std_msgs::Float64` respectively. These messages can then be plotted in real time using `graph_RVIZ_plugin` [54]

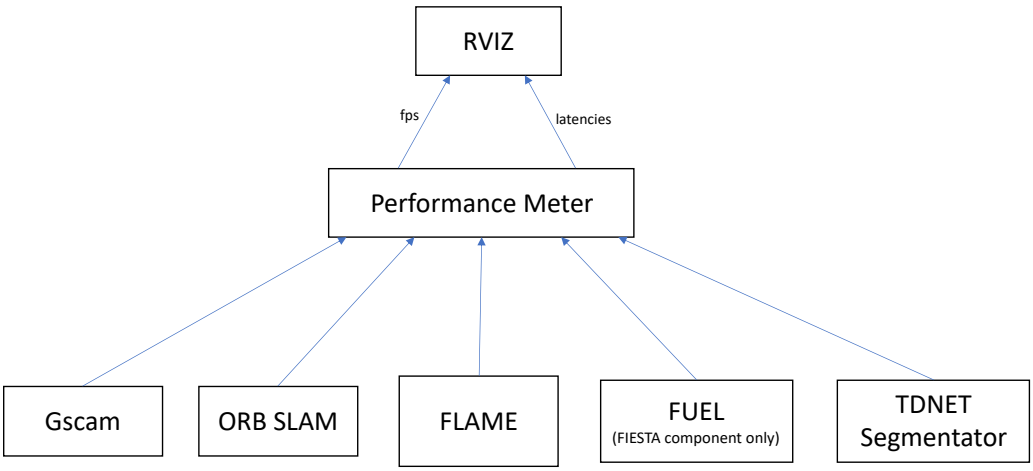


Figure 5.1: The Performance Meter node.

5.2 Testing the explorer deliverable

5.2.1 Testing the Fuel-Controller-AirSim closed loop

FUEL publishes several topics for visualization purposes such as frontiers, planned bspline trajectories, camera's field of view, etc, which together with other tools like the GDB debugger helped the testing process. The closed loop has been successfully tested for functional correctness on several PEDRA [42] environments. These are small environments of both indoor and outdoor scenes with a wide range of complexities in terms of obstacles and optical features which proved extremely useful to run tests on.

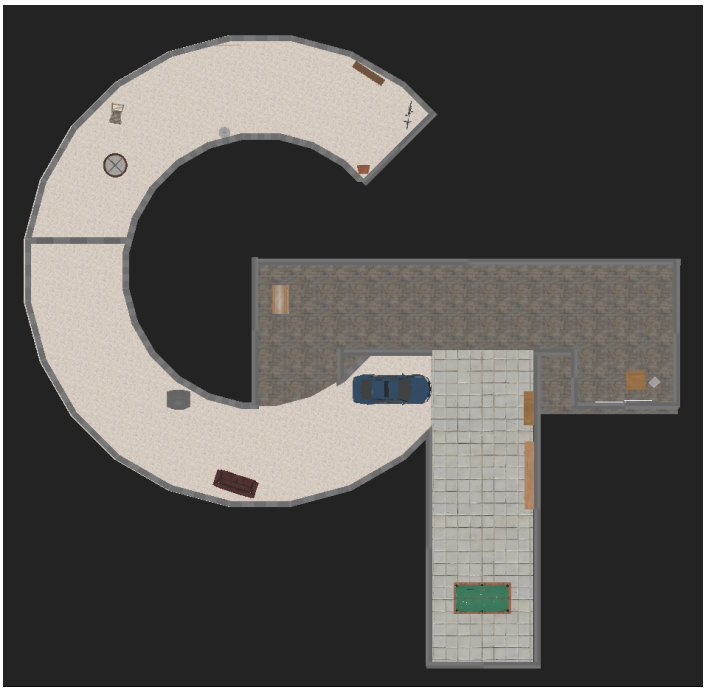


Figure 5.2: The "GT" pedra environment.

5.2.2 Creating custom AirSim environments

In order to test the behaviour of the drone when navigating around obstacles with different semantic labels, a small environment with a high density of static obstacles of different types is desired. An environment fitting this description was built using Microsoft's "Blocks" environment as a template. AirSim environments can be edited using the Unreal Engine editor (see fig. 5.7). The editor comes with a marketplace feature where texture packs and other project extensions can be downloaded. In this case, free extension packs for person characters and furniture were found and added to the project. The final result was a small environment consisting of randomly placed people and furniture enclosed between four walls (see fig. 5.4)

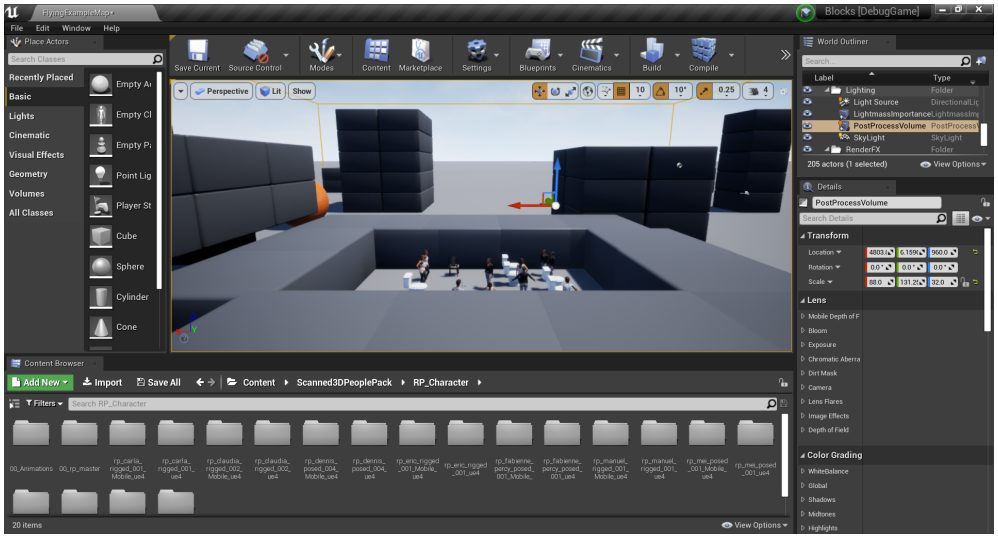


Figure 5.3: The unreal engine editor.

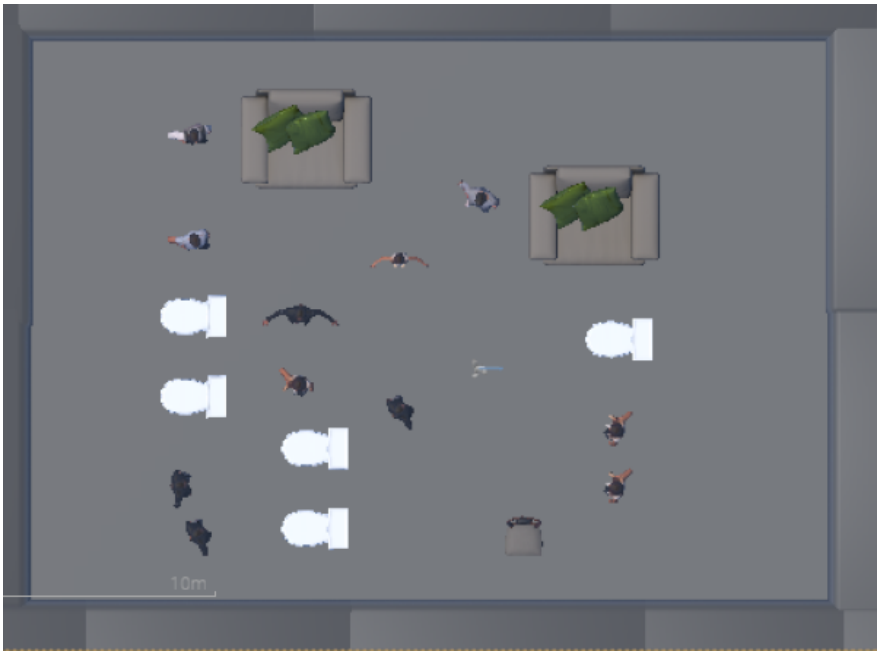


Figure 5.4: The "Semantic Obstacles" Airsim environment created with the unreal engine editor.

5.2.3 Visualizing semantic voxels

The function `MapROS::publishMapAll()` is used to publish a point cloud where each point corresponds to an occupied voxel. This cloud can then be visualized as a voxel map in Rviz by configuring the desired shape, size and color of each point. `MapROS::publishMapAll()` has been extended so that it publishes two point cloud: The original point cloud without semantic labels (see figure 5.5), an RGB point cloud (see figure 5.6), where the mapping from semantic label to RGB color is produced by the same function used in the TDNet ROS wrapper (see section 4.4.2).

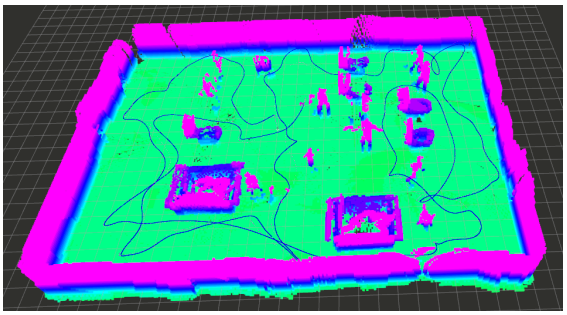


Figure 5.5: Visualizing an occupancy map of the Semantic Obstacles environment with semantics off. The color indicates z axis position.

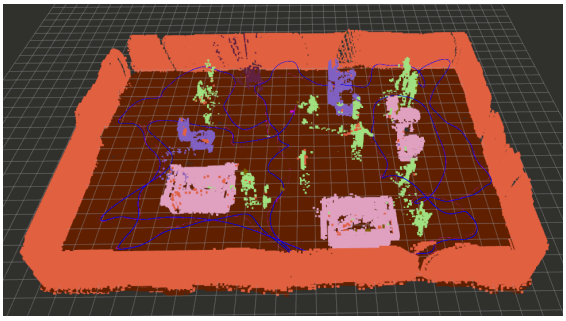


Figure 5.6: Visualizing an occupancy map of the Semantic Obstacles environment with semantics on.

5.2.4 Visualizing the ESDF

The ESDF is a 3D field where every voxel in space, whether empty or not empty, has a value. This can be hard to visualize. The function `MapROS::publishESDF()`, which came within the original FUEL code, publishes a labelled point cloud of the values of the ESDF at a certain z coordinate. This can then be esaily visualized in RVIZ.

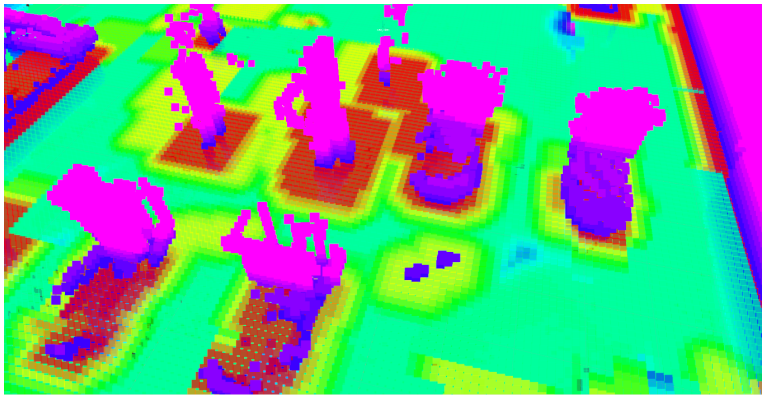


Figure 5.7: Visualizing a slice of the ESDF from the Semantic Obstacles environment.

Chapter 6

Results

6.1 How well are environments mapped?

The mapper's performance in terms of the accuracy of the map produced was seen as largely constrained by issues with FLaME. Figures 6.2 and 6.1 illustrate some common issues found. Firstly, in figure 6.2, the left hand wall and the ceiling have extensive and correct mesh estimations, while the right hand wall and the floor show sparser mesh predictions. FLaME's performance has been observed to be highly dependent on the texture of the surfaces being mapped. Secondly, in figure 6.1, the further away parts of the floor are predicted as being closer to the camera (the mesh is orange rather than blue). Similar results have been observed every time an environment with a heavy presence of reflections and other optical properties is mapped.

Attempts were made to improve FLaME's performance by tweaking some of its parameters. FLaME's authors explain that `min_grad_mag`, `win_size` and `nltgv2/data_factor` are particularly important parameters. Significant improvements were seen to FLaME's ability to map fine details by decreasing `win_size`, however, this came at the expense of a higher latency and didn't fix the issues with texture and optical features already mentioned.

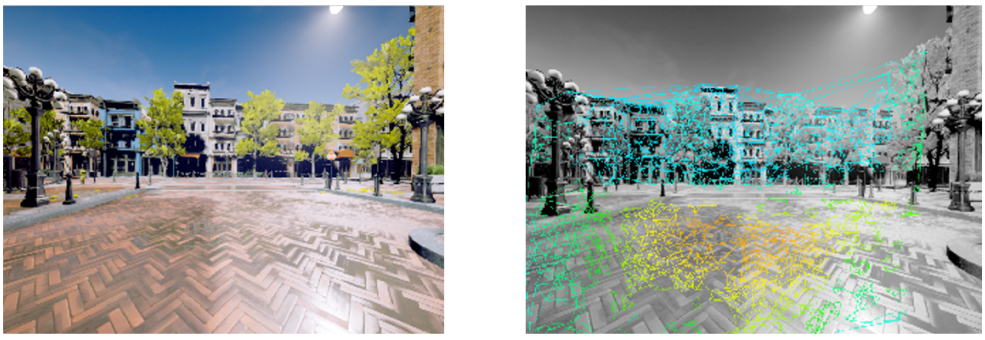


Figure 6.1: Original image (left) and flame-estimated depth mesh (right) from a section of PEDRA's "Town" environment.

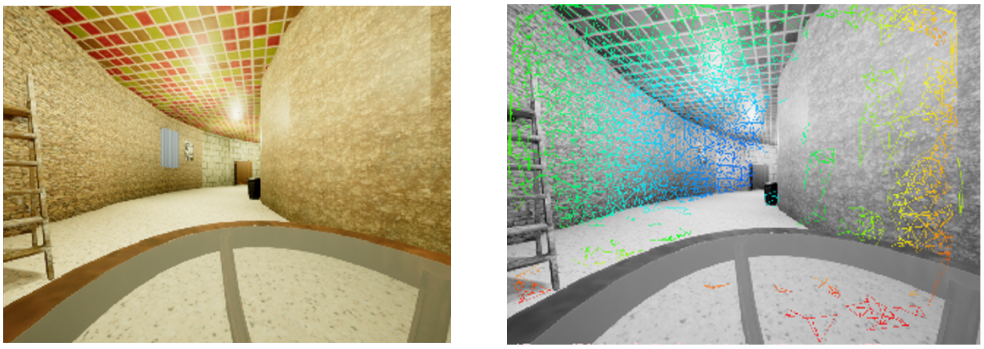


Figure 6.2: Original image (left) and flame-estimated depth mesh (right) from a section of PEDRA's "GT" environment.

The observations made above are translated into a somewhat poor result when trying to create voxel maps. Firstly, attempts were made to reconstruct PEDRA's GT environment by manually controlling the drone using an XBox controller while feeding a stream of monocular images from AirSim into the mapper (see figure 6.3). Secondly, attempts were made to map real-world spaces in the EEE building at South-Kensington campus (see figures 6.4 and 6.5.). One additional observation made while generating these maps was that FLaME tended to loose tracking of some regions as they move out of the camera's field of view. Therefore, voxels firstly successfully recognized as occupied would end up being labelled as unoccupied as they move out of the camera's field of view. One possible way to solve this issue would be to ignore FLaME's depth predictions near the edges of the depth image.

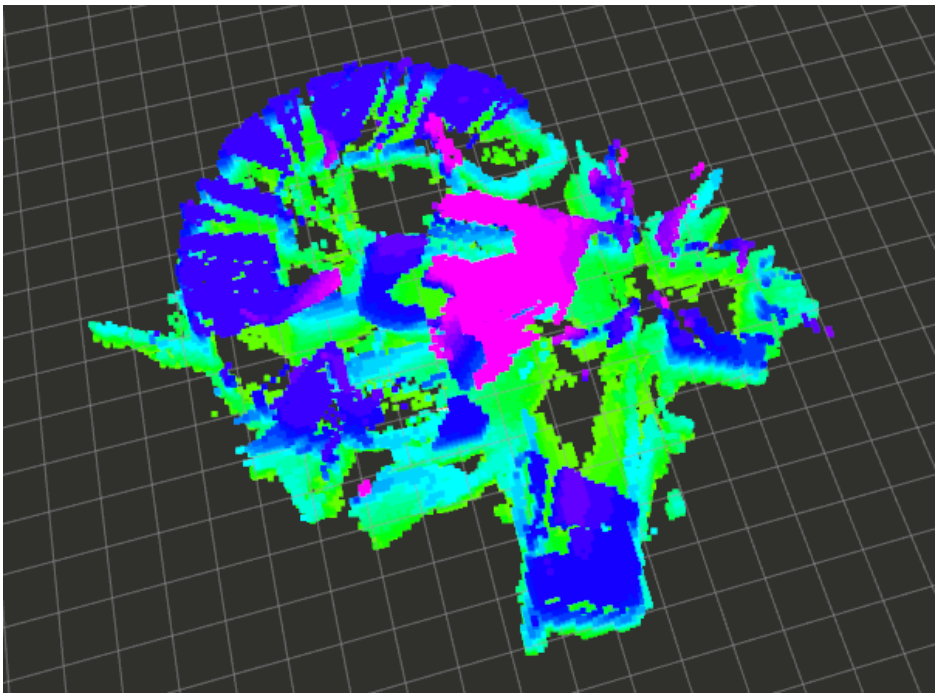


Figure 6.3: Reconstructing PEDRA's GT environment with the mapper (see the original map of GT in figure 5.2).

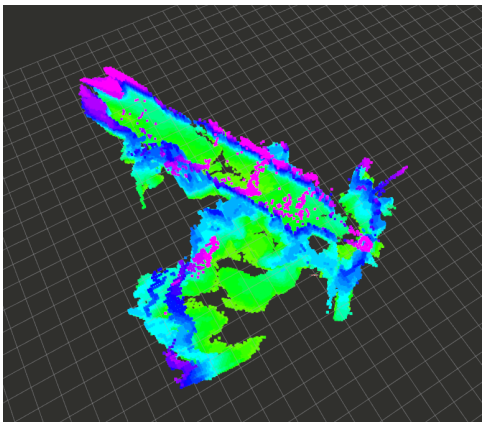


Figure 6.4: Reconstructing room 905 and the corridor of floor 9 at the EEE building in South Kensington Campus.

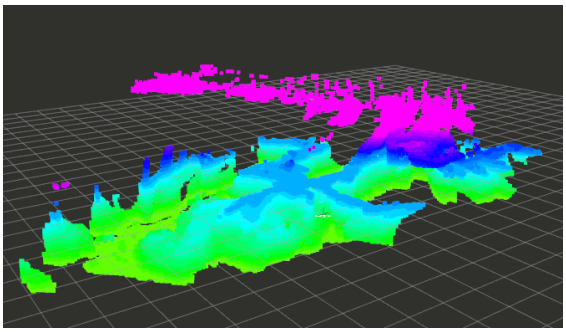


Figure 6.5: Reconstructing room 905, the east side stairs, and the corridor of floor 10 at the EEE building in South Kensington Campus.

6.2 The mapper's computational performance

6.2.1 Profiling results

Both ORB-SLAM2 and ORB-SLAM2-CUDA were profiled while processing a video sequence from the EuRoC dataset. The results confirmed what others ([51], [52], [53]) had found: In the original ORB-SLAM2, the task taking most CPU time is feature detection (more specifically, the function `ORB_SLAM2::Frame::ExtractORB`). VTune showed a derease in CPU usage from this function from 18% in ORB-SLAM2 to 9% in ORB-SLAM2-CUDA.

A system-wide profile was also performed while running the whole mapper. This revealed that the `MapROS::PublishMapAll` function was the most intensive function, issuing 17% of all CPU instructions. This function is called by a `ROS::Timer` every 100ms, and its only purpose is visualization; it iterates through every voxel in the map, checks whether it's occupied, and adds it to a point cloud message if so. These tests were run with a map size of 70x70x10 and a resolution of 0.1, which means 49 million voxels have to be checked every time this function is called. Because this function is for visualization purposes only, a big performance improvement could be obtained by either decreasing the frequency in which `MapROS::PublishMapAll` is called, or totally removing it using `MapROS::PublishMapLocal` for visualization instead, which publishes only a smaller section of the map.

Attempts were made to find differences in the system-wide profiling results between the "nodelets only" and the "nodes only" version of the mapper, but given the limited amount of information on function calls provided by this type of profile, no insights were found.

6.2.2 Measuring latencies with the performance meter

Latency tests were run with the `MapROS::PublishMapAll` function removed. A rtsp stream of 30fps was first set up, and semantics were initially turned off. As seen in figures 6.6 and 6.7, every component was capable of keeping up with the 30fps, and no evidence was seen of frames being

dropped. When running the nodelet only version of the mapper, a total latency between 75 and 110 milliseconds was recorded. However, latency measurements for FLaME were observed to slightly increase when movement between frames became larger.

When using the nodes-only version of the mapper, latencies of around 6ms were measured to take place when transporting 640x480 images between nodes. The addition of these latencies accross all nodes in the pipeline explains that when running the same experiments with the nodes only version of the mapper, total latency measurements increased from an average of around 90ms to one of around 120ms.

When turning semantics on, the system was observed to be unstable, and the rtsp stream had to be reconfigured to transmit at 7fps. Figure 6.8 shows that a latencies were typically recorded in the range between 190ms and 260ms . All these results are summarized below:

nodelets	semantics	fps	average latency (ms)
off	off	30	120
on	off	30	90
on	on	7	225

Table 6.1: Average latencies.

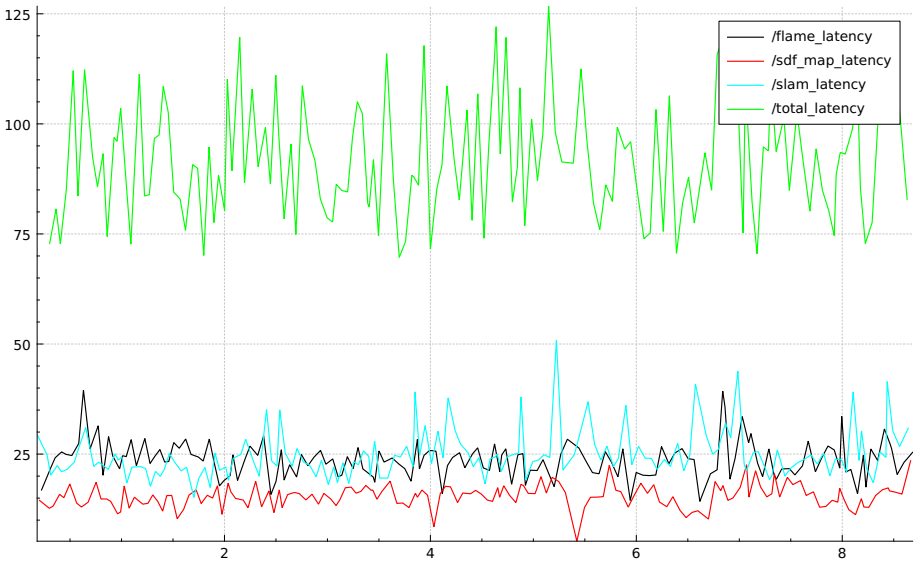


Figure 6.6: Latencies plotted when mapping while receiving a 30fps stream. Semantics were off and nodelets were on.

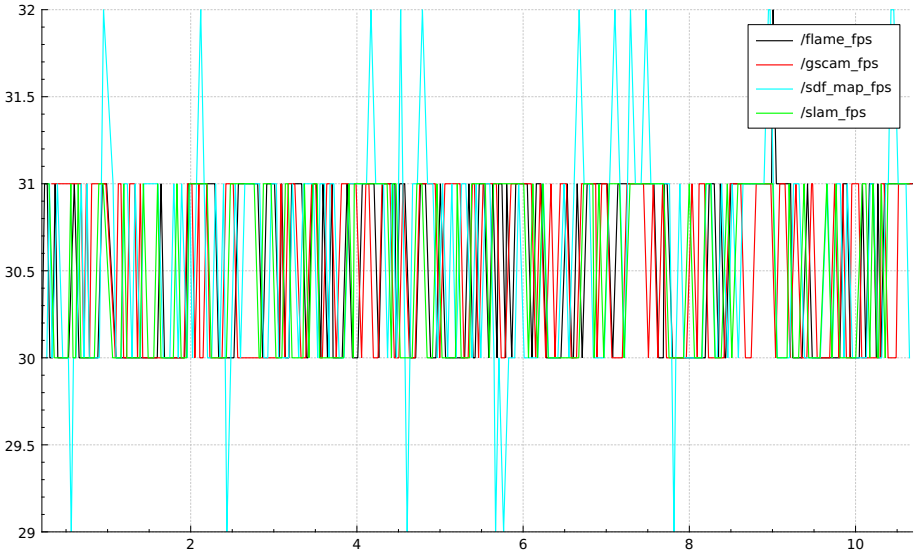


Figure 6.7: Fps plotted when mapping while receiving a 30fps stream. Semantics were off and nodelets were on.

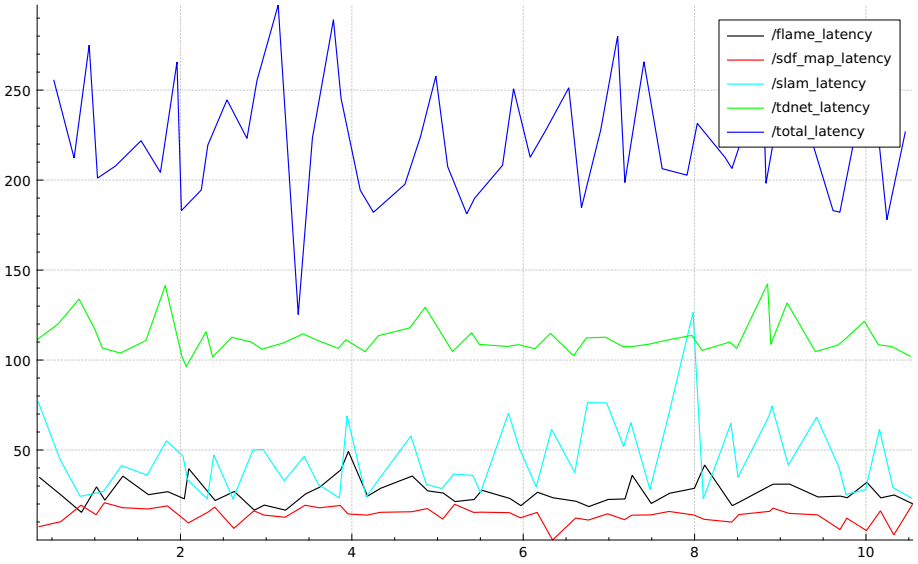


Figure 6.8: Latencies plotted when mapping while receiving a 30fps stream. Semantics were on and nodelets were on.

6.3 How well is exploration executed?

When a control loop was first established with AirSim, FUEL was verified of being capable of fully exploring the PEDRA environments. It was observed, however, how the exploration process was taking longer and was overall more erratic than it would be expected from the claims made in FUEL's paper. Figure 6.9 shows an initial exploration of the GT environment.

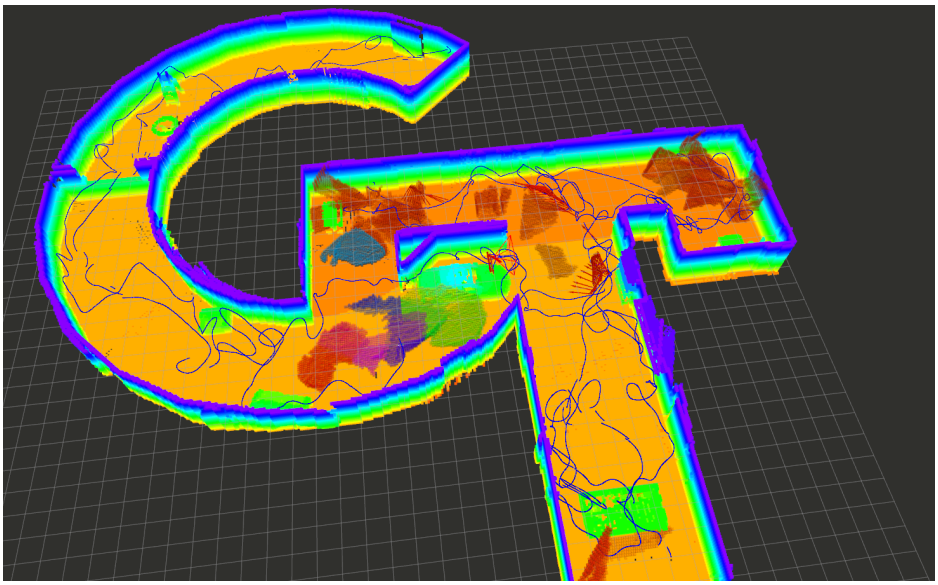


Figure 6.9: GT environment exploration using the original FUEL parameters. The blue line has been formed from the position component of the trajectories outputted by FUEL.

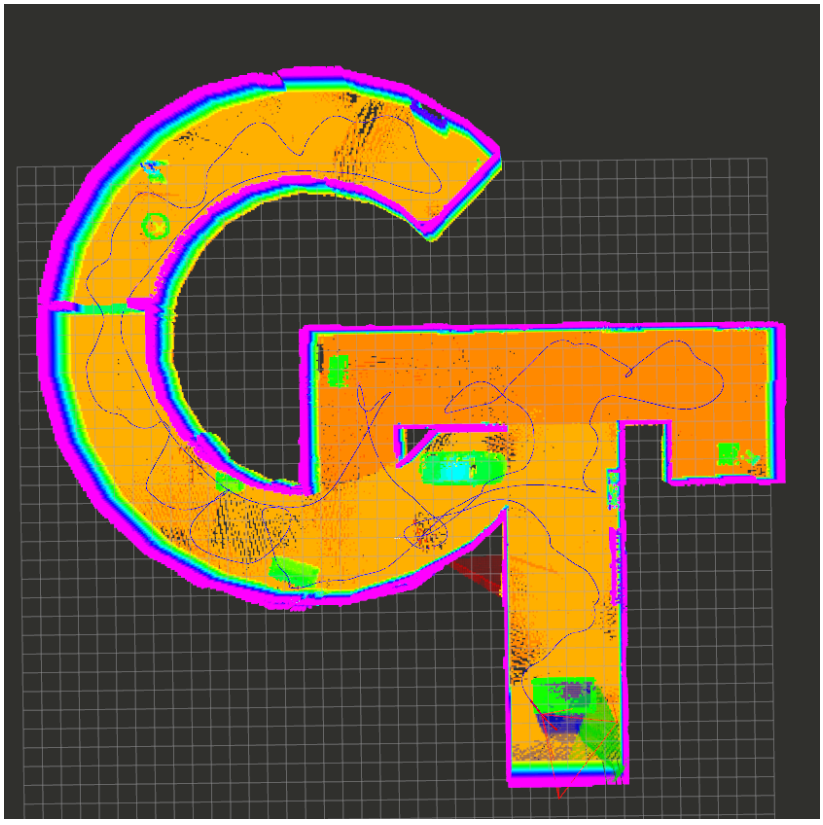


Figure 6.10: GT environment exploration path after tweaking FUEL's parameters.

Attempts were made to better understand FUEL's parameters, and soon it was concluded that FUEL's performance is highly dependent on some of these parameters being appropriately tuned depending on the characteristics of the drone, and on the nature of the environment being explored.

After parameter tuning, a faster and smoother exploration of the GT environment was produced (see fig. 6.10). The following FUEL parameters were modified from their original values:

- `frontier/cluster_size_xy`: Because the GT environments has larger sections of empty space than the simulation provided in the original FUEL code, this parameter was increased to prevent the drone from having to travel to many small frontiers that can in reality be discovered from the same viewpoint.

- Field of view parameters: These were changed to account for the wider and deeper field of view that the AirSim depth camera has.

- `max_vel`, `max_acc` and `max_jerk`. The AirSim drone was configured for a maximum speed and acceleration of 0.5m/s and m/s^2 respectively. FUEL also has parameters for maximum speed and acceleration that had to be adjusted accordingly. The maximum jerk (m/s^3) FUEL parameter also had to be reduced. The authors recommend that when these parameters are changed, others such as `search/max_tau` and `search/resolution_astar` should also be increased or decreased. This was found to be necessary, but it is yet not understood why.

6.4 The controller's critical role

The controller was observed to play a critical role on the following aspects of the exploration process:

- Speed: If the controller fails to keep up with the desired trajectory, the drone will be more likely to miss the viewpoints that FUEL desires to visit. FUEL was seen to be capable of successfully recovering from such scenario by ordering the drone to go back to the desired viewpoint, however, this will still negatively impact the speed and efficiency of the exploration process.

- Safety: When the controller fails to keep up with the desired trajectory, the drone's behaviour becomes unpredictable, thus the drone will be more likely to collide with obstacles. This issue can be mitigated by increasing the obstacle inflation parameter.

The following constant parameters (look back at section 3.10 for reference) were found to work well:

$$K_v = 0.5 \tag{6.1}$$

$$K_x = 0.85 \tag{6.2}$$

$$K_\psi = 1 \tag{6.3}$$

The controller was never found to prevent the drone from fully exploring an environment, however, as seen in figure 6.12, a few instances were observed where the actual drone trajectory deviated from the desired trajectory by a significant amount.

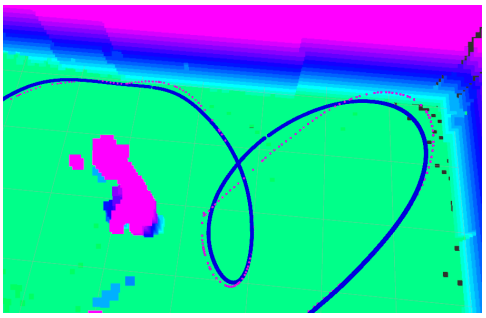


Figure 6.11: Instance when actual and desired trajectory match well.

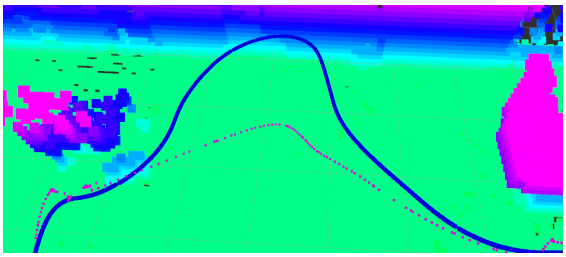


Figure 6.12: Instance when actual and desired trajectory don't match well.

6.5 The semantic extension's performance

6.5.1 Building semantic maps

Figure 6.14 shows a region of the Semantic Obstacles environment explored with FUEL when using TDNet's segmentation and AirSim's ground truth depth. Several observations have been made:

Firstly, several features of the map are correctly identified by TDNet. In figure 6.14, the two women (■), the chair (■) and the walls (■) are mostly correctly labelled with their true colors. The floor is partially labelled with its true color (■).

Secondly, on still images, TDNet's performance was found to be significantly lower when testing on the Semantic Obstacles environment than when testing with videos from the NYUD or TUM datasets (see figures 2.12 and 4.3 respectively). This is likely to be because of a failure of TDNet to generalize on scenes that look different from the indoor apartment environments featured in the NYUD dataset, which the pretrained model used in this project was trained with. A way to improve performance on AirSim environments like this one would be to retrain TDNet using AirSim's ground truth semantics from different environments.

Thirdly, TDNet's performance was found to be highly dependent on the position of the camera; as objects leave the field of view of the camera, TDNet would tend to stop labelling them correctly, giving rise to the situation where voxels discovered for the first time are labelled correctly, only to be left with an incorrect label as the camera stops pointing at them. The explanation for this could be a failure of the NUYD dataset to include a large enough variety of camera angles for different scenes, as well as a failure to include enough scenes containing objects partially out of the field of view of the camera. Training TDNet on video sequences of AirSim footage taken while FUEL explores could make TDNet learn to make better predictions on these camera angles. Another possible solution could be replacing the current voxel labelling algorithm with a more sophisticated solution that takes into account the way that a voxel has been labelled in the past, giving less importance to labels produced when the field of view was moving fast or a given voxel was leaving the field of view.

Figure 6.13: Section of the Semantic Obstacles environment mapped using ground truth depth and TDNet.

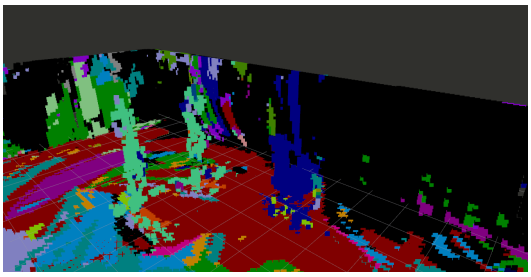


Figure 6.14: Section of the Semantic Obstacles environment mapped using ground truth depth and TDNet.

6.5.2 Navigating around people

Figure 6.15 shows the semantically labelled occupancy map formed from AirSim's ground truth depth when placing the drone in front of three obstacles. The person obstacle, labelled with the color "■", is shown to have a larger red (zero distance) area around it when plotting a slice of the ESDF. When using AirSim's ground truth segmentation and setting FUEL to explore the Semantic Occupancy environment, FUEL was shown to successfully explore the Semantic Obstacles environment while keeping a larger distance from person obstacles (see figure 6.16).

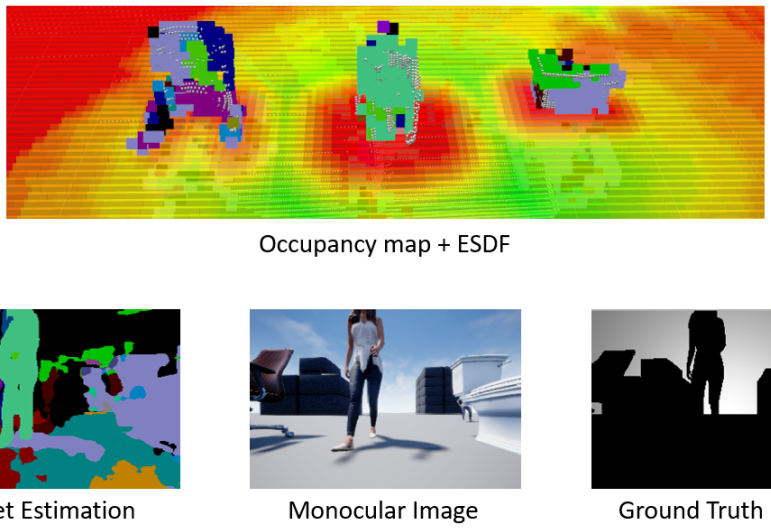


Figure 6.15: ESDF slice around three obstacles: Chair in the left, person in the middle, toilet in the right.

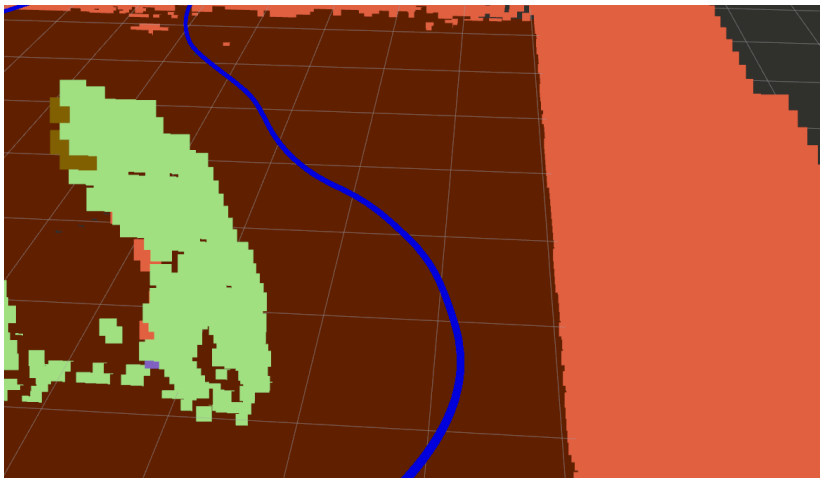


Figure 6.16: Trajectory followed when exploring a region of the Semantic Obstacles environment between two obstacles: A person on the left and a wall on the right.

Chapter 7

Evaluation, Conclusions and Further Work

7.1 Contributions and Limitations

In summary, these are the contributions made in this project:

- FUEL has been set up to work with the AirSim, a popular drone simulation for autonomous vehicle research. FUEL has also been shown capable of efficiently exploring large, highly realistic AirSim environments after doing some parameter tuning.

- The `plan_env` component of FUEL (aka, FIESTA) has been expanded to add semantic labels to occupied voxels when receiving a stream of semantically segmentated images from TDNet. A case study has been proposed and tested where the obstacle inflation parameter can be selectively adjusted depending on a voxel's semantic label.

- FUEL's semantic extension has also been tested with TDNet, a semantic segmentator. The results of the semantic maps obtained when using TDNet have room for improvement. As already mentioned in the results chapter, improvements could be made retraining TDNet using AirSim environments and/or modifying the way in which voxels are semantically labelled.

- A mapper capable of reconstructing 3D occupancy maps from monocular images has been developed by putting together ORB-SLAM2, FLaME and FIESTA. The mapper has been optimized to run in a laptop in real time, but in its current state its mapping performance has been found to have several limitations, and further work would be needed to allow for full monocular exploration. Unfortunately, no improvements have been implemented with respect to the work by HKUST [8].
  FLaME's authors claim that they used their work to enable obstacle avoidance in a monocular drone in both indoor and outdoor scenarios. Therefore, it is unclear whether by doing some parameter tuning and modifying the way in which FLaME's depth estimations are treated, this mapper could still be used to enable autonomous monocular exploration. If not, there are a few alternatives such as MVDepthNet [29], by the same authors that produced FUEL and FIESTA that look like promising depth estimator alternatives.

7.2 Connecting the deliverables

As an estimate, between 2 and 4 weeks of full-time work would be necessary to convert the two deliverables produced in this project to the final system described in figure 3.1. To do this, the following steps would have to be taken:

1. Deliverable number 2 forms a closed loop, but this system also assumes that depth and odometry data will be available from frame one. This will not be the case if ORB-SLAM2 and FLaME are used instead of AirSim's ground truths. Instead, assumptions must be made about the initial position of the drone being distant from any potential collision points, and the controller must be extended to order the drone to move slowly following some predefined pattern until ORB-SLAM2 starts sending odometry messages.

2. ORB-SLAM2 cannot estimate scale. The scale of the constructed map always depends on the frames where ORB-SLAM2 first initialized tracking. For this system to work for arbitrary real-world to map-world scales, additions would have to be made to the controller. As an initial guess, an adaptive controller capable of tuning its parameters to different scales during the first few frames to of exploration could be the solution to this. Another possible solution could be to substitute ORB-SLAM2 with Inertial-Visual SLAM frameworks such as VINS-Mono [9], which can provide real-world scale estimates.

3. It is believed that certain parameters within FUEL such as maximum yaw rate would have to be fine-tuned when using ORB-SLAM2's odometry and FLaME's depth. This is because both ORB-SLAM2 and FLaME can loose tracking if frames become blurred or change a lot from one another, specially at lower FPS. It is believed that more complex parameters would also have to be fine-tuned for the system's navigation and exploration performance to be closer to that of a system that uses AirSim's ground-truths. For example, FUEL could be ordered to avoid facing the camera too closely to obstacles or to parts of the environment where ORB-SLAM2 and FLaME are detecting very few features. The idea is to maximize the probability of loss of tracking events occurring, as recovering from these would be very hard.

4. The final step would consist in making the jump from the simulation world (fig. 3.10) to the real world (fig. 3.1). This would involve connecting the Gstreamer and Rectificator modules, and creating a new controller that can pass commands that PyParrot (the python library used to controll the parrot mambo) understands.

7.3 Future Work

Some possible directions in which this project could be expanded are as follows:

- Expand FUEL's semantic extension: In this project, a simple case study is proposed where semantic segmentation allows for the selective modification of the ESDF. It is believed that other components within FUEL's hierarchy could be benefited from the semantically labelled voxels, and therefore a lot of opportunities for further work exist in this area. As an example, semantics could give FUEL a better idea of what frontiers is worth exploring before others: Suppose that FUEL was exploring an environment with two frontiers, one corresponding to a fly of stairs and one corresponding to a door. In this scenario, it is more likely that the door

leads to a room that doesn't produce many more extra frontiers, while the stairs will likely lead to another floor and therefore to a larger space. If the stairs are chosen, FUEL is likely to produce a less efficient exploration because of the time necessary to return to the door's frontier after exploring the larger space.

- FIESTA's capability to update the voxel map as new observations of the environment are made were verified. When testing the mapper with AirSim's groundtruth depth within the "Building 99" (fig. 7.1) AirSim environment provided by Microsoft, the occupancy map changed appropriately as the characters move through the scene. However, the FUEL planner doesn't have any built-in capability to replan the drone's behaviour based on dynamic objects. One possible extension to this project could rely on combining the semantically labelled voxels with techniques such as optical flow to allow FUEL to make predictions on where dynamic objects are heading, and using that to reduce chances of collision.
- As discussed in the background chapter of this report, modern SLAM systems incorporate a loop closing system that correct previous odometry estimations when loops are detected in the trajectory. One way to improve the mapper's performance could be to publish the loop closer's corrections, and adjust the position of previously projected voxels accordingly.
- Solutions such as SIMVodis [30], which, as mentioned in the background section, pack several computer vision functions into one could be considered as a substitute of the SLAM module, the depth estimator and the image segmentator.

Figure 7.1: Microsoft's Building 99 environment.

Chapter 8

User Guide

This project's repo can be found **here**.

As mentioned previously in this report, one particularly frustrating problem encountered when working in this project has been the lack of good quality repositories available when wanting to try-out other people's work. Many hours have been needlessly spent finding out, for example, which version of the g++/gcc compilers would prevent FUEL from crashing, which CMake flags OpenCV needs to be compiled with in order to work with ORB-SLAM2-CUDA. Providing a repository with instructions for compiling and running this project's source code in a quick and painless manner is a challenge, specially given that some of the components the project relies on lack of such instructions. Unfortunately, there has been no time in the days leading to the submission of this report to produce a bash script that can install all necessary dependencies in a fast and reliable manner in a new computer. I will aim to produce such script after project submission. In the meantime, this User Guide will describe how to run the different demos using the a Docker image currently stored in a SSD drive connected to the gaming laptop in room 905 in the EEE building.

Firstly, in order to run the docker image provided, a computer is needed with an Nvidia graphics card and "Nvidia Docker" support. The tutorial here gives details on how to install Nvidia Docker. The gaming laptop using for this project already has support for this. Secondly, the docker image must be run like this:

```
1  docker run -it \
2  --runtime=nvidia \
3  --privileged \
4  --net=host \
5  --ipc=host \
6  -v /tmp/.X11-unix:/tmp/.X11-unix \
7  -e DISPLAY=$DISPLAY \
8  -v $HOME/.Xauthority:/root/.Xauthority \
9  -e XAUTHORITY=/root/.Xauthority
10 $IMAGENAME
```

A terminal window will pop up. A copy of this project's repository can then be accessed at `root/monocular_exploration`, and the project can be compiled as follows:

```
conda activate TDNet
cd /root/monocular_exploration/third_party/AirSim/ros/
rm -r ./build
rm -r ./devel
export CC=/usr/bin/gcc-8
export CXX=/usr/bin/g++-8
catkin build
source ./devel/setup.bash
cd ../../../
rm -r ./build
rm -r ./devel
export CC=/usr/bin/gcc-7
export CXX=/usr/bin/g++-7
./catkin_make_python3.sh
source ./devel/setup.bash
```

8.1 Running the mapper demos

First, an RTSP source must be made available in the same local network as the computer running the demo. For example, there are android apps that can be used for this purpose. Second, the demos can be run by using the following launch files. The first one launches the "nodelets only" version of the mapper. The second one launches the "nodes only" version of the mapper. Before running the launch command, the line inside the launch files specifying the IP address of the RTSP source must be indicated.

```
roslaunch my_pipeline gstreamer_esdf_nodelets.launch
roslaunch my_pipeline gstreamer_esdf.launch
```

8.2 Running the explorer demos

An AirSim environment of the user's choice must be firs run on either the same or on another computer. The Semantics Obstacles environment used to obtain some of the results in this report can be downloaded **here**. The PEDRA environments can be downloaded here. Once an AirSim environment is running, the IP address of its host must be indicated in the relevant line of one of these launch files, and the roslaunch command can then be run. The first one launches the explorer with semantics off, the second one launches the explorer with semantics from TDNet and the third one launches the explorer with groundtruth semantics from AirSim.

```
roslaunch my_pipeline play_with_airsim.launch
roslaunch my_pipeline play_with_airsim_semantic.launch
roslaunch my_pipeline play_with_airsim_semantic2.launch
```

Bibliography

[1] H. Schulzrinne, A. Rao, and R. Lanphier, "Rfc2326: Real time streaming protocol (rtsp)," USA, 1998. pages 2

[2] M. Bojarski, D. D. Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, and K. Zieba, "End to end learning for self-driving cars," 2016. pages 5

[3] D. S. Chaplot, D. Gandhi, S. Gupta, A. Gupta, and R. Salakhutdinov, "Learning to explore using active neural slam," in *International Conference on Learning Representations (ICLR)*, 2020. pages 6, 11

[4] M. Savva, A. Kadian, O. Maksymets, Y. Zhao, E. Wijmans, B. Jain, J. Straub, J. Liu, V. Koltun, J. Malik, D. Parikh, and D. Batra, "Habitat: A platform for embodied AI research," *CoRR*, vol. abs/1904.01201, 2019. [Online]. Available: http://arxiv.org/abs/1904.01201 pages 6

[5] T. Chen, S. Gupta, and A. Gupta, "Learning exploration policies for navigation," *CoRR*, vol. abs/1903.01959, 2019. [Online]. Available: http://arxiv.org/abs/1903.01959 pages 6

[6] D. Pittol, M. Mantelli, R. Maffei, M. Kolberg, and E. Prestes, "Monocular 3d exploration using lines-of-sight and local maps," *Journal of Intelligent & Robotic Systems*, vol. 100, no. 2, pp. 465–481, Nov 2020. [Online]. Available: https://doi.org/10.1007/s10846-020-01208-x pages 6, 16

[7] M. J. M. M. Mur-Artal, Raúl and J. D. Tardós, "ORB-SLAM: a versatile and accurate monocular SLAM system," *IEEE Transactions on Robotics*, vol. 31, no. 5, pp. 1147–1163, 2015. pages 6, 10, 11

[8] Y. Lin, F. Gao, T. Qin, W. Gao, T. Liu, W. Wu, Z. Yang, and S. Shen, "Autonomous aerial navigation using monocular visual-inertial fusion," *Journal of Field Robotics*, vol. 35, no. 1, pp. 23–51, 2018. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/rob.21732 pages 7, 16, 48

[9] T. Qin, P. Li, and S. Shen, "Vins-mono: A robust and versatile monocular visual-inertial state estimator," *IEEE Transactions on Robotics*, vol. 34, no. 4, pp. 1004–1020, 2018. pages 7, 12, 18, 49

[10] J. Sandino, F. Vanegas Alvarez, F. Maire, P. Caccetta, C. Sanderson, and L. Gonzalez, "Uav framework for autonomous onboard navigation and people/object detection in cluttered indoor environments," *Remote Sensing*, vol. 12, p. 3386, 10 2020. pages 8

[11] D. S. Chaplot, D. Gandhi, A. Gupta, and R. Salakhutdinov, "Object goal navigation using goal-oriented semantic exploration," in *In Neural Information Processing Systems*, 2020. pages 8

[12] L. Freda, "Visual slam, an overview," May 2016. pages 9

[13] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000. pages 9

[14] K. Yousif, A. Bab-Hadiashar, and R. Hoseinnezhad, "An overview to visual odometry and visual slam: Applications to mobile robotics," *Intelligent Industrial Systems*, vol. 1, no. 4, pp. 289–311, Dec 2015. [Online]. Available: https://doi.org/10.1007/s40903-015-0032-7 pages 10

[15] L. Clemente, A. Davison, I. Reid, J. Neira, and J. TardÃ³s, "Mapping large loops with a single hand-held camera," in *Proceedings of Robotics: Science and Systems*, Atlanta, GA, USA, June 2007. pages 10

[16] S. Wang, R. Clark, H. Wen, and N. Trigoni, "Deepvo: Towards end-to-end visual odometry with deep recurrent convolutional neural networks," in *2017 IEEE International Conference on Robotics and Automation (ICRA)*, 2017, pp. 2043–2050. pages 10

[17] C. Yu, Z. Liu, X. Liu, F. Xie, Y. Yang, Q. Wei, and F. Qiao, "DS-SLAM: A semantic visual SLAM towards dynamic environments," *CoRR*, vol. abs/1809.08379, 2018. [Online]. Available: http://arxiv.org/abs/1809.08379 pages 11

[18] P. Guan, Z. Cao, E. Chen, S. Liang, M. Tan, and J. Yu, "A real-time semantic visual slam approach with points and objects," *International Journal of Advanced Robotic Systems*, vol. 17, p. 172988142090544, 02 2020. pages 11

[19] N. Joshi, Y. Sharma, P. Parkhiya, R. Khawad, K. M. Krishna, and B. Bhowmick, "Integrating objects into monocular SLAM: line based category specific models," *CoRR*, vol. abs/1905.04698, 2019. [Online]. Available: http://arxiv.org/abs/1905.04698 pages 11

[20] S. Yang and S. A. Scherer, "Cubeslam: Monocular 3d object detection and SLAM without prior models," *CoRR*, vol. abs/1806.00557, 2018. [Online]. Available: http://arxiv.org/abs/1806.00557 pages 11

[21] L. Nicholson, M. Milford, and N. Sünderhauf, "Quadricslam: Constrained dual quadrics from object detections as landmarks in semantic SLAM," *CoRR*, vol. abs/1804.04011, 2018. [Online]. Available: http://arxiv.org/abs/1804.04011 pages 11

[22] F. Zhong, S. Wang, Z. Zhang, C. Chen, and Y. Wang, "Detect-slam: Making object detection and slam mutually beneficial," in *2018 IEEE Winter Conference on Applications of Computer Vision (WACV)*, 2018, pp. 1001–1010. pages 11

[23] R. F. Salas-moreno, R. A. Newcombe, H. Strasdat, P. H. J. Kelly, and A. J. Davison, "Slam++: Simultaneous localisation and mapping at the level of objects." pages 11

[24] W. N. Greene and N. Roy, "Flame: Fast lightweight mesh estimation using variational smoothing on delaunay graphs," in *2017 IEEE International Conference on Computer Vision (ICCV)*, 2017, pp. 4696–4704. pages 12, 16, 19

[25] A. Rosinol, T. Sattler, M. Pollefeys, and L. Carlone, "Incremental Visual-Inertial 3D Mesh Generation with Structural Regularities," in *IEEE Int. Conf. Robot. Autom. (ICRA)*, 2019. pages 12

[26] R. Garg, V. K. B. G, and I. D. Reid, "Unsupervised CNN for single view depth estimation: Geometry to the rescue," *CoRR*, vol. abs/1603.04992, 2016. [Online]. Available: http://arxiv.org/abs/1603.04992 pages 12

[27] R. Wang, S. M. Pizer, and J. Frahm, "Recurrent neural network for (un-)supervised learning of monocular videovisual odometry and depth," *CoRR*, vol. abs/1904.07087, 2019. [Online]. Available: http://arxiv.org/abs/1904.07087 pages 12

[28] F. Aleotti, F. Tosi, M. Poggi, and S. Mattoccia, "Generative adversarial networks for unsupervised monocular depth prediction," in *Computer Vision – ECCV 2018 Workshops*, L. Leal-Taixé and S. Roth, Eds. Cham: Springer International Publishing, 2019, pp. 337–354. pages 12

[29] K. Wang and S. Shen, "Mvdepthnet: Real-time multiview depth estimation neural network," *CoRR*, vol. abs/1807.08563, 2018. [Online]. Available: http://arxiv.org/abs/1807.08563 pages 13, 48

[30] U. Kim, S. Kim, and J. Kim, "Simvodis: Simultaneous visual odometry, object detection, and instance segmentation," *CoRR*, vol. abs/1911.05939, 2019. [Online]. Available: http://arxiv.org/abs/1911.05939 pages 13, 50

[31] L. Han, F. Gao, B. Zhou, and S. Shen, "FIESTA: fast incremental euclidean distance fields for online motion planning of aerial robots," *CoRR*, vol. abs/1903.02144, 2019. [Online]. Available: http://arxiv.org/abs/1903.02144 pages 13, 16, 19

[32] H. Oleynikova, Z. Taylor, M. Fehr, R. Siegwart, and J. Nieto, "Voxblox: Incremental 3d euclidean signed distance fields for on-board mav planning," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2017. pages 13, 19

[33] B. Yamauchi, "A frontier-based approach for autonomous exploration," in *Proceedings 1997 IEEE International Symposium on Computational Intelligence in Robotics and Automation CIRA'97. 'Towards New Computational Principles for Robotics and Automation'*, 1997, pp. 146–151. pages 14

[34] A. Bircher, M. Kamel, K. Alexis, H. Oleynikova, and R. Siegwart, "Receding horizon "next-best-view" planner for 3d exploration," in *2016 IEEE International Conference on Robotics and Automation (ICRA)*, 2016, pp. 1462–1468. pages 14

[35] B. Zhou, Y. Zhang, X. Chen, and S. Shen, "Fuel: Fast uav exploration using incremental frontier structure and hierarchical planning," 2020. pages 14, 19, 20

[36] T. Dang, F. Mascarich, S. Khattak, C. Papachristos, and K. Alexis, "Graph-based path planning for autonomous robotic exploration in subterranean environments," in *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2019, pp. 3105–3112. pages 14, 20

[37] M. Dharmadhikari, T. Dang, L. Solanka, J. Loje, H. Nguyen, N. Khedekar, and K. Alexis, "Motion primitives-based agile exploration path planning for aerial robotics," in *2020 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2020. pages 14, 20

[38] D. Nilsson and C. Sminchisescu, "Semantic video segmentation by gated recurrent flow propagation," *CoRR*, vol. abs/1612.08871, 2016. [Online]. Available: http://arxiv.org/abs/1612.08871 pages 14

[39] B. Mahasseni, S. Todorovic, and A. Fern, "Budget-aware deep semantic video segmentation," in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017, pp. 2077–2086. pages 14

[40] E. Shelhamer, K. Rakelly, J. Hoffman, and T. Darrell, "Clockwork convnets for video semantic segmentation," 2016. pages 14

[41] D. Teso-Fz-Betoño, E. Zulueta, A. Sánchez-Chica, U. Fernandez-Gamiz, and A. Saenz-Aguirre, "Semantic segmentation to develop an indoor navigation system for an autonomous mobile robot," *Mathematics*, vol. 8, no. 5, p. 855, May 2020. [Online]. Available: http://dx.doi.org/10.3390/math8050855 pages 16

[42] A. Anwar and A. Raychowdhury, "Autonomous Navigation via Deep Reinforcement Learning for Resource Constraint Edge Nodes using Transfer Learning," *arXiv e-prints*, p. arXiv:1910.05547, Oct 2019. pages 16, 35

[43] NVIDIA, "Nvidia video codec sdk documentation." [Online]. Available: https://docs.nvidia.com/video-technologies/video-codec-sdk/nvdec-video-decoder-api-prog-guide/ pages 17

[44] M. Servières, V. Renaudin, A. Dupuis, and N. Antigny, "Visual and visual-inertial slam: State of the art, classification, and experimental benchmarking," *Journal of Sensors*, vol. 2021, p. 2054828, Feb 2021. [Online]. Available: https://doi.org/10.1155/2021/2054828 pages 18

[45] S. Das, "Simultaneous localization and mapping (SLAM) using RTAB-MAP," *CoRR*, vol. abs/1809.02989, 2018. [Online]. Available: http://arxiv.org/abs/1809.02989 pages 19

[46] N. Ratliff, M. Zucker, J. A. D. Bagnell, and S. Srinivasa, "Chomp: Gradient optimization techniques for efficient motion planning," in *Proceedings of (ICRA) International Conference on Robotics and Automation*, May 2009, pp. 489 – 494. pages 19

[47] P. Hu, F. C. Heilbron, O. Wang, Z. Lin, S. Sclaroff, and F. Perazzi, "Temporally distributed networks for fast video semantic segmentation," 2020. pages 20, 21, 22

[48] P. K. Nathan Silberman, Derek Hoiem and R. Fergus, "Indoor segmentation and support inference from rgbd images," in *ECCV*, 2012. pages 21

[49] T. Lee, M. Leok, and N. H. McClamroch, "Geometric tracking control of a quadrotor uav on se(3)," in *49th IEEE Conference on Decision and Control (CDC)*, 2010, pp. 5420–5425. pages 24

[50] R. wiki, "Ros tf package." [Online]. Available: http://wiki.ros.org/tf pages 28

[51] T. Peng, D. Zhang, D. L. Hettiarachchi, and J. Loomis, "An evaluation of embedded gpu systems for visual slam algorithms," vol. 2020, 01 2020, pp. 325–1. pages 29, 40

[52] D. Bourque, "Cuda-accelerated orb-slam for uavs," Thesis, Worcester Polytechnic Institute, June 2017. pages 29, 40

[53] Y. Chen, "Orb-slam2-gpu," 2016. [Online]. Available: http://yunchih.github.io/ORB-SLAM2-GPU2016-final/ pages 29, 40

[54] V. Lamoine, "Graph rviz plugin." [Online]. Available: http://wiki.ros.org/graph_rviz_plugin pages 34