Automatic Generation of a Floor Plan from a 3D Scanned Model

Making the Analogue World Digital

Presented by:
Bradlee Kenneth Wilson

Prepared for:
Dr. Simon Winberg & Prof. Daniel O'Hagan

Radar Remote Sensing Group
Department of Electrical Engineering
University of Cape Town

Submitted to the Department of Electrical Engineering at the University of Cape Town in partial fulfilment of the academic requirements for a Masters of Science degree in Radar Engineering.

October 2018

Declaration

1. I know the meaning of plagiarism and declare that all the work in the document, save for that which is properly acknowledged, is my own.
2. I have used the IEEE convention for citation and referencing. Each contribution to, and quotation in, this report from the work(s) of other people has been attributed, and has been cited and referenced.
3. This thesis/dissertation has been submitted to the Turnitin module (or equivalent similarity and originality checking software).
4. I confirm that my supervisor has seen my report and any concerns revealed by such have been resolved with my supervisor.
5. I have not allowed, and will not allow, anyone to copy my work with the intention of passing it off as their own work or part thereof.

Signature:.. Signed by candidate

B. K. Wilson

Date:..... 19 October 2018

Acknowledgements

There are a large amount of people that I would like to thank for helping me with this dissertation. Everyone in my life, one way or another, contributed towards this dissertation. However, there are those who I would like to make a special effort in acknowledging and thanking.

Firstly, I would like to thank my supervisor Dr. Simon Winberg, without whom I would not be completing this dissertation. Thank you for having the utmost patience with me over the past two years, for always being so understanding, and for all the guidance and insight you have given me throughout this time.

I would also like to thank Prof. Daniel O'Hagan for all that he has done for me, as well as for convincing me two years ago to pursue a Radar orientated Masters; it has made all the difference.

To my parents, Glisson, Lynne, and Shelagh, thank you for all the support, love, and help you have given me towards completing this dissertation.

I would also like to thank my sister Tayla, without whom the past two years would have been even more difficult, thank you for being so awesome!

To my best mate Callen, thank you for ruining my drafts with red pen, answering any and every question I asked you, and being there whenever I needed help. You have played a huge roll in helping me finish, thank you!

Thank you Erin for your continuous love and support in everything I do!

Thank you, Caitlin, Kimberly, Royden, Shane, Jess, Dr. Watson, Etienne, Chad, Raquel, and Rob. You have all encouraged and believed in me, which definitely helped me focus on what has been a hugely rewarding process.

I would like to thank all my other close friends and family. Your continuous support and encouragement on a daily basis helped me in completing this dissertation.

And finally, thank you to my Grandpa for all his love, and for being a large influence in me becoming an engineer.

Abstract

The processing of three-dimensional (3D) room models is an area of research undertaken by many academics and hobbyists due to multiple uses derived from the information obtained - such as the generation of a floor plan; an example of bridging the real and digital world. A floor plan is required when an existing room, floor, or building requires alteration. By having the floor plan in the digital domain it allows the user to alter the room via simulation and render the environment in a life-like manner to determine if the alterations will suffice. This is done using Computer Aided Design Software (CAD).

Designing a new room or building would be done using CAD software. However, not all building's digital files are readily available or exist - making the creation of a floor plan necessary. The floor plan can created up by a person on pen and paper, or with using software tools and sensors. Commercial systems exist for this task but there are no automated, open-source systems that can do the same. Current research tends to focus on the processing algorithms and not the sensors or methods for capturing the environment.

This dissertation deals with testing and evaluating off-the-shelf (OTS) sensors and the processing of 3D modelled rooms captured with one of these sensors. The tests performed on the OTS sensors determine the overall accuracy of the sensors for 3D room modelling. The rationale for designing and conducting these tests is to provide the community with suggested practical tests to assist in selecting an OTS sensor for 3D room modelling. The 3D room models are captured using an open-source application and are imported into custom software. The 3D models undergo pre-processing algorithms producing 2D results, which were further processed to determine the walls of rooms. The dimension information about these features are used to create a 2D floor plan.

3D modelled environments are inherently noisy, requiring efficient pre-processing to remove the noise without hampering processing performance of the 3D model. One of the largest contributors to noise and accuracy is the sensor. Selecting the appropriate sensor can mitigate the need for complex pre-processing algorithms and will improve overall processing time.

The project was able to extract dimension information within an acceptable error. The tests that were designed and used for sensor testing were able to determine which sensor was the better choice for 3D room modelling. The optimal sensor was found to be Microsoft's Kinect[1]. Tests were performed in which the Microsoft Kinect was required to map a room. The results show that dimensional information about the given scene could be successfully extracted with an average error of 4.60 %.

[1]The Kinect is discontinued, however, the underlying technology of the sensor is produced by PrimeSense [1], who have their own range of RGBD sensors.

Contents

List of Figures

List of Tables

Nomenclature

2D	Two Dimensions
3D	Three Dimensions
ADC	Analogue-to-Digital Converter / Conversion
CSV	Comma-Separated Values
DAC	Digital-to-Analogue Converter / Conversion
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HD	High Definition
LiDAR	Light Detection and Ranging
OpenCV	Open Source Computer Vision Library
OTS	Off-The-Shelf
PC	Personal Computer
PCL	Point Cloud Library
PNG	Portable Network Graphics
RGB	Red, Green, Blue
RGB-D	Red, Green, Blue, Depth
ROS	Robot Operating System
RTAB-MAP	Real-Time Appearance-Based Mapping
SLAM	Simultaneous Localization and Mapping
TOF	Time-of-Flight

Chapter 1

Introduction

Bridging the gap between the 'real-world' (i.e. analogue world outside the computer) and the 'digital-world' inside the computer, has been an ongoing research area for many decades. We are continuously finding ways which are faster and more cost effective to enable computers to sense the analogue world. Computers are able to store and process data at a much higher rate than any one person. Tasks which would require real-world data capture, such as the production of floor plans [2] by taking measurements of the physical rooms inside a building, are a manually intensive process. The effort required for a human to measure the rooms, note down the measurement, and draw the sketches of the room, could be done by a computer in a fraction of the time. The task could be done with more accuracy by a computer as it would remove the possibility of human error, caused by the human accidentally misreading the tape measure, or recording the measurements incorrectly.

Commercial products, such as Autodesk's AutoCAD [3], already exist and are able to create floor plans from point clouds, but they are out of the price range of the general enthusiast. This dissertation explores ways in which readily available off-the-shelf (OTS) products can be combined with straightforward algorithms and open-source libraries to create a floor plan modelling system at a fraction of the cost.

Ideally, the task of floor plan creation could be further automated by making use of robotics. Robotic automation is a large area of research at universities around the world, where room and building modelling in 3D is a commonly tackled topic. 3D mapping purposes can range from simple tasks, such as navigation of a room for a robot, as done by A. Oliver *et al.* [4], to more advanced tasks such as room detection from point clouds as done by C. Mura *et al.* [5]. Components to build simple robotic systems are becoming more affordable as they become more involved in day to day activities. This has allowed for hobbyists, not just academics with funding and companies with large R&D departments, to get involved in robotics and add to the open-source community.

There are many open-source libraries and tools available for room modelling and processing of 3D and 2D data. The purpose of this dissertation is to determine the best amalgamation of tools, libraries, algorithms, and self-made methods to create a floor plan from an inputted 3D room model. An additional purpose of this dissertation is determining criteria to compare sensors, as many OTS components can be used for room modelling - such as the Intel Realsense [6], LiDAR scanners [7], or the Microsoft Kinect [8]. It is necessary to create comparison tests because there is little to no viable knowledge on determining the accuracy of these sensors for 3D room modelling.

1.1 Background to the study

Whilst researching papers dealing with 3D room modelling, it was noted that there was limited research available addressing the topic of creating a 2D labelled floor plan from a 3D room model given as an input to the system. The work presented by Ochmann *et al.* proved suitable for detecting features, such as doors in a room, so that optimal navigation paths can be calculated through a building [9]. However, most research found dealt mainly with taking an input point cloud, processing it, and producing the same point cloud with altered data (i.e. room detection). The capturing of the point clouds in most of these papers were not covered, and the accuracy of the sensors were not explored.

The goal of this dissertation is to perform room detection and flatten it to produce a 2D image with dimension labelling, i.e. a floor plan. However, creating a floor plan is only a part of the problem at hand, with the main objective being to determine the best way to perform room and wall detection. The initial design approach to the problem was similarly tackled by Oesau *et al.* [10], which is discussed further in Chapter 4. For both Ochmann and Oesau's research, the point clouds had to be captured using sensors.

Many types of sensors exist for modelling the real world, most of which are used for video-games. Sensors in the video-gaming industry allow for greater interaction with the game by using body movements instead of a hand-held controller. Early-generation sensors, like the PlayStation EyeToy [11], was an example of such a sensor. It was simply just a camera where motion detection was performed on its video stream to capture player movement in a two-dimensional plane. Due to advancements in technology, these sensors are now able to gauge the distance between sensor and player - expanding the interaction to that of three dimensions. This added dimension allowed the interactiveness of games to become more natural and popular, resulting in a boom in the console gaming industry.

This boom caused these advanced sensors to become affordable to the general public and their low cost, when compared to their commercial counterparts, have made them an attractive purchase - widely used by academics and hobbyists alike. They gained popularity as well due to the companies which created

these sensors making the software development kits easily accessible. By improving the algorithms and techniques employed by these sensors, they can be enhanced to carry out the same tasks as the commercial systems, with reasonable accuracy. Although algorithms can improve the accuracy of the sensor, choosing the optimal sensor for the task will improve the accuracy of the system without sacrificing processing power or coding time.

1.2 Requirements and Functions

This project has two main objectives, primarily, the generation of a floor plan and its dimensions from a 3D scan, and secondarily, determining the optimum sensor for open-source 3D modelling. Although the determination of the optimum sensor was done first, it was not the main focus of the project, but was necessary in determining which sensor to use. The requirements give an overview of the end result that needs to be achieved and the functionality describes what must be done to achieve these requirements. A brief explanation of the requirements and their functions are given below, with the tests for these outlined in Chapter 3.

1.2.1 Requirements

The system designed in this project must be able to import a 3D modelled point cloud of a room and perform processing operations on it to output a floor plan using open-source software and freely available tools. One of the largest challenges will be determining from a point cloud, which is essentially a collection of points, what is a wall and what is not. Subsequently, the lengths of the walls will need to be found and displayed. The processing of the point cloud, that results in the output of a floor plan, should be completely automated.

As stated previously, the processing operations can be eased by choosing an optimum sensor for the task at hand. It should, therefore, be determined which tests can be created to decide if a sensor is adequate for 3D room modelling. These tests should then be used on the sensors available for this project to determine the best one to use.

1.2.2 Functionality

Functionality describes the more precise functions that the system should be able to do, relating back to the requirements listed above. The processing operations should remove unnecessary data from the

point cloud, such as unwanted noise, unnecessary points, and horizontal surfaces. Noise is data that is not beneficial to the data-set and may introduce processing errors or cause a reduction in the accuracy of the processing. Unnecessary points are any points that, after removal, make no difference to the information retained in the 3D model. Therefore, unnecessary points may be considered noise as well.

It is assumed that the input point cloud uses a coordinate system whereby the z-axis is in the direction of the floor and ceiling. This means the x-y plane will span the length and width of the room. The x-y plane is considered the horizontal plane, and a horizontal surface is any surface on this plane. Horizontal surfaces are removed as they are unnecessary in creating a floor plan, which deals only with vertical surfaces (walls). Following this, it is necessary to determine potential walls and smooth them in order to find their lengths.

Other functionality for this project deals with the testing of the sensors on four main criteria. These tests must provide results with respect to sensor noise, accuracy, distance resolution, and overall 3D modelling capabilities. Overall 3D modelling capabilities are included because a sensor might appear accurate on paper but the overall information it returns may not perform well in practice. The results from these tests must be used to make an informed decision on which sensor, used in this project, is best for creating floor plans of a room.

1.3 Problem Description

The task in question is the creation of a floor plan using an OTS 3D sensor (section 2.4), with a PC which was not designed for the task, such as a low-end consumer computer. The reason behind this is that if an individual doing home renovations wanted to create a floor plan they would not purchase a commercial piece of software to do so; it would be easier to manually measure the entire room. However, if the person has a depth sensor from their gaming console at their disposal, it could be convenient to plug it into their computer and run a simple piece of software that will output a floor plan of their house. This project aims to provide this solution.

Due to the variety of readily available OTS sensors, it would be convenient to determine which of the sensors available to the individual is the optimal choice for the task of 3D room modelling. This could be the case if an individual has multiple types of sensors available to them. The reason for testing which sensor is the best is to help to increase the initial accuracy of the 3D model by decreasing noise and making sure the results from the sensor are accurate.

Therefore, the choice of the optimal sensor can mitigate complicated noise reduction techniques and distance accuracy corrections by simply preventing the noise from getting into the system in the

first place. This will also decrease the processing power required for the task. Tests should exist to compare OTS sensors against one another for 3D room modelling. Tests for this are created within this dissertation and are used to choose an OTS for this project.

1.4 Scope and Limitations

This dissertation covers the processing of 3D point cloud data in order to determine the dimensions of the room(s) within it, and output a floor plan of the modelled environment. Acquisition of the point cloud data will be touched on but not explored fully, as a number of open-source methods exist to perform this task [12]. The dissertation also deals with the formulation of tests with which to compare different sensors based on criteria such as noise and distance accuracy. The test results are used to determine which sensor is best for a specific project. The main scope of this dissertation deals with:

- Formulation of sensor tests.
- Using the formulated tests to choose a sensor for the project.
- Processing of 3D room models.
- Processing of 2D images.
- Creation of a floor plan using simple image processing techniques.
- Creation of a floor plan of a single room.

There are a number of limitations on this project, as it focuses on proof of concept, as opposed to exhaustive testing of different sensors and situations. The project made use of only two available sensors, and as such did not perform tests on any other types. These sensors were used as they are the two most commonly used sensors in this area of academic research. No commercial systems were tested for comparison against results drawn using OTS sensors. This limits the conclusions that could be drawn about the efficiency of OTS sensors in comparison to commercial systems.

The dissertation aims to create a proof of concept in order for the approaches within to be applied in a multitude of different ways. However, the project did not cover the measurement of multiple rooms, or rooms of different heights, such as a lecture theatre. The room modelling algorithm in the project follows a logical systematic approach, which will allow for any scenes that the current algorithm cannot process to be added. These other cases can be tackled separately and easily integrated into the system.

The tests developed to compare OTS cameras can not be compared to other tests to check their efficiency in drawing accurate conclusions. This is due to the fact that there are no hobbyist type tests for 3D sensor choice readily available for use. These tests cover broad aspects of the sensor while also extracting intricate details about them, and they will be able to be used for existing sensors as well as any new sensors coming out on the market. Limitations for the project can be summarised below:

- Only two cameras to compare.
- Not having access to commercial products.
- The system is not designed for multiple rooms.
- The system cannot handle rooms of varying height.
- The lack of other tests to compare against as there exists no hobbyist type tests for 3D sensor choice.

1.5 Document Outline

The contents of this dissertation follow the following layout:

Chapter 1 The project introduction is described here, and gives an outline of what is going to be discussed in the dissertation and why it is important.

Chapter 2 Chapter 2 provides the literature review, and introduces methods for processing 3D models and extracting useful information from these models. It then goes into an explanation of 2D image processing techniques using OpenCV.

Following this, the technology behind capturing the scene is discussed, under SLAM. The different ways of capturing depth with a sensor, visually or using light, are also covered in this section.

An explanation of any libraries, software, or algorithms necessary for the completion of the project are included in this chapter. Overall, this chapter will cover all information required to understand any subsequent chapters in the paper.

Chapter 3 This chapter explains the methodology followed to create the system for this dissertation. The system design life-cycle is explained in this chapter for both the floor plan creation software and the sensor testing software and sensor tests.

The manner in which the outcomes of the project will be tested and compared to the objectives and requirements of the dissertation are also covered here.

Chapter 4 The focus of this chapter is the system design, for both the floor plan creation software as well as the sensor testing.

The chapter will deal with the different sub-systems that will work together to eventually produce the finalised floor plan from an inputted 3D model. The design of the tests used to determine which sensor is best is included within this chapter.

Chapter 5 Chapter 5 presents the results of the system whilst discussing the results to ensure that reasons are given as to why things have happened, and to explain unexpected results.

The results are summarised in this chapter to aid in concluding if the systems passed the tests which are outlined in Chapter 3.

Chapter 6 The final chapter contains conclusions and recommendations for the project. Whether the system passed the tests in Chapter 3 are concluded in this chapter with the use of a test matrix and the results summarised in Chapter 5.

Recommendations and ideas for future work is discussed within this chapter.

Chapter 2

Literature Review

This literature review covers the main information required for the understanding and completion of this research. The project makes use of multiple pieces of software and hardware to meet its requirements. The focus of this dissertation is on floor plan creation and off-the-shelf sensor testing for 3D room modelling. The project touches on scanning a room for the purpose of creating a 3D model, but this is only done in order to collect data for the sensor tests as well as for input data into the floor plan creation algorithm. Open-source libraries and tools are used wherever possible. Many different libraries and pieces of existing code and systems were looked at, but were not necessarily included in this literature review. There is additional information within this literature review, which although not directly used within this project, gives the reader a better understanding of the topic being dealt with to facilitate in understanding arguments made later in the paper.

The literature review begins with covering the Point Cloud Library which is used for 3D point cloud processing, and introduces useful features from the library that are used in this dissertation, or aid in explanation of other aspects which are explored. For 2D processing OpenCV is used, from which certain functions are made use of and explained. A library which makes use of both of these libraries is the Robot Operating System, which is covered under SLAM in this literature review. SLAM is used by the Robot Operating System for room exploration and room modelling, often using a depth sensor to create 3D room models. The types of depth sensors relative to this dissertation, and how they work, are covered under the Depth Capturing section in this chapter. All sensors covered in this section are able to return depth information which is capable of creating a point cloud. An outline of this can be seen in Figure 2.1.

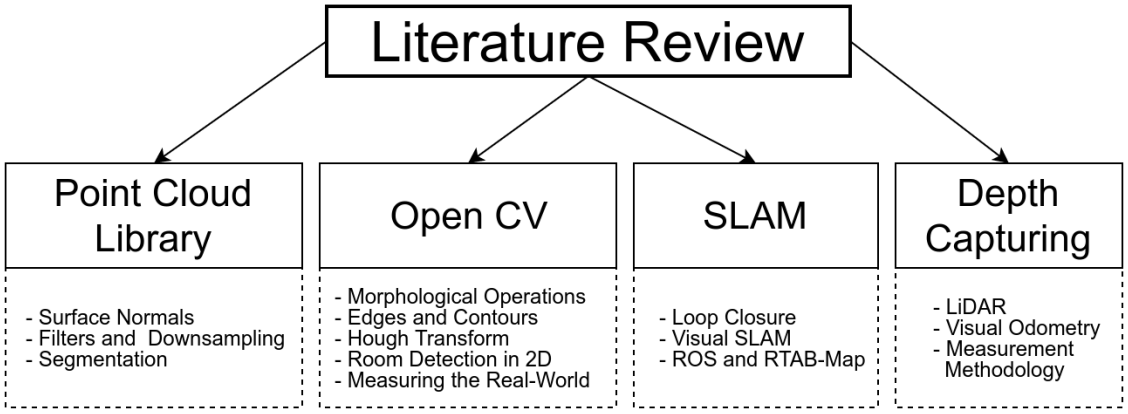


Figure 2.1: An outline of what the literature review is going to cover.

2.1 Point Cloud Library

A point cloud, simply put, is a set of data points with an x, y, and z location within a coordinate system. They are used for a multitude of applications, an example being a graph plotted using three dimensions. Each point on the graph has an x, y, and z value associated to it, where z could be a function of x and y - $f(x, y)$. Although it isn't necessarily called a point cloud it is a simple representation of what a point cloud is that most are familiar with.

A point cloud that is used for room modelling, or object modelling, follows the same structure, it consists of an x, y, and z value for its location. A number representing the red, green, and blue intensity (RGB) for a point may be present as well, this helps with the visual reconstruction of the room.

The generation of point clouds can be from images, LiDAR sensors, as well as RGB-D sensors (touched on later). The point clouds are generated using specialised Simultaneous Localisation and Mapping (SLAM) algorithms with the input being from those previously listed. An example of a point cloud can be seen in Figure 2.2

The Point Cloud Library (PCL) is a collection of tools designed to process point clouds [13]. As mentioned earlier, commercial software exists that contains features to create floor plans. These features can be replicated using tools from the PCL. There are many tools within the PCL, however, the main ones used in this dissertation deal with the extraction of points based on user inputted criteria. Most tools within the library make use of surface normals calculated from the cloud.

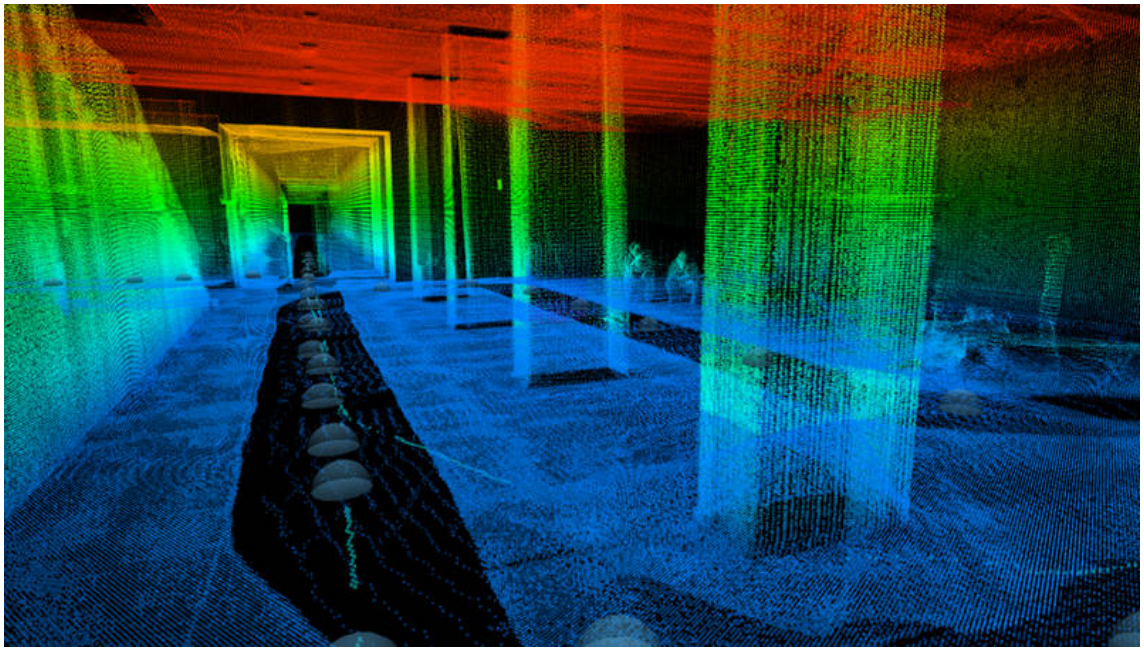


Figure 2.2: A point cloud of a room, with colour representing height from the ground.[1].

2.1.1 Surface Normals with Point Clouds

A surface normal, or normal vector, is a vector which is perpendicular to a surface [14], shown in Figure 2.3. By using surface normals, algorithms can differentiate between points belonging to different objects. This is helpful in determining things such as walls (as the surface normals for a wall will point inwards/outwards) or the roof (which will be surface normals pointing up/down at the highest location in the cloud).

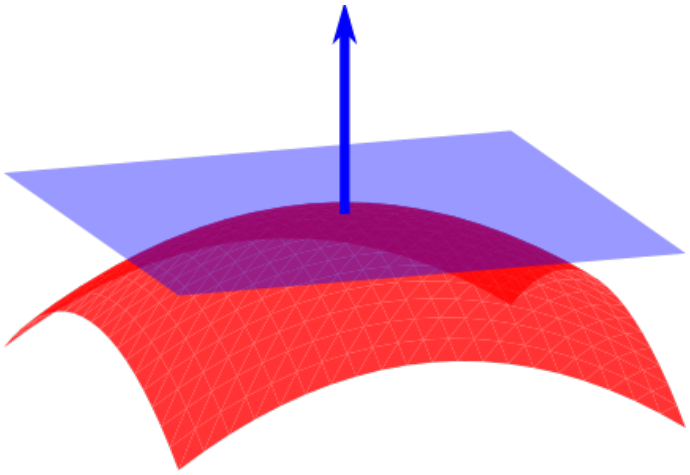


Figure 2.3: An illustration of a surface normal on a surface (red) and the plane (blue) it represents[2].

[1]https://www.tum.de/typo3temp/_processed_/csm_Vorschau_Navvis_8b2d7fcdc7.jpg
[2]https://en.wikipedia.org/wiki/Normal_(geometry)#/media/File:Surface_normal_illustration.svg

Surface normals don't have to all face the same direction for points to be categorised on the same surface. As the surface normal vector in Figure 2.3 moves along the surface, it will continue to be perpendicular to the surface at every point. Using the notion that a surface must have the same surface normal throughout will cause the surface in Figure 2.3 to not be classified as a surface. Allowing for a certain amount of deviation from one surface normal to the next, to account for rounded surfaces, almost any object's surface can be detected.

However, there is one flaw with this; an increase in allowed deviation will cause more error, leading to the algorithm believing points belong to the same surface when they do not. Many algorithms exist for feature and object extraction from a point cloud, most of which are aided by using the surface normals of the point cloud to improve the accuracy of the results. If the surface normals are not accurate then the extracted information may not be accurate either. To accurately calculate the surface normals for an object or a surface, the object or surface would need to be known, however, the surface normals are needed to determine what is the surface or object.

There are algorithms available to overcome this *catch-22* situation. These algorithms find surface normals by comparing the location of a point in question to its surrounding points (neighbours). All that is required from the user is the choice of the search radius, this can be inputted as a physical radius in millimetres or as the amount of nearest neighbours to compare. The effects of choosing an incorrect radius, or number of nearest neighbours, can be seen in Figure 2.4.

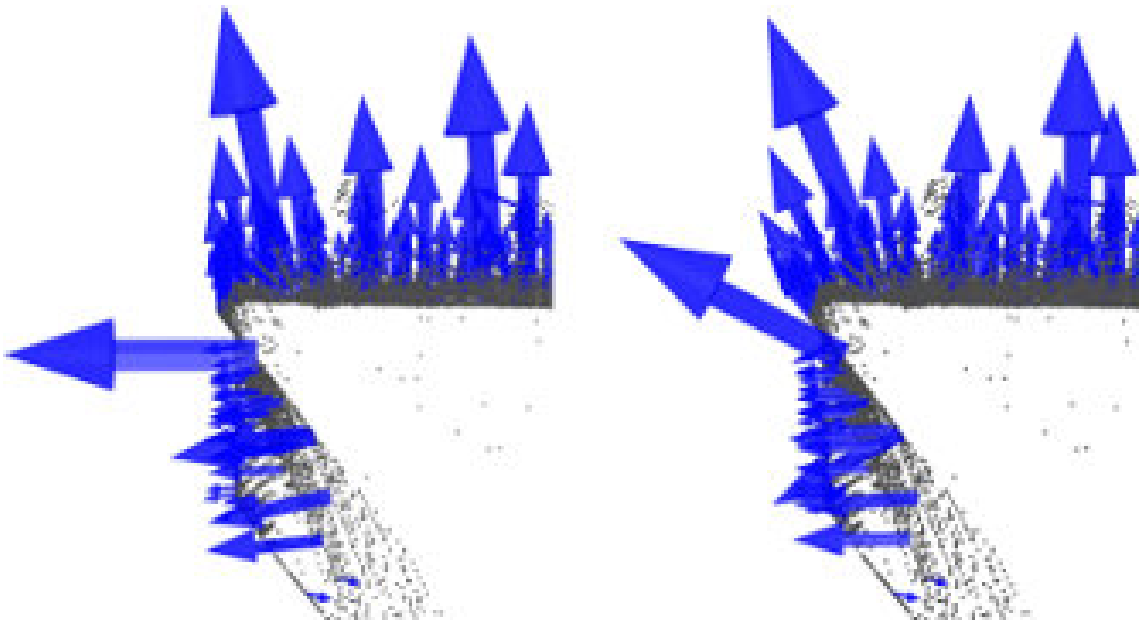


Figure 2.4: A comparison of a well chosen radius (left) and an incorrectly chosen radius (right) for surface normal estimation[3].

A series of trial and error calculations would have to be performed for varying datasets in order to determine the best surface normal estimation thresholds. In Figure 2.4, the surface normals on the

[3]http://pointclouds.org/documentation/tutorials/_images/normals_different_radii.jpg

left were calculated with a smaller radius than those on the right. If the object in question has small, intricate details, a surface normal estimation with a small enough radius to capture the details will need to be chosen. If the object is a large, relatively flat surface (i.e. a wall), increasing the search radius, thereby increasing the nearest-neighbours that will be compared, will generate more accurate surface normals, especially if the input point cloud is noisy.

Surface normal calculation is a computationally intensive process for large point clouds; especially if the input cloud has noisy data, which will waste processing power and add error to the results. To overcome this, filtering and downsampling are performed on the point clouds. Libraries which perform these tasks exist within the PCL.

2.1.2 Filters and Downsampling

Due to the noise of the system as a whole there will be points that are present in the cloud that do not represent anything meaningful. These points cause errors in calculations performed on the point cloud and slow down the processing time as more points need to be processed. A method available in the PCL for removing noisy points (outliers) is the Statistical Outlier Removal algorithm. This algorithm makes use of the nearest neighbours of points to determine if the point does not belong [15].

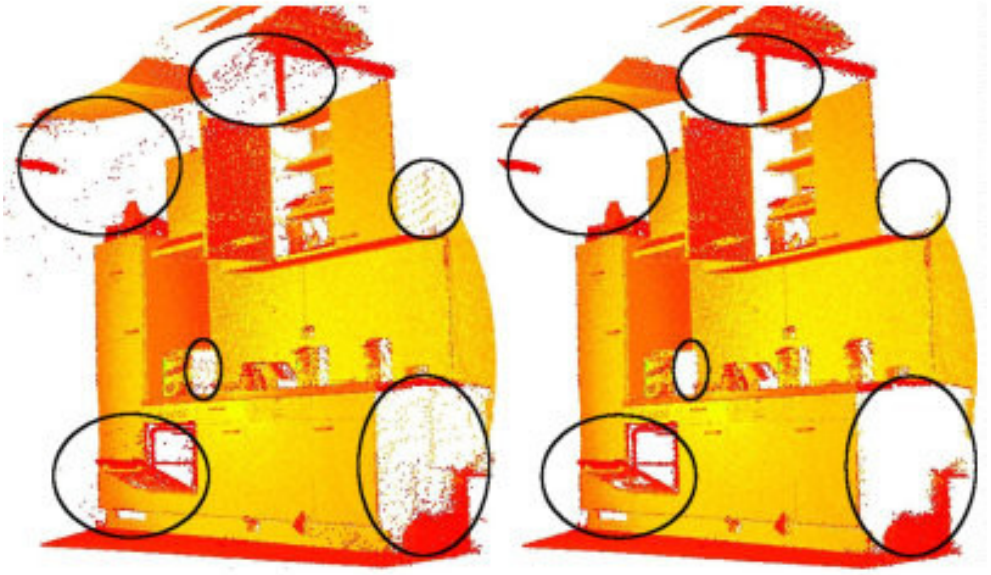
The Statistical Outlier Removal algorithm starts by taking a point and calculating the mean euclidean distance from the point and its K nearest neighbours, where K is a variable defined by the user. If the point's mean distance and standard deviation from its neighbours are greater than an interval defined by the global distances mean and standard deviation, the point is then removed from the dataset.

Figure 2.5(a) shows the removal of outliers from a point cloud and Figure 2.5(b) shows the mean K-nearest neighbours distances of the point cloud before and after outlier removal. The purpose of showing the mean K-nearest neighbours is to illustrate how much noise the algorithm cleared, as it is not always obvious comparing the clouds by eye.

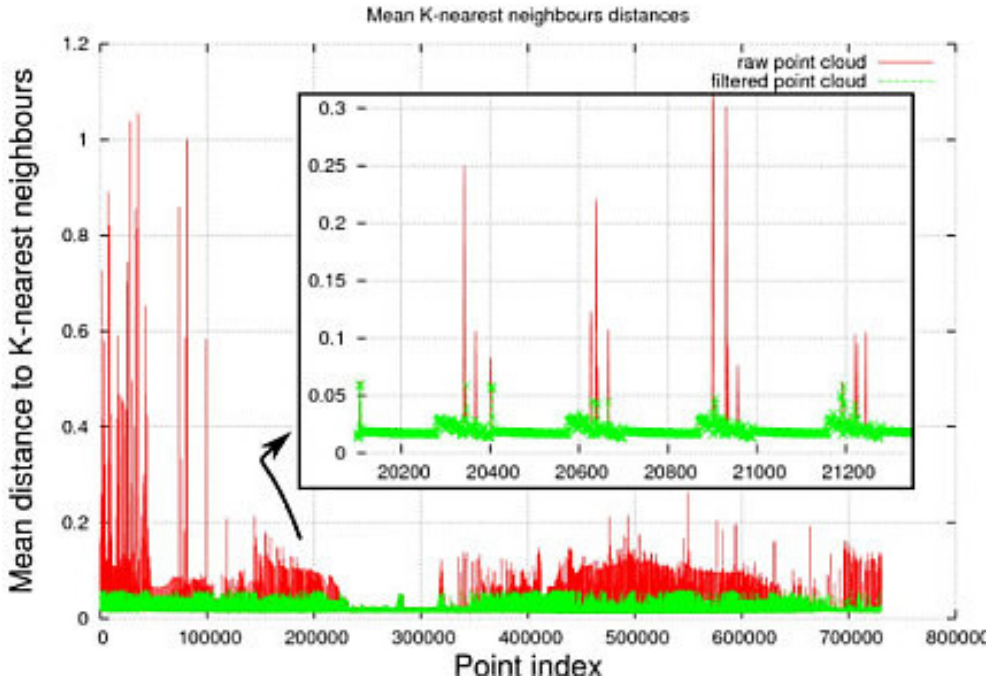
Figure 2.5(b) shows the mean K-nearest neighbours distance, on the y axis, for every point in the cloud, on the x axis. The red lines represent the mean distances for the raw input cloud, and the green lines represent the mean distances for the filtered cloud. A point whose nearest neighbours are very far, increasing the mean distance to its neighbours above the user inputted threshold, will be considered an outlier and removed.

This procedure is resource intensive on the computer and the processing time is relative to the amount

[4] http://pointclouds.org/documentation/tutorials/_images/statistical_removal_2.jpg



(a) Comparison of a point cloud before and after outlier removal.



(b) Graphical representation of how many outliers are removed.

Figure 2.5: An illustration of how statistical noise removal works (a) and a graphical representation of it (b)[4].

of points in the point cloud. To speed up the filtering process the points can be downsampled using another type of filter called a Voxel Grid filter [16].

This filter works by progressively moving through the point cloud using a 3D box in space called a Voxel Grid. This Voxel Grid approximates the centroid of the points within the 3D box and uses this to create a singular point representing these points, thus downsampling. This gives fewer points to process whilst still retaining enough important information for algorithms to still work on the point cloud. The averaging of points will inherently lose accuracy information of what each individual point represented, but this isn't a concern when dealing with planes.

Taking a vertical wall (or a plane parallel to the z axis) in 3D and projecting it onto the $x - y$ plane will result in a line. Averaging any two or more positions on a line will produce another point on the same line. This is the only necessary information for a floor plan and the z value does not come into play, therefore averaging it will have little to no effect on the system. This means that even a large Voxel Grid will not skew the data for floor plan creation. Due to the inherent size of the possible point clouds which will be imported, downsampling without losing important information is crucial and will speed up segmentation.

2.1.3 Segmentation

Segmentation in the context of point clouds refers to the division of points into clusters where each cluster is considered a surface [17]. The method explored here is called Region Growing and follows a relatively simple approach. The explanation is a summary of what is found in the PCL tutorials [17].

Points within the point cloud are sorted by their curvature value. The curvature value represents the size of the change in direction from one normal vector to the next, i.e. how sharply the surface is changing [18]. Therefore, a flat surface will have a smaller curvature value than a rounded surface. By beginning from the point with the minimum curvature it will decrease the amount of inaccurate cluster segments.

Cluster segments are clusters of points that are considered to belong to the same surface. If the algorithm was to start at a random point, assuming it is an outlier on the edge of a surface, this random point's nearest neighbours may meet the criteria for the region growing algorithm. These neighbours will then be removed from the point cloud and added to this points cluster and will not appear in the correct cluster. Starting with the flattest area, it is more likely that the algorithm will be starting on a surface, therefore decreasing the likeliness of points ending up in incorrect, discarded, clusters.

For each region that is grown there consists a set called **seeds**, from which the comparisons will begin,

and another set called **current region**, that contains the points which are considered to be on the same surface as the current seed points. The algorithm will continue looping through the set of seeds until there are none left which indicates a region has been found.

The region growing algorithm [17] calculates the curvature of the points within the cloud from the surface normals, and the minimum curvature is taken as the first seed point. When processing each point, its nearest neighbours are compared to itself. The user defines how many of the nearest neighbours, K, will be compared to the point. Each neighbour is tested for two things:

1. The difference in the angle between the current neighbour's surface normal and the surface normal of the current seed point.
2. The difference in curvature between the current neighbour and the current seed point.

Thresholds are set for both the difference in angle and curvature between the points. If the difference is less than the threshold for the angle the current point is added to the **current region**. If the difference in the curvature is less than the threshold the current point is added to the list of **seeds**. Once the seed has been processed it is removed from the list of seeds. Once the region has been found it is stored separately and the points within the region are removed from the input point cloud, reducing the number of points to be processed. The algorithm repeats until there are no points left.

After processing there will be a collection of regions which can then be analysed separately, or subtracted from the input point cloud if deemed unnecessary - thereby increasing processing performance of the resultant information. For example, by selecting all regions which are parallel to the floor and subtracting them from the input cloud it will remove the floor, roof, tables, etc. This will result in mainly the walls as well as other regions which are not parallel to the floor being left behind.

Another approach supplied by the PCL is the extraction of planar components [19]. Although this sounds similar to the extraction of surfaces, the extraction of a planar component will output a series of points considered to be on the same plane. To determine this the algorithm simply takes in:

1. The surface normal vector to which the planes it is searching for must be perpendicular to.
2. The allowable offset between the surface normal it is searching for and the surface normal of the points it is examining.
3. The distance a point must be from another point to be considered on the same plane.

There is therefore no need to calculate the curvature and grow the region outwards from a seed point by using this method. The horizontal planes will also not need to be removed as they wouldn't have been included in the output, nor would any points which are not on the plane orientation being searched for. This approach will perform a lot quicker than the region growing approach, with the only draw back being that if the surface is curved, or takes on a non-flat shape, the algorithm will struggle as it is focused on planar extraction. By performing region growing or planar extraction many points will be removed which will improve overall processing performance down the line. The points removed, if outliers, could have hampered processing accuracy on the point cloud.

Another example of removing points from a point cloud in order to find cluster segments is the difference of normals (DoN) segmentation tutorial from the PCL. It is a prime example of how surface normals along with other parameters can allow us to extract useful data from a scene [20]. The surface normals for a scene are calculated twice, using a small radius and a large radius. The purpose for this is if a surface is relatively flat, the surface normal that is calculated for a small radius and a large radius should be the same. The only time they will differ a lot is if the surface is not smooth.

By subtracting these normals, as shown in Figure 2.6, you can see that if the vectors were the same the resultant vector would be zero.

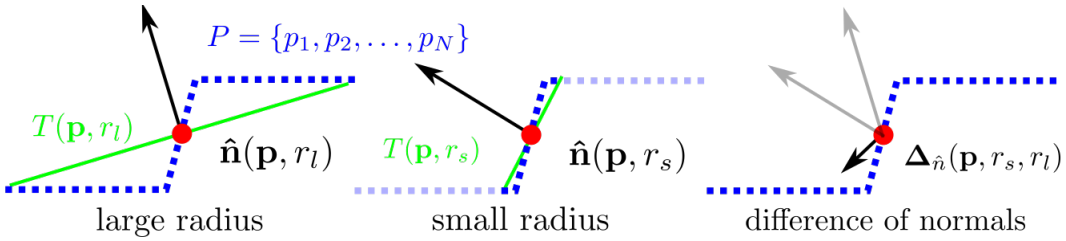


Figure 2.6: Illustration of the difference between a normal with a large and small radius.[5].

After this subtraction takes place additional filtering can be done to refine the areas of interest, and clustering can take place to separate the remaining points into their own segments. An illustration of how this works can be seen in Figure 2.7. It is obvious from this example that the algorithm can be useful for finding the edges of rooms and windows or doors. Once features have been found, they can be flattened from 3D to 2D and processed in 2D using other libraries such as OpenCV.

2.2 OpenCV

The Open Source Computer Vision Library (OpenCV) is an open source computer vision and machine learning software library [21]. Its purpose is the manipulation and processing of images and videos. It

[5]http://pointclouds.org/documentation/tutorials/_images/don_scalenormals.svg

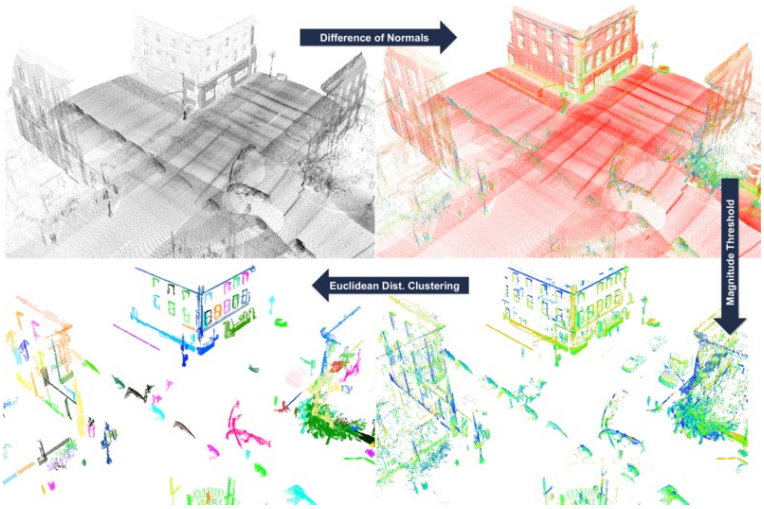


Figure 2.7: Overview of the pipeline in Difference of Normals Segmentation [20].

can track objects, produce point clouds from stereo cameras, as well as establish markers for augmented reality. The main difference between OpenCV and the PCL is that the PCL processes in 3D (using an entire point cloud for processing) whereas OpenCV specialises in processing 2D images.

2.2.1 Morphological Operations

OpenCV contains many functions that are considered "building blocks". Building blocks in this context refer to functions which are used to teach simple image processing techniques in OpenCV. Although they are simple algorithms they produce useful results. These simple functions are called "Morphological Operations" [22] and are called so because they morph *(i.e. change the image)* when they are used. The two morphological operations used in this project are the erosion and dilation of pixels in an image.

These two morphological operations follow the same procedure, but slightly differ from each other. For both dilation and erosion a kernel, which can be thought of as an x by y grid, must be defined. A kernel can have any shape or size and has a fixed anchor point, normally the centre of the kernel. This kernel is then convolved across the image pixel by pixel.

What this means is each pixel in the image is examined one by one. The anchor point of the kernel is placed on the current pixel being processed. The pixel being processed will take on the maximum value present in the kernel if it is being dilated, or the minimum value if it is being eroded. An example of this can be seen in Figure 2.8.

The dilation function in Figure 2.8(b) will make the letter J thinner. This is because the colour white is a larger value (255 if 8 bit) than the colour black (0, regardless of data type). Dilation takes the

(a) Input Image. **(b)** Output of Dilation **(c)** Output of Erosion

Figure 2.8: An example of what the two morphology operations dilation (b) and erosion (c) are capable of [22].

largest value in the kernel, which will result in the brighter pixels overwriting the dark images.

Erosion in Figure 2.8(c) is eroding away the brightest pixels, as it takes on the lowest value in the kernel. Erosion and dilation, if performed one after the other can then cancel each other out. However, if used correctly they can be used to join points together to help smooth data. This smoothed data can then be taken and manipulated with other OpenCV functions, such as finding the edges of the lines within the image.

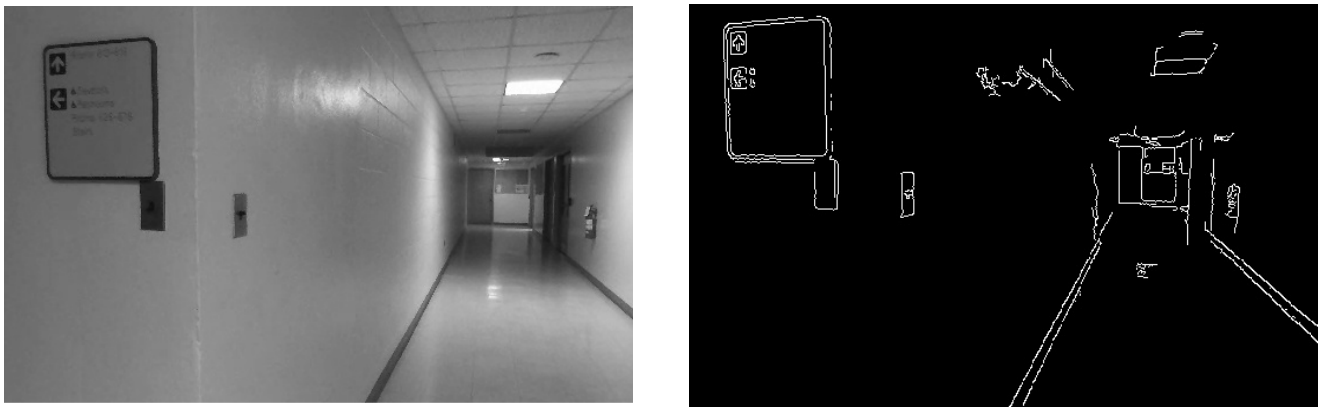
2.2.2 Edges and Contours

We determine the edge of an object within an image, be it a table or person, by the differences in colour and contrast our eyes see within the image. Similarly, we can find the edges of a room following the same principles. An edge is identified as points at the extreme of an image gradient, in the direction of the gradient; therefore, an edge doesn't necessarily represent an object [23]. An example of this is if there are two contrasting colours painted on a wall, by the explanation above, the edge between these colours will be found, but this edge is not representative of an object.

Although edges aren't a definitive method for detecting objects they can help in narrowing down where to search. The outline of an object can be seen as a closed-loop around an object. If edge points belong on the same closed-loop, they are considered to be on the same contour. OpenCV provides a function called 'findContours' which takes in a greyscale image with edge points [24] and produces the contours from the image.

In order to find the required edge points for the 'findContours' function, OpenCV's main edge detector, the Canny Edge Detector, is used. This edge detector is widely used in lots of computer vision projects

and was developed by John Canny in 1986 [25]. The Canny Edge detector on its own will produce all edges it can find in an image, as demonstrated in Figure 2.9.



(a) Input image for Canny Edge Detection[6]

(b) Output of Canny Edge Detector[6]

Figure 2.9: An example of what the Canny Edge Detector will produce from an image [26].

The outputted edges will then be inputted into the Contour Detector that is included in OpenCV. The contour detector, as demonstrated in Figure 2.10, will then output an array of contours. Each contour can be manually selected and processed accordingly, whereas in Figure 2.9(b) the output is simply just pixels in the location of edges, and do not contain any additional information relating them to each other. Another method to extract shapes from an image is using the Hough Transform.

Figure 2.10: A visual representation of the contours found from Figure 2.9(b), where each colour represents a different contour.

[6]https://sites.google.com/site/cnvvision/_/rsrc/1468848439800/home/corridor_canny.png

2.2.3 Hough Transform

The Hough Transform is present as a tool within the OpenCV library and is a technique used to isolate the features of a particular shape (line, circle, ellipsis, etc.) within an image [27]. The Hough Transform requires the input of edge description points, like in Figure 2.9(b), which are commonly obtained from feature detectors or edge detectors and generally contain noise. The Hough Transform examines each edge point and transforms it into the Hough parameter space. Given the input edge points, line segments can be found to join the points as shown in Figure 2.11.

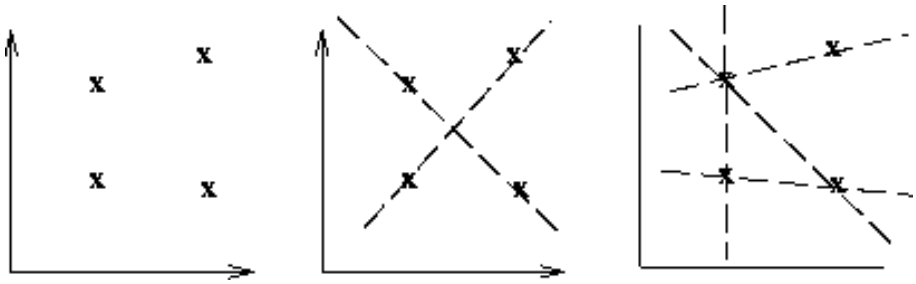


Figure 2.11: Input edge points and the line segments joining the points[28].

A line can be described using a parametric description:

$$x \cos\theta + y \sin\theta = r \tag{2.1}$$

Therefore, taking a point (x_i, y_i) in the space, where i is the index of the point being analysed, every possible line passing through it will have a corresponding $(r_{(i,j)}, \theta_{(i,j)})$, where j represents every Hough parameter pair for every Cartesian index i. θ will range from $-\pi$ to π. This is shown in Figure 2.12.

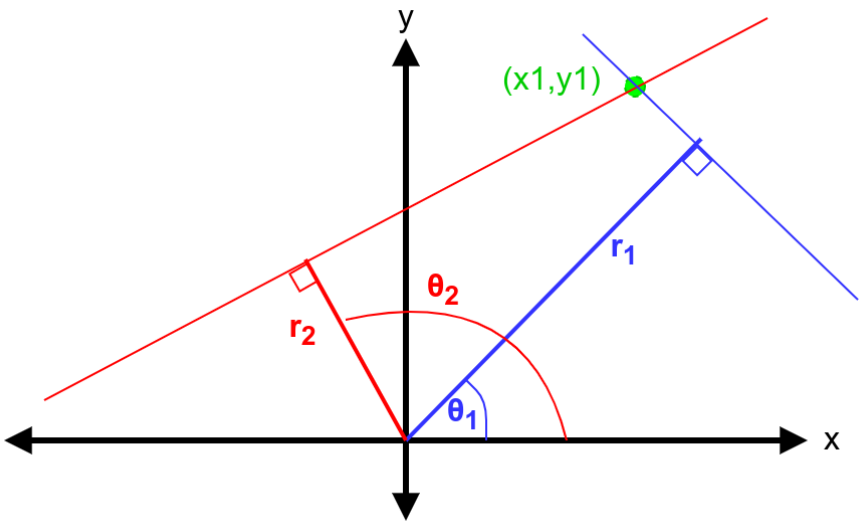
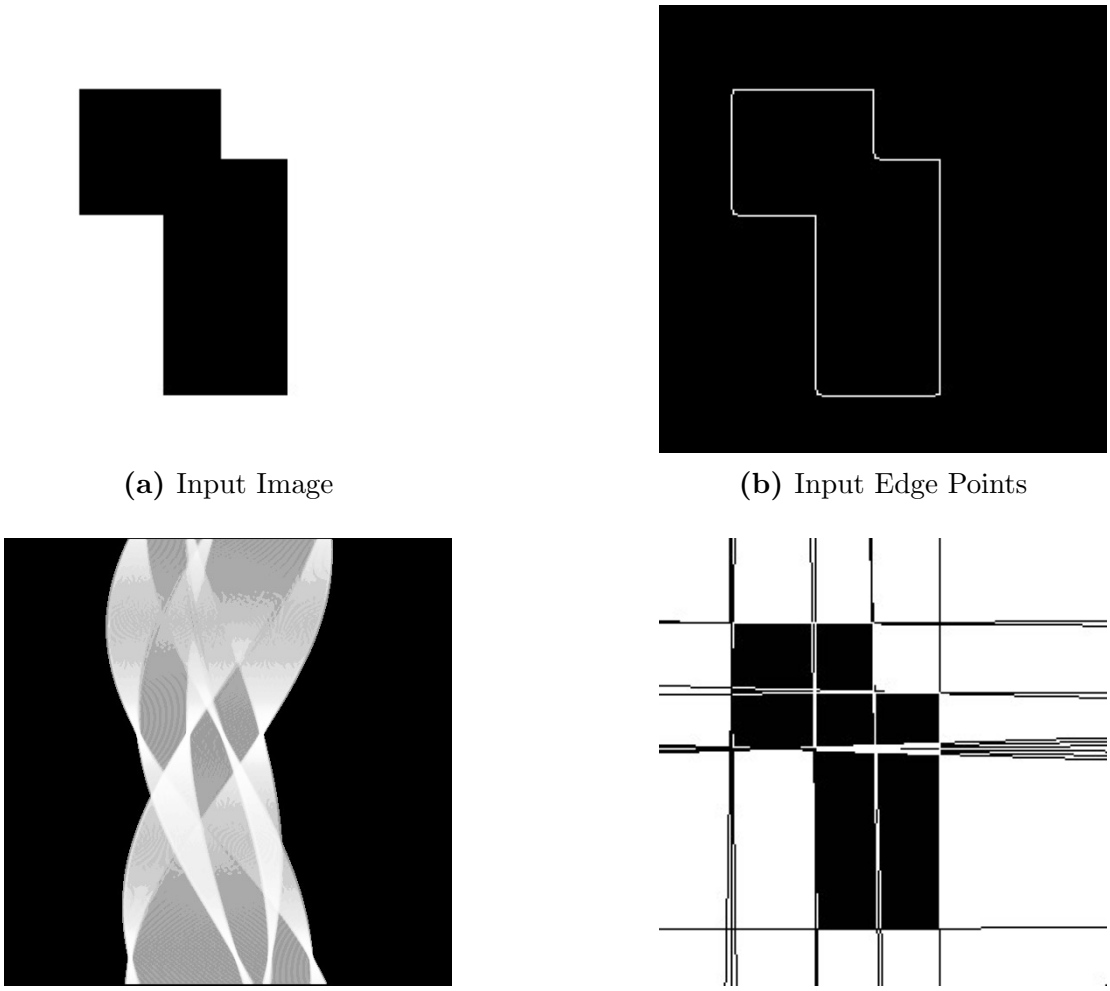


Figure 2.12: Parametric visualisation of a straight line [7]

By changing our domain from the Cartesian domain, (x, y), to the Hough parameter space, (r, θ), making (x, y) the independent variables, all possible values of $(r_{(i,j)}, \theta_{(i,j)})$ for every (x_i, y_i) can be found. The output of this can be seen in Figure 2.13(c). Points which are collinear in the Cartesian image space yield curves which intersect at common (r, θ) points. Figure 2.13(c) demonstrates that any intersection points characterise straight line segments in the original image, producing the image in Figure 2.13(d).



(a) Input Image

(b) Input Edge Points

(c) Hough Parametric Space

(d) Hough Line Output

Figure 2.13: The steps involved for detecting lines in an image[28].

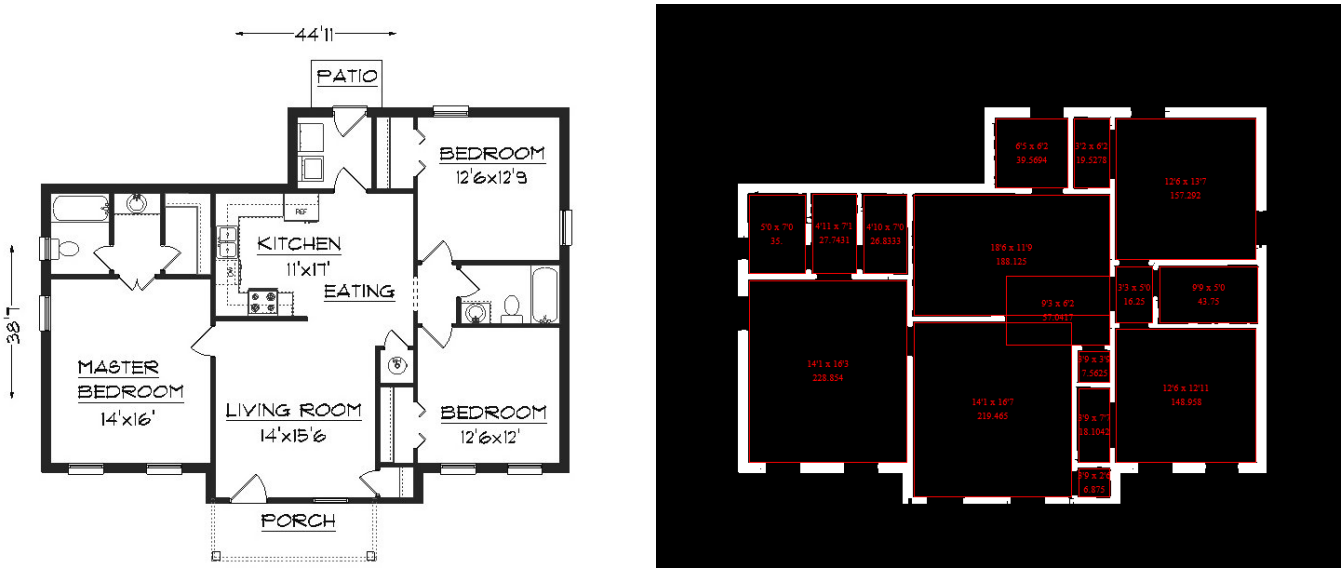
The output of the Hough Transform or Contour detectors produce useful information to find out details about a room. An implementation of this useful information is shown in the example in the next subsection.

[7]The graphic was based on the graphics from https://docs.opencv.org/2.4/doc/tutorials/imgproc/imgtrans/hough_lines/hough_lines.html

2.2.4 Room Detection in 2D

The initial approach decided on to complete the goals of this dissertation is to extract useful information (such as only data from points on vertical surfaces) and to then use this to find the layout of a room. This approach will keep the amount of data to process to a minimum, which is ideal for a robotic system as it may not have an extremely powerful processor. Upon researching this approach a paper by S. Oesau *et al.* was found[10]. The paper deals with taking a 3D point cloud model and segmenting the cloud. It does so by taking all the pixels in the point cloud which are located at a certain height in the z-axis, and plotting their x and y coordinates. By plotting the x and y coordinates a 'slice' at that horizontal level has been extracted. This is repeated then for multiple heights.

The slices at different heights are used to determine the walls of point cloud by finding common lines in each slice and concluding that these are walls. Most papers which deal with similar tasks use their own algorithms. During research, a 2D approach was found that performs region growing of a floor plan to detect individual rooms [29], which could be applied in OpenCV.



(a) Floor plan image inputted into algorithm[8] **(b)** Output of floor plan algorithm[9]

Figure 2.14: A 2D approach for creating a floor plan from an existing image[29].

The relation between pixel size and real-world distances will need to be calculated beforehand. This can be done by calibrating the dataset to the sensor being used.

[8]https://i.stack.imgur.com/qDhl7.jpg
[9]https://i.stack.imgur.com/dhu2r.png

2.2.5 Measuring The Real-World Size Of Objects

Allowing the digital-world to measure the real-world requires certain conversions to be made between the two domains. Without these conversions the recorded and processed data will mean nothing and serve no use. There exists multiple methods for calibrating sensors to represent real-world measurements. A method by Adrian Rosebrock in [30] explains how to measure the size of real-world objects using OpenCV.

To determine the size of an object from an image a 'pixels per metric' ratio is needed. This ratio represents how many metric units one pixel represents. To determine the size of an object for example, the amount of pixels the object occupies will be multiplied by this ratio, which will output the real-world size. This ratio will be dependant on the sensor, as well as the orientation of the sensor to the objects.

Determining this ratio is a form of calibration, and once it is known it doesn't have to be recalibrated unless variables within the scene change, such as the distance from the camera to the object. However, a method to overcome this is by having a reference object within the frame whose size is known, thus using it as a live calibration value. These principles of relating digital data dimensions to real-world dimensions must also be applied to point clouds outputted by SLAM.

2.3 SLAM

When robotics first began they would carry out tasks under the control of an operator. As robotics became more popular the need for automating them arose, however, when automation occurs there will be some error within the system which the system itself must compensate for. A paper released by the Massachusetts Institute of Technology in 1982 explains the design of a "plan checker", one which will check that the 'plan' which was given to the robot is being carried out correctly [31].

When robots began moving around this approach was adjusted for the navigation of a robot in an unknown space. The estimation of the robot's location must only use sensor information from the robot. This is explored in detail in a paper by Smith and Cheeseman in 1986 [32]. The problem explored within [32] gained popularity and become known to academics and hobbyists as Spatial Localisation and Mapping (SLAM). In the world of mobile robotics, SLAM plays a large role, helping to answer these three key questions: "where am I, where am I going, and how do I get there" [4].

SLAM is required to make a fairly accurate map for the navigation of the robot, however, the map that has been created could also be exported and used in other applications. The map of a room for robot

navigation is normally done in 2D and will have data for the space in question at the ground level, as this is where the robot is moving around, but this isn't an accurate representation of the room. More advancements have been done to incorporate SLAM with 3D sensors to produce point clouds.

The point clouds are not captured instantaneously, but in sections, and stitched together using features of objects to determine correlation. This is where feature detection, as discussed in Section 2.1 and 2.2, is useful. Problems arise if the system does not know exactly how far it has travelled between subsequent scans, resulting in errors in the constructed map.

There are various approaches to fixing this problem. Along with the sensor scanning the environment, other sensors could be placed in the system to better track the location of the system within the environment. In the case of a wheeled system, the wheels could have encoders on them to assist with the estimation of the location of the system within the space.

Simultaneous Localisation and Mapping (SLAM) is the process whereby a system constructs a map of an unknown environment using features it detects within the space, whilst simultaneously keeping track of its own location within the environment [4]. An example of a result of SLAM can be seen in Figure 2.15. The environment shown here is a 2D floor mapping of a room for robotic navigation.

The random lines extending out from the room are from points that have been placed incorrectly due to noise. The points could also not be placed incorrectly and may actually be accurately measured points on the outside of the room through a window or door. Although these points may be 'accurate' they are not useful for modelling this room, and therefore fall into the noise category. Any modelling of real-world environments will contain some error due to the presence of noise or unnecessary points. An approach to mitigate the error is by using loop closure.

2.3.1 Loop Closure

The error within a mapped environment will come from multiple points within the system. These include: the error in the sensor(s) within the system (be it noise or non-accurate components), bad features within the environment, or calculation errors within the algorithms. To minimise these errors, loop closure is used. A loop is considered to be found when the system scans an area it has already scanned - and realises this.

For explanation purposes, the already scanned area within the environment is position A, and the calculated position of the system at the current point in time is position B. These positions should be equal, but if these positions are not equal then the system knows it's calculations for its location in the map is incorrect. Knowing that position B should be equal to position A, it re-adjusts the map. This

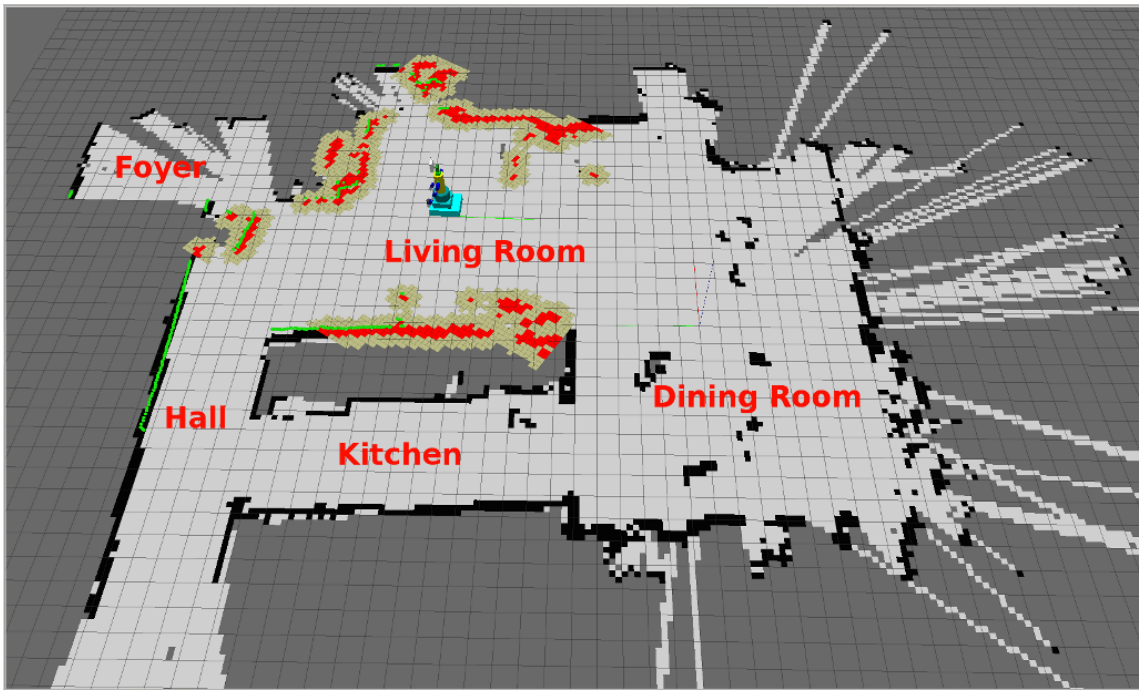


Figure 2.15: An example of 2D SLAM. The environment was captured using an automated robot. [10].

re-adjustment will attempt to account for the build up of errors to this point in the scan. This can be seen in Figure 2.16. Loop closure allows the use of cheaper, not so accurate components, as their errors can be fixed in the processing stage.

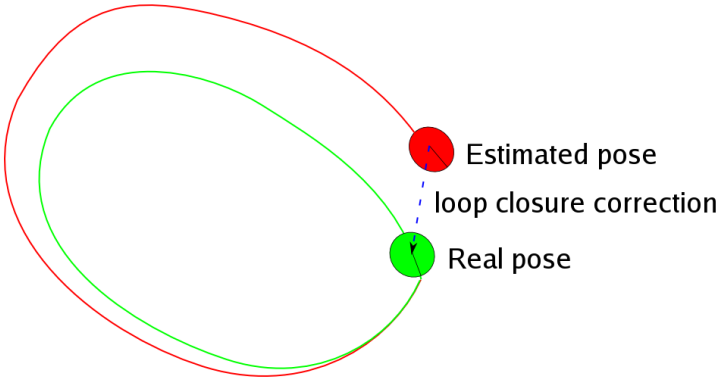


Figure 2.16: Visualisation of loop closure within a system[11].

Whilst scanning, objects can obstruct other objects within a room. This could hamper the accuracy of loop detection, because an object which has been observed from one orientation may not be observable from another orientation - potentially preventing loop closure from happening. This is better explained

[10]http://www.pirobot.org/blog/0015/map-1b.png
[11]http://cogrob.ensta-paristech.fr/assets/loop-closure.png

in Figure 2.17, whereby the red wall is hidden behind the orange wall. If this wall was observable from another point, the system would not know it is seeing it again and not perform loop closure. As loop closure is the most important aspect to ensure accuracy in SLAM, especially Visual SLAM, a room may need to be scanned multiple times to ensure all points are viewed from all viewpoints.

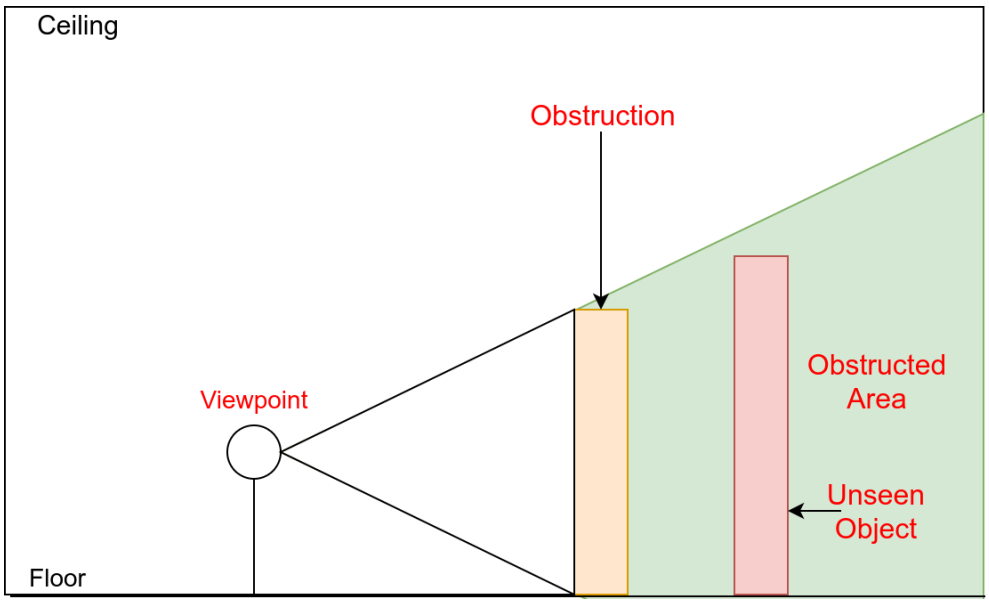


Figure 2.17: Visualisation of object obstruction during room capture.

2.3.2 Visual SLAM

With increased processing power, SLAM can be done in 3D in real-time, and the easiest method is with the use of visual inputs, such as cameras. The SLAM pipeline, shown in Figure 2.18, is a very broad overview of many algorithms working together to produce a 3D rendered Point Cloud.

Within the first box of the figure, "Visual Odometry Estimate poses", many tasks are being performed here. Some would include: feature detection, stitching of scans, error correction from sensors, etc. Within this one step many sub-systems will exist, and within some of those sub-systems will lie other sub-systems, and so on. Visual SLAM, and SLAM in general, is a very complex task.

As SLAM is not a focus for this dissertation, but merely a means of data collection, it is not explored any further. There exists many libraries out there that perform 3D point cloud reconstruction, using SLAM as a sub-system, and many of them are available as packages for the Robot Operating System (ROS).

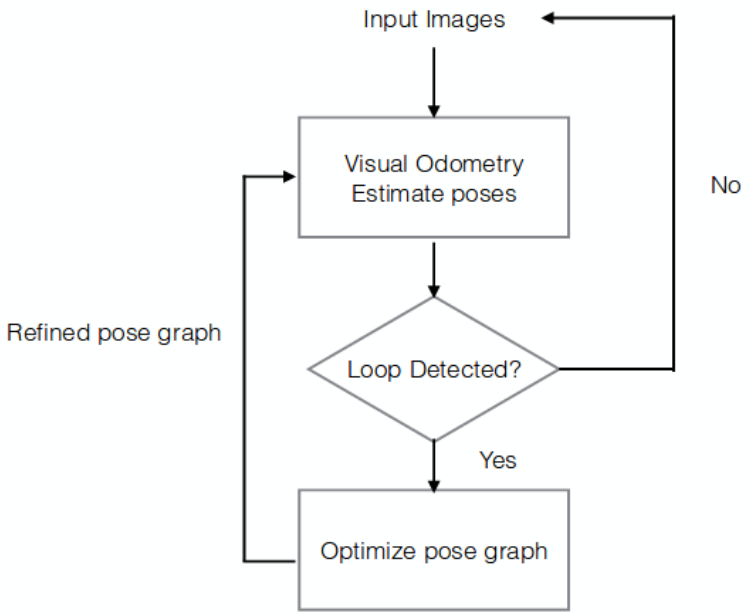


Figure 2.18: Visual SLAM Pipeline

2.3.3 ROS and RTAB-Map

The Robot Operating System [33] was explored, as the capturing of the point clouds was intended to be automated. However, ROS is not simply just an operating system for robot applications, it is a collection of tools, libraries, and conventions which carry out a multitude of operations. ROS libraries are a collection of open-source work from many individuals, and has knowledgeable users in its community forums providing extensive help.

A complete installation of ROS includes both OpenCV and the PCL. These libraries are used for most processing algorithms in ROS. The main features of ROS are room exploration and room mapping, therefore implying the use of SLAM algorithms. A ROS library that takes care of the creation of Point Clouds and SLAM with the use of an RGB-D camera is called RTAB-Map.

RTAB-Map (Real-Time Appearance-Based Mapping) was created by Mathieu Labbé and uses RGB-D sensors and SLAM to produce a point cloud [12]. Although this library is available as a ROS package, due to its popularity, it has been designed to be able to run stand-alone. The package can work with many RGB-D sensors which are available on the market. ROS itself can work with different types of depth sensors, the main other variation being LiDAR.

2.4 Depth Capturing

SLAM makes use of 3D information to assist it in mapping a room. The depth information needs to be as accurate as possible for the system to ensure that it is mapping the room accurately and keeping track of itself well enough. There are many ways to capture depth, and using a variety of sensors. The main types covered in this section are LiDAR and visual sensors.

2.4.1 LiDAR

Laser SLAM requires scanning the environment in question with a LiDAR sensor. This laser-type sensor measures the round trip time between the sensor and the object off of which the laser reflects. The round trip time is used to calculate the distance between the sensor and object. The following are some of the major pro's and cons for the use of LiDAR for SLAM (these insights are based on [34]):

- Pros:
  - Large range (up to 40m [7])
  - More accurate than most other sensors
  - Some sensors like the Sweep [7] can scan 360 degrees at a time
- Cons:
  - Only scans a single point, resolution is determined by how finely you can move the sensor
  - Cannot detect anything which the light would pass through (i.e. a window) or not be reflected by (something which absorbs the light)

If time is not a constraint then LiDAR is definitely the superior choice. Currently, it is used to accurately map buildings, inside and out, but requires a lot of time to stitch subsequent scans together. This is due to the drift factor caused by the platform moving while scanning. A drift factor is when the synchronisation between the scanned points and the location at which they were scanned from is not accurate.

This is because the scanner will usually be moving whilst capturing the point cloud. This needs to be accounted for in the outputted point cloud during post processing, usually using loop closure. The reason this is not often done in real-time is because a LiDAR point cloud is often very dense (i.e. has a lot of points) which requires more time to capture. This longer duration of capture between points

results in a larger drift factor and requires a lot of processing power. However, if the amount of points captured per second is kept relatively small, then it can be done in real-time.

Ji Zhang *et al.* created a 3D scanner using LiDAR that maps a room in real-time [35] using a sensor that captures information at 40 lines per second. Ji Zhang *et al.* processes the input data with two parallel algorithms, where one runs at a higher frequency accounting for the movement of the sensor during image capturing (drift factor), and the second at a lower frequency for fine matching of points. It is a novel approach for this type of modelling and could potentially be adapted for larger sensors by adding in more processing power.

Although their system works in real-time, it is using a sensor which captures less information about a scene than that used in industrial settings. An example of an industrial LiDAR system is the Lynx HS-600 which can scan up to 600 lines per second according to its data-sheet [36]. The amount of information captured from a scene by the industrial system is over ten times the amount the system used by Ji Zhang *et al.* captures. A real-time system is not only determined by the algorithms used but also the size of the data being processed. However, the paper by Ji Zhang *et al.* does go on to show that the error produced by the system for indoor environments is approximately 1%. Therefore, also indicating that the highest quality of sensors may not necessarily define how accurate your system can be.

This adds on to the idea introduced in Section 2.1.2, whereby downsampling a point cloud saves you space and reduces processing time, but keeps enough data so the point cloud is still useful. An industrial LiDAR system may obtain more points, allowing for small details in the observed space to be visible, but for room modelling this is not necessary. A room consists of large walls which don't need many points to represent them, therefore a smaller sensor, like that used by Ji Zhang *et al.* can be used. Different methods of depth-capturing will use different terminology to describe how much information they can capture in a scene, in the case of LiDAR it is lines per second; visual sensors are normally rated by the pixel resolution of their sensor.

2.4.2 Visual Odometry

Cameras are able to record enough data to be used for SLAM as well. A camera is able to see entire sections of a room at once, unlike LiDAR which can only examine one point at time. This will require less time to scan an entire room. The better choice instead of a normal camera is a camera with a depth sensor attached or built in. The type of sensor used for this is an RGB-D camera or a fusion system - which makes use of a normal camera and a LiDAR system [37].

The fusion system will use a normal LiDAR sensor to get the depth points, then using synchronisation

techniques between the LiDAR and camera, the RGB information can be overlaid onto the points in the 3D point cloud generated by the LiDAR sensor. Depending on the system this can be done in real-time or by using post processing.

An RGB-D sensor will stream an RGB matrix with the colour information for each pixel in the camera (i.e. a digital image). A matrix of the same dimensions will also be streamed containing depth data for each pixel of the camera; therefore, each depth point will have a corresponding RGB value. An RGB-D sensor can work in a few different ways.

Stereoscopic Images - Dual Camera System

This approach allows a system to calculate how far objects are from the system by using two cameras. Stereoscopy is a technique used to create 3D point clouds using only images [38]. The depth is calculated by placing two cameras slightly apart from each other and synchronising their image capturing (only if the system is moving, otherwise this isn't too important). The cameras should be horizontally aligned, as this makes processing the overlap of the images easier.

If they are horizontally aligned then the algorithm knows the overlap should occur by shifting the images in only one dimension over each other: left-right. However, this isn't a requirement if you are willing to spend extra processing power finding the overlap in more than 1 dimension. This would include having to shift the images over each other in the left-right direction for each possible up-down overlap. It will still work as long as there is some overlap. The ideal camera setup is shown in Figure 2.19(a) as this negates any vertical error - requiring additional processing. Figure 2.19(b) shows a tree observed by two camera, for which the distance of the tree from the cameras can be calculated. To calculate the distance, D, we can use the following equation:

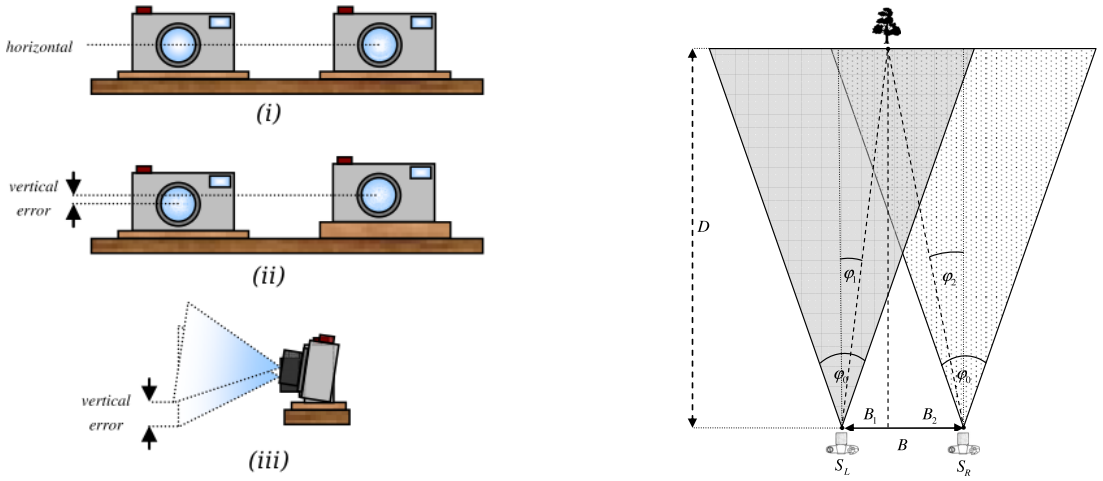
$$D = \frac{B}{\tan\phi_1 + \tan\phi_2} \tag{2.2}$$

B - Represents the distance between the cameras

ϕ_0 **-** The cameras horizontal angle of view

ϕ_1 **-** Left camera's angle between its optical axis and the object

ϕ_2 **-** Right camera's angle between its optical axis and the object



(a) Horizontal alignment of stereoscopic cameras.

(b) A tree observed by two cameras.

Figure 2.19: In (a): The proper horizontal alignment of the cameras (i). The incorrect horizontal alignment with vertical errors (ii) and (iii). In (b): An illustration of the camera setup in an environment [38].

Using simple trigonometry the following can be calculated:

$$\frac{x_1}{\frac{x_0}{2}} = \frac{\tan\theta_1}{\tan(\frac{\theta_0}{2})} \tag{2.3}$$

$$\frac{-x_2}{\frac{x_0}{2}} = \frac{\tan\theta_2}{\tan(\frac{\theta_0}{2})} \tag{2.4}$$

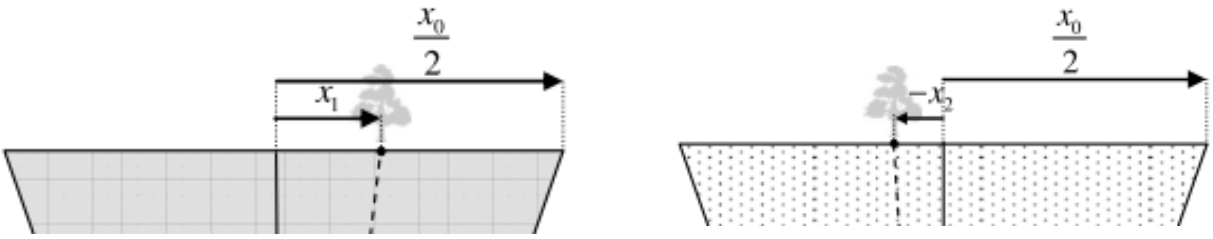
$$D = \frac{Bx_0}{2\tan(\frac{\phi_0}{2})(x_1 - x_2)} \tag{2.5}$$

x_0 **-** Number of horizontal pixels

$x_1 - x_2$ **-** Horizontal difference between the same object on both cameras

Therefore, using equation 2.5, the distance of an object within the view of both cameras can be determined within a certain error. The error is due to the discrete property of using camera pixels, as each pixel represents a certain real-world distance. This error is called ΔD and is defined by:

$$\Delta D = \frac{D^2}{B}\tan(\Delta\phi) \tag{2.6}$$



(a) View of tree from left camera.

(b) View of tree from right camera.

Figure 2.20: Location of object as viewed by both cameras[38].

$$where\ \Delta\phi = \frac{\phi_0}{x_0} \tag{2.7}$$

This approach requires visual feature alignment between the two cameras, if there are no features (a smooth white wall) the camera will not be able to calculate the distance. This is why most RGB-D sensors that work on this principal will project an infra-red scatter pattern which contains random dots that will give the image features if it lacks them. A camera which performs this type of depth-finding is the Intel RealSense Camera R200 [6].

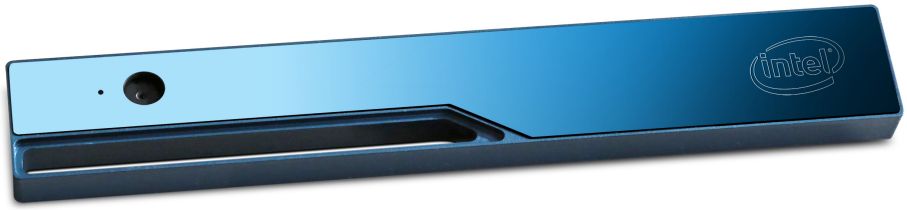


Figure 2.21: Intel RealSense R200 Camera[12].

The Intel RealSense's was primarily designed for use with laptops and portable tablets to model objects in a close range, up to 1.5 meters, in 3D [6][39]. The RealSense has been compared to other sensors, such as the Microsoft Kinect (covered next) with regards to 3D modelling, and comes out better overall [40]. However, the point of this dissertation is 3D room modelling, which will be the basis for the tests the sensors undergo[13].

The Intel RealSense sensor covered in this dissertation supplies both an image and depth stream to the user, with the depth data being calculated on-board the camera itself. The video stream comes from a separate third RGB camera as the two infrared cameras used for the depth calculation are greyscale cameras. The components of the Intel RealSense are structured as seen in Figure 2.22.

[12]http://reconstructme.net/wp-content/uploads/2015/11/r200.jpg

[13]As a note, at the time of completion of this dissertation, Intel released a new range of depth sensors specifically designed to model things at a greater distance up to 10 meters [41].

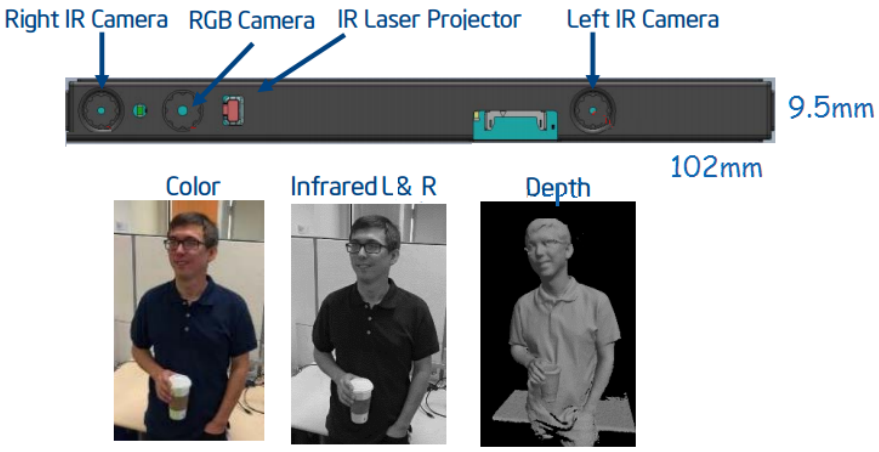


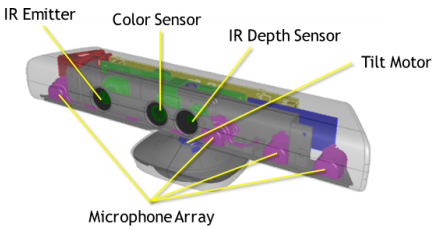
Figure 2.22: Intel RealSense Camera Components[14].

The colour camera is capable of full HD resolution (1920x1080) and the depth sensor can only achieve a maximum of 480x360 according to the data sheet [6]. To make sure the RGB pixels correlate with the depth pixels both streams will run at the resolution of the depth stream.

Single Camera with Depth Sensor

A sensor of this design will contain a single RGB camera, an infrared sensor, and an infrared projector. The RGB camera collects the colour image from the scene in question. This sensor works similarly to the stereoscopic approach. The infrared projector projects a scatter pattern which contains many unique clusters of points, similar to Intel RealSense mentioned earlier, onto a surface. An example of this type of sensor is The Microsoft Kinect [8], and is shown in Figure 2.23.

(a) Microsoft Kinect V1[15].



(b) Microsoft Kinect V1 component overview[16].

Figure 2.23: Microsoft Kinect V1 Sensor and Component Overview

The depth sensor itself is a camera and is of the same resolution as the RGB camera. By knowing the distance between the projector and camera the depth for each pixel can be calculated similarly to that

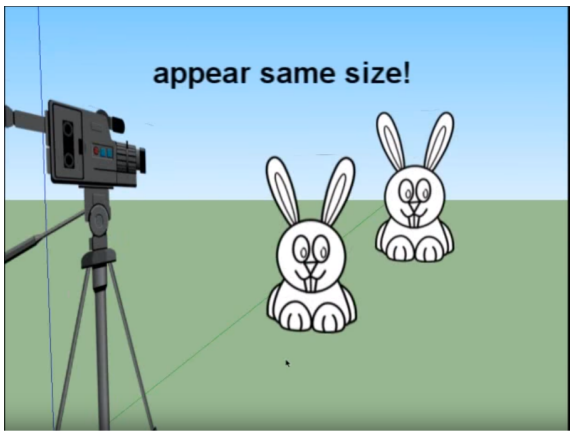
[14]https://software.intel.com/sites/default/files/managed/d0/5d/R200.png
[15]https://www.blogcdn.com/www.engadget.com/media/2012/05/k4w-sensorangle.jpg
[16]https://i-msdn.sec.s-msft.com/dynimg/IC584396.png

for the stereoscopic image. Therefore, this is similar to Figure 2.19(b), where one of the cameras would be the projector and the arithmetic remains the same.

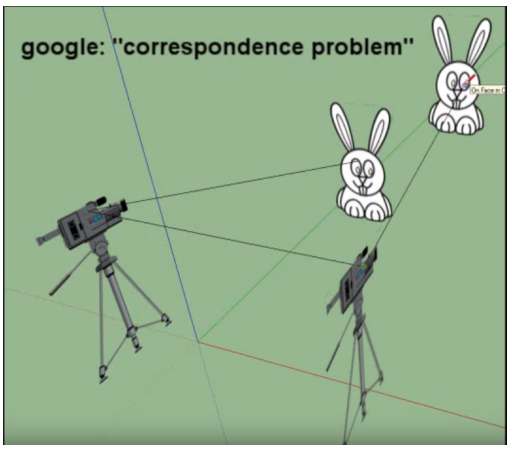
However, the objects that are detected are the clusters of pixels and not the features of the objects themselves. This makes the Kinect more robust as the features it is detecting will be the same every time regardless of the object. As with the real-sense, using this scatter pattern allows distance to be calculated even if the object itself has no features.

This type of sensing also gets rid of another problem, called the **correspondence problem**. When determining distance using stereoscopic principles, the features of an object allow the system to locate the object on both sensors. However, if there exists two of the exact same objects within the view of the system, it will get confused. If the first identical object is detected by the first sensor, and searched for on the second sensor, it won't know which one is the right one. Also, if the objects are identical, but one is larger than the other but placed at a different distance from the sensor such that it appears the same size from the viewpoint (shown in Figure 2.24(a)), the system will also get confused and not be able to return accurate depth information.

Figure 2.24(b) illustrates the confusion that can arise. The lines from each camera in Figure 2.24(b) should meet on the same object in order for the trigonometric equations for stereoscopic depth finding to occur. However, with one camera analysing the position of the first object on its sensor, and the other camera doing the same with the second object, the trigonometric equations will not work [42].



(a) Two of the same objects of different sizes placed at different distances



(b) The stereo camera set-up showing the confusion of similar object

Figure 2.24: Correspondence problem visualised [43].

Using Figure 2.25 as a reference, the scatter pattern can be thought of as a grid the same size as the pixel resolution of the camera. If the projector is thought of as one of the cameras in Figure 2.19(b) and each block as one of its *received* pixels we then have the horizontal location of each block in the scatter grid. As the clusters (or blocks) are considered the objects being detected, it is just left up to

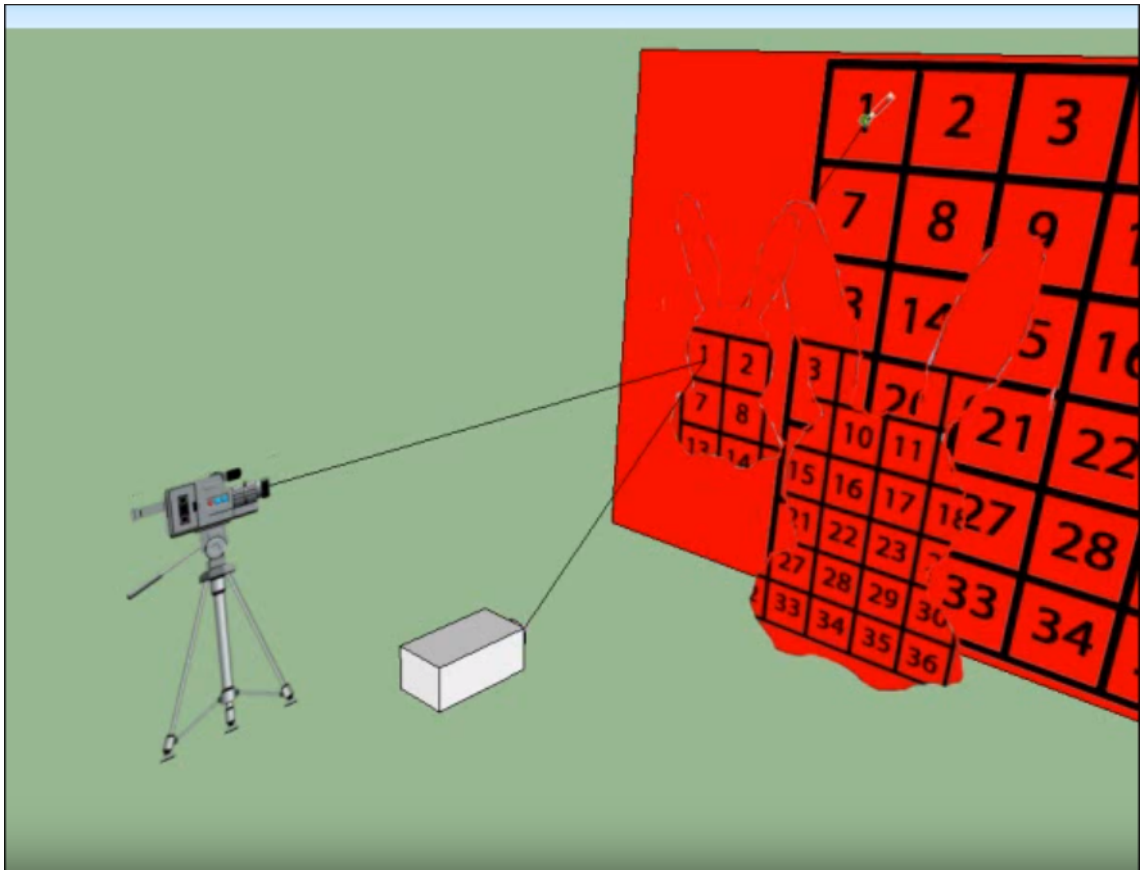


Figure 2.25: Microsoft Kinect IR Projector - Depth Finding illustration [43].

the sensor to calculate the distance of each pixel, or 'block', from the sensor itself.

There exist many papers that use the Kinect for interactive applications due to the Kinect being so readily available. Many papers dealt with tracking of people [44] [45] [46] and a few dealt with determining just how accurate the sensor actually is [47]. The accuracy tests performed on the Microsoft Kinect gave valuable information towards how accurate the sensor is when examining a stationary object, or just having to detect an object, but none seemed to cover the accuracy of the sensor when used with 3D room modelling software.

Single Camera with Time of Flight Sensor

A time-of-flight (TOF) sensor is able to determine how far away an object is by using the time it takes for a signal to reflect off an object and return back to the receivers location, which is usually located at the transmitters position. This is the same principle that LiDAR uses. The type of signal transmitted is normally an electromagnetic signal which travels at the speed of light.

The transmitted signal reflects off of the object in question and the time difference between the received

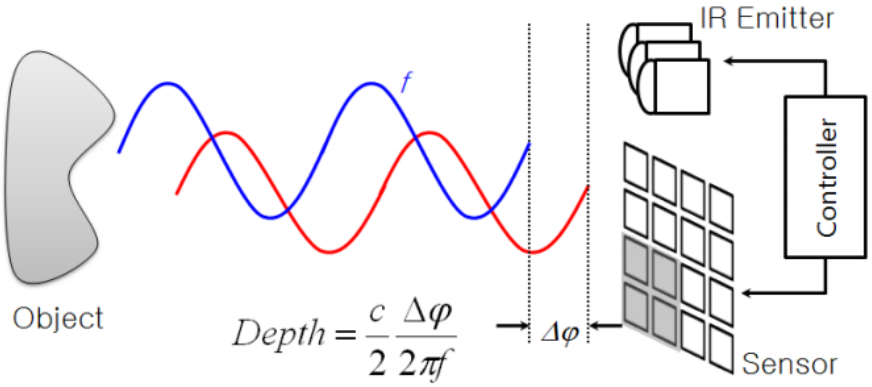


Figure 2.26: Time-Of-Flight illustration of a grid-type sensor [48].

signal and transmitted signal is used to calculate the distance. The distance light will travel in a given time frame is equal to $c * t$, where c is the speed of light. As we can see from Figure 2.26 if the phase difference that is returned is greater than one period, 2π, of the transmitted signal then the system will think that the object is at a distance which it is not, meaning its distance is ambiguous. This is because one period of the signal looks exactly the same as the next and so on, therefore it cannot differentiate between different periods. The maximum distance we will be able to measure is how far light will travel over one period T. The maximum distance travelled in one period is $c * T = \frac{c}{f}$ where f is the frequency of the signal being transmitted. The total distance that needs to be travelled is the trip to the object and back, therefore the maximum distance that can be measured is $\frac{c}{2f}$.

The equation shown in Figure 2.26 is the maximum distance that can be measured ($\frac{c}{2f}$) multiplied by the percentage of phase shift ($\frac{\Delta\phi}{2\pi}$), which corresponds to the time it took the signal to return. A camera that uses this approach is Microsoft's Kinect V2. This camera has large infra-red emitters in order to illuminate the entire room.

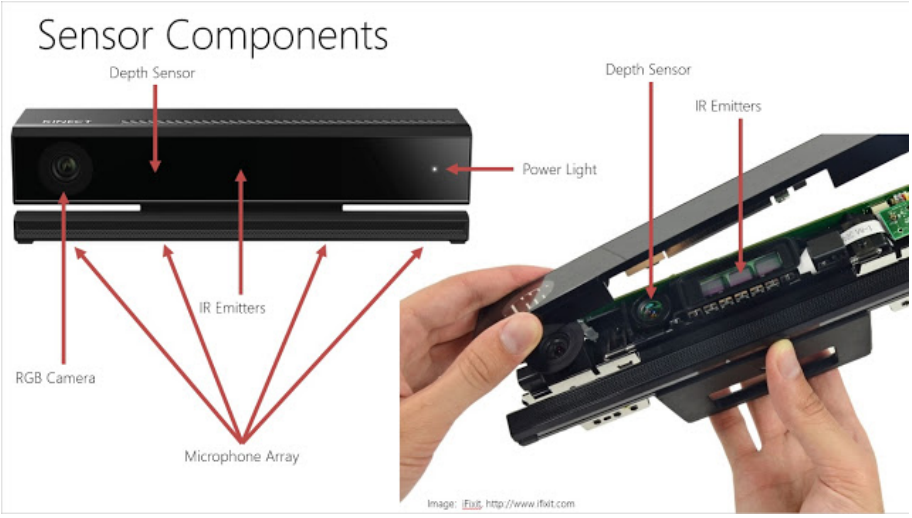


Figure 2.27: Microsoft's Kinect V2 Camera[17].

The depth sensor has a depth resolution of 512x424 [49]. Each pixel in the depth sensor, shown in Figure 2.27, is ideally halved, with each half being able to be switched on and off very quickly i.e. absorbing and not absorbing photons [50]. Each half is offset by 180 degrees, meaning that when the one is on, the other is off, and vice versa. To find the TOF the IR transmitter is pulsed on and off. When the IR transmitter is transmitting, one half of all pixels will be absorbing. When it switches off the other half of each pixel will be absorbing whilst the first half is no longer absorbing. The ratio of the photons received between each of the halves of a pixel will determine how far away the object is in that pixels view.

The pulse time of the camera's IR transmitter will determine its maximum unambiguous distance. The TOF illustration in Figure 2.28 shows the periods when the two halves of a pixel are on and off, along with the received photons (in red) on the sensors. The top row of photon's are from an object right in front of the sensor, reflecting the wave immediately. The middle row of photons are from an object approximately half of the maximum unambiguous distance from the sensor. The bottom photons show the result if the object is further than the unambiguous distance.

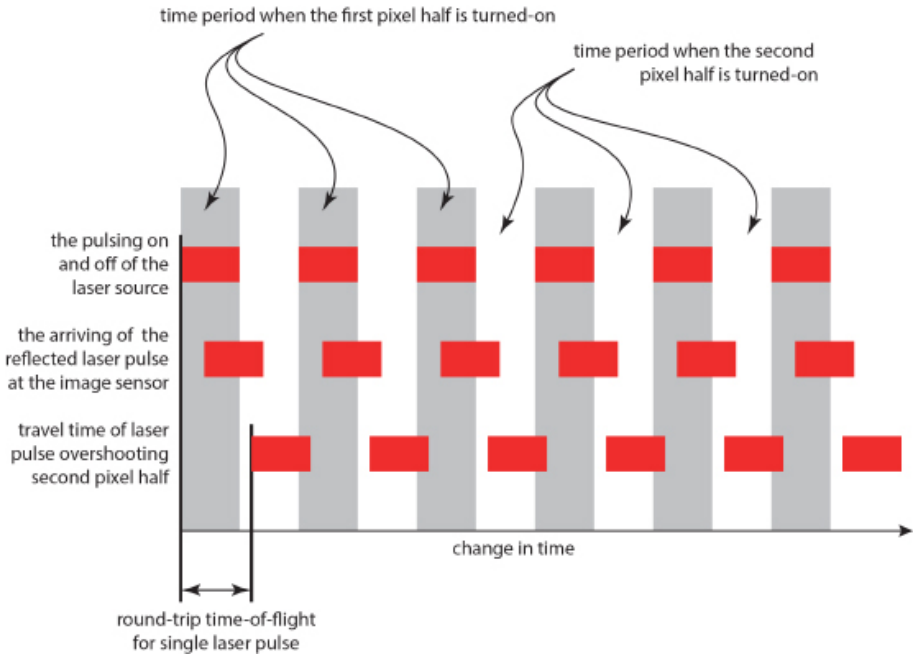


Figure 2.28: Microsoft's Kinect V2 Camera TOF Illustration[18].

If the frequency is decreased then the maximum unambiguous distance will increase, however, this will decrease the distance resolution of the system (*resolution refers to the ability of the sensor to distinguish between two different objects at similar distances*). This is due to the same ratio values representing the larger range. If each half of a pixel will represent a value between 1 and 500, the range of the difference between the two will be -499 to 500 (which is a total of 1000). If the maximum unambiguous distance is 1 meter, the resolution will be $\frac{1000\ mm}{1000} = 1\ mm$. If now the maximum unambiguous distance

[17]https://www.physio-pedia.com/images/d/d9/Microsoft_Kinect.png
[18]http://lau.engineering.uky.edu/files/2013/11/Slide4.jpg

was increased to 5 meters, the resolution will be $\frac{5000\ mm}{1000} = 5\ mm$. To get large ranges but still keep accuracy the Kinect changes its pulse-interval on-the-fly. Thus getting the actual unambiguous distance of the objects with a slow frequency, then with this data it can shorten the pulse-interval (increase the frequency) and more accurately calculate the distance of the objects [50].

The various depth sensors mentioned above perform all their depth processing on-board, and only return the depth information to the user. In order to test the quality of this depth information from the sensors adequate ways of measuring them will have to be found.

2.4.3 Measurement Methodology

There exist different measurement methodologies for many things in the world. The most applicable to the focus of this dissertation, being on depth sensors, would be that similar to RADAR system [51]. The ideas within [51] which are of interest are the accuracy and distance resolution of a sensor. Accuracy refers to how accurately the sensor is able to determine the real-world distance. If the sensor is able to determine the real-world distance accurately for every scan it performs, it is then considered to also be precise. The difference between accuracy and precision can be explained using Figure 2.29 as a reference.

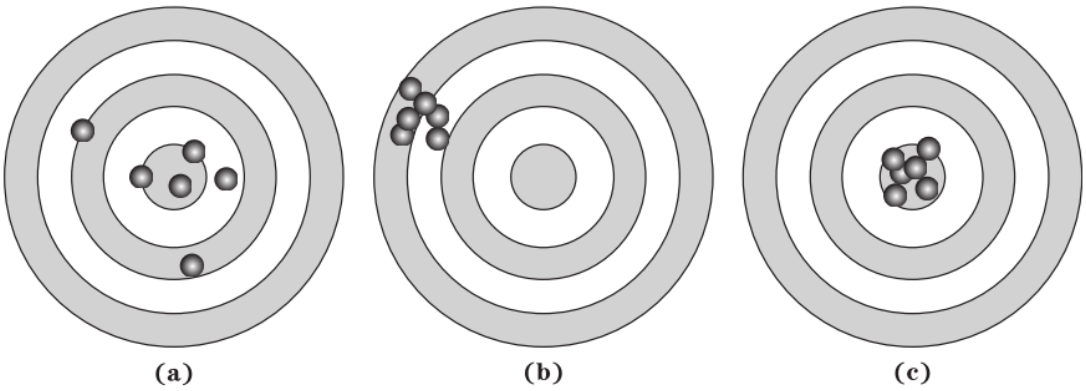


Figure 2.29: The difference between accuracy and precision. (a) Shows an accurate system which is imprecise. (b) Shows a precise system but inaccurate. (c) Shows a precise and accurate system [51]

.

Figure 2.29 shows targets with points on them. If a system is considered accurate it will have a low average difference between intended and actual results, however, the standard deviation between the results will be high. This is shown in Figure 2.29(a). If the system is precise it means that the standard deviation is low, but the difference between actual and expected results are large, shown in Figure 2.29(b). The ideal system will be both precise and accurate, seen in Figure 2.29(c). Therefore, any system from which the accuracy is taken should also be examined for the precision of the data.

Distance resolution has already been touched on in Section 2.4.2 for the stereoscopic and TOF sensor. Distance resolution is the minimum difference in distance the system is able to detect. It is ideal to have a system which can detect as fine of a difference as possible, but the finer the distance resolution is made other sacrifices have to come into play. In the case of the TOF sensor, it will be the maximum unambiguous distance the sensor is able to detect. For the stereoscopic sensor, the pixel resolution of the depth cameras (the two greyscale cameras) will have to be increased. However, if the pixel resolution is increased the processing power will need to be increased to handle the extra pixels, and in turn might result in a larger system or one which becomes more expensive.

Therefore, accuracy and distance resolution are useful tests, as they apply to any sensor that performs depth capturing. These tests will then be able to adequately compare various sensors for the quality of their depth results.

Chapter 3

Methodology

The research papers and tools listed in Chapter 2 helped determine what approaches to follow for the completion of the project. The method used by Oesau *et al.* [10] helped form the basis for the approach used in this dissertation. However, instead of converting the raw point cloud directly into horizontal 2D slices, the 3D point cloud was processed in 3D using the PCL, and then its horizontal slices were extracted and processed in 2D using OpenCV.

The accuracy of the sensors being used in most research papers with regards to 3D room modelling is never touched on [2] [5]. In these research papers, the point clouds are imported assuming they are accurate, and the papers then deal with the processing of the point clouds. However, when dealing with a system which can make use of any OTS sensors, the choice of OTS sensor will greatly impact the accuracy of the system as a whole. Therefore, tests to determine what makes a sensor optimal were created and used to determine which of the sensors available (RealSense and Kinect) are best for 3D modelling.

The floor plan creation followed a V-Shaped life-cycle design model, seen in Figure 3.1. This model was chosen due to its constant verification with the requirements and designs. After the system was designed each of its subsystems were verified against their design functionality. If this wasn't done, the entire project would risk being completed with missing functionality, and therefore unmeetable requirements. Fixing this once a project has been completed and subsystems integrated, could create further complications where other subsystems stop working or the integration needs to be redone. Therefore, testing each sub-system earlier on in the design aided in less complications integrating all systems and performing system acceptance testing.

The floor plan creation was the initial task of the project and only these requirements and functionality existed to start. The V-Shaped model was used. During the implementation phase of the design model,

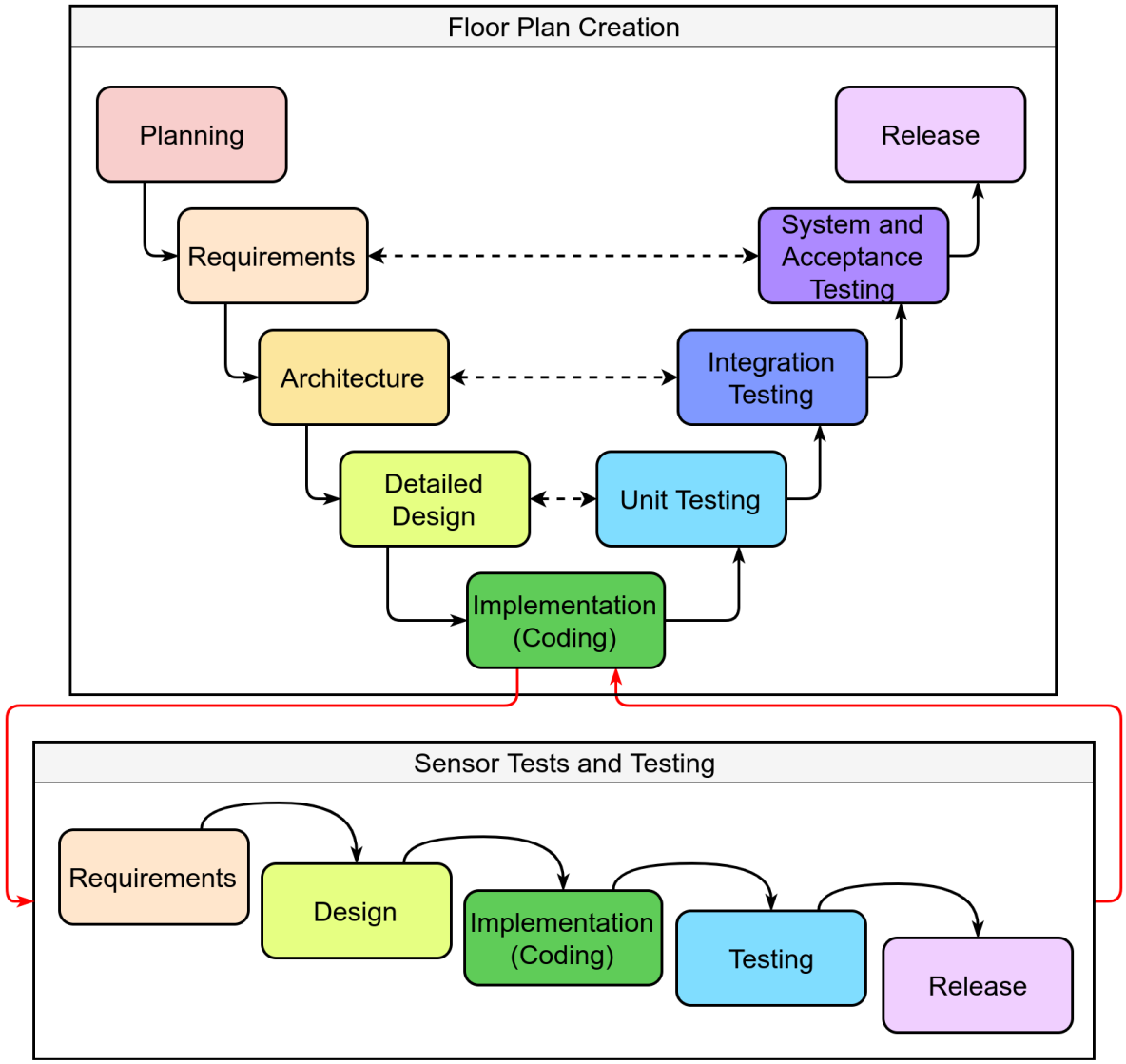


Figure 3.1: The two life-cycle design models used in this project - V-Shape and Waterfall.

whilst unit testing the Intel RealSense for basic 3D modelling, it was discovered that the sensor was producing seemingly noisy results. The implementation of the floor plan creation software was put on hold and the sensor testing part of the project started.

The design of the sensor tests didn't need to follow as rigorous of a process as the floor plan creation system, and therefore followed a Waterfall life-cycle. The requirements and functionality required for designing the tests were decided on and carried out. The system produced here was then implemented with the sensors available and the optimum sensor was chosen. The rest of V-Shaped Model was then carried out using this sensor for the tests.

The phases which were carried out in the project can be summarised as follows, linking closely to Figure 3.1. The phases which are similar between the V-Shaped model and the Waterfall model follow the same outline here.

Phase 1 System Planning

The system as a whole was planned here. Mainly pertaining to the floor plan creation system. Within this phase the literature required for this dissertation was researched. All literature relative to the information within this dissertation is included in Chapter 2.

Phase 2 Requirement Analysis

The requirements and functionality that are needed for the project was planned here, using the broad system planning in Phase 1 as a guide. The requirements and functionality of the testing system were only introduced during the implementation phase of the floor plan creation system.

Phase 3 Architecture and Design

The overall block diagram of how the systems will work was decided on (Seen in Figures 4.4 and 4.5). The subsystems were then created and designed according to the functionality that was required and documented in Phase 2.

Phase 4 Implementation

The floor plan creation and sensor testing software was created in this phase. Phase 4 and 5 were done as an iterative procedure, whereby each sub-system, once finished, was tested against the functionality it was intended to provide. If alterations were needed the implementation was changed until tests were passed.

Phase 5 Unit Testing and Integration Testing

Due to the inputs and outputs of each subsystem being known, their unit tests and integration tests were the same, and some were carried out in Phase 4. Each software unit was designed so that it would simply connect to the next one as easily as possible, therefore integration was part of the test for the individual subsystem.

Phase 6 Acceptance and Release

The requirements and functionality developed in Phase 2 were checked against the integrated system after Phase 5. If all requirements and functionality were met, the system is considered to have passed as a whole.

Due to this project being a heavily software based project, the cycle from detailed design to implementation, on to unit testing and back to detailed design, was the main portion of this project. A lot of ideas and approaches were explored in software, and when one approach appeared not to work a new approach had to be designed, implemented, and tested. The literature review of this dissertation serves as evidence of this approach as not all algorithms or methods mentioned in the literature review were used, but their exploration lead to finding other methods and implementing them for this system.

3.1 Objectives, Requirements and Functionality Development

The project has two main objectives that are further analysed by breaking them down into requirements and functionality in the next section. The two objectives are:

Objective 1 Generating a Floor Plan from a 3D Model

Objective 2 Determining the Optimum Sensor for Open Source 3D Modelling

The overview of requirements and functionality given in Chapter 1 were refined into separate points which needed to be met for the systems to be considered successful. The requirements and functionality were decided on in phase 2 of the system design life-cycle, and checked against in phases 5 and 6. In order to verify that a requirement and a functionality has been met, they have to be defined clearly. The purpose of the section to follow is to clarify the requirements and functionality.

3.1.1 Requirements

The requirements for both the floor plan creation and the sensor testing are listed below. Each requirement is accompanied by a short explanation making sure that the requirement is clear and will not be misunderstood.

Floor Plan Calculation

R1. **Import a point cloud and make use of its data.**

Making use of the data is being able to extract useful information from the point cloud, and is not limited to just being able to display it.

R2. **Remove unnecessary data from the point cloud.**

This is data, that when removed, does not change the accuracy of the results extracted from the point cloud.

R3. **Determine the location of walls from the data.**

This challenge is listed as a requirement as a point cloud will output points, not lines. It will be the job of 3D and 2D processing techniques to determine what are walls from these points.

R4. **Find dimensions of walls and visualise this.**

Once potential walls are determined, useful information must be extracted from these discovered walls such as length. The offset must be no greater than 5 %.

R5. **Automate the process from beginning to end.**

For ease, the process must be automated from beginning to end. Ideally, a single command must be called in the terminal that will produce a floor plan from a point cloud.

Sensor Testing

R6. **Determine tests for choosing an OTS sensor for 3D room modelling.**

The tests that are created must pertain to 3D room modelling capabilities. Therefore, all things related to depth data will be analysed, and things like colour accuracy, will not be examined as it does not pertain to 3D room modelling with depth sensors.

R7. **Using tests, determine the best sensor between the Intel RealSense and Microsoft Kinect.**

The tests that are created for **R6** must be used to determine which of the sensors available to this project is the optimum one.

3.1.2 Functionality

The functionality comes from the requirements above. The functions below give a more focused direction for the project in order to fulfil the requirements. Each required functionality has a short description to make sure there is no uncertainty about what the functionality must achieve.

Floor plan Calculation

F1. **Import two main types of point clouds: *.ply and *.pcd.**

The two main types of point clouds that most point cloud processing software was found to support were *.pcd and *ply. Therefore, the software created must be able to read these formats as well.

F2. **Remove noise from the data.**

Noise must be removed from the system in order to reduce the amount of unnecessary points, but to also decrease the amount of incorrect points that will incorrectly influence results.

F3. Remove unnecessary points.

Unnecessary points must be removed to increase processing speed. The points are unnecessary if removing them does not affect the results of the system.

F4. Remove horizontal surfaces.

The focus of the project is the location of walls, which are vertical. Therefore, the initial processing can remove all horizontal surface without the concern of removing useful information.

F5. Determine potential walls from scattered point data.

A point cloud is made up of points which contain no information besides their location. Therefore, the system must be able to determine what are walls from this data, making using of 3D or 2D processing techniques.

F6. Smooth potential walls.

Due to the noise of having scattered points representing walls, there will inherently be incorrect walls created. The resultant walls must be compared against each other to determine which are potentially the same wall, and smooth the results.

F7. Find length of potential walls.

The length of the walls must be found and measured to the nearest pixel.

F8. Display the dimension of the walls in real-world dimensions.

The dimensions of the wall will be in pixels. Therefore, these lengths must undergo a conversion to convert from pixels to real-world distances (cm). The goal of the system is to be within a 5 % offset.

F9. Create a single program to run all necessary code to automate the process.

The system requirement of automating the process will require a single program to be the controller for the entire system. If other programs or scripts are called, they must be called by the main program.

Sensor Testing

F10. Determine sensor accuracy test, and test sensors with this test.

The accuracy of the sensor must be determined. This will be a comparison of the distance value the sensor says it is recording versus the actual distance. The accuracy of the sensor should be within 5 % of the distance it is measuring.

F11. Determine sensor noise test, and test sensors with this test.

There is always going to be noise in a sensor. This test will need to determine how noisy the sensor is and whether or not its information is useful.

F12. Determine sensor distance resolution, and test sensors with this test.

The distance resolution is the smallest distance difference the sensor is able to detect. This is useful when determining the difference between two objects which are placed at a similar distance from the sensor.

F13. Determine sensor 3D room modelling capabilities test, and test sensors with this test.

The 3D room modelling capability of the sensor will need to be tested. Although there are other tests that have been defined that produce more definitive results, such as accuracy, this test must be included as a 'final check'. The reason for this is the sensor in question might perform well enough under the above tests, but the final goal is to determine how good it is for 3D room modelling therefore, 3D room modelling should be a test.

F14. Process results to make informed decisions on sensor choice.

Record the test results from the tests performed on the sensors. The recorded results must be processed to determine useful information about the tests. Ideally to determine if the sensors have passed or failed.

F15. Where possible, produce graphical plots to back up results.

Graphical plots aid in the explanation of information and should be recorded where possible to back up results listed in F14.

By using the functionality above for the floor plan creation, a sturdy system will be produced. The functionality is broad, but precise at the same time. It doesn't dictate how the noise removal, or wall location must be carried out, but by completing these milestones it will be one step closer to completing a floor plan creation system. By completing the functionality it will lead to the requirements being met as well, therefore completing the goal of creating a floor plan from a 3D modelled room. The broadness of the functions allow it to be applied to any floor plan creation system dependent on the goal.

This system's goal is the creation of a floor plan without furniture or random counter tops. Other systems might require this information, therefore their filtering techniques will be different, but the design of these systems can use the functionality described here as a basis, adding or removing functionality dependent to their system. For example, it might not always be necessary to remove horizontal surfaces if the roof or floor is used for calculation. The same can be said for the sensor testing.

The tests designed here were orientated around finding the best depth sensor for 3D modelling, but the tests could be altered if the requirement was different. If the requirement was for the ability of the depth sensor to work in a dark environment, or outdoors, other tests would be carried out.

3.2 System Design

The data captured for the tests were done using RTAB-Map and ROS, mentioned in Section 2.3.3. The recorded data was imported into the software where it was pre-processed using a C++ program that makes use of the PCL. This program outputs PNG images of the processed 3D model at different heights, or 'slices', which were processed by a Python script. The Python script outputs a final PNG with the dimensions of the room on the image.

The accuracy and resolution of the Intel RealSense and Microsoft's Kinect were researched, but no useful information was found comparing these two cameras, especially for 3D room modelling. 3D modelling information about the RealSense was found at [6] but even this information was not adequate to understand the quality of the RealSense's 3D room modelling. This information could also not be compared to 3D modelling specifications of the Kinect, as there were none. A Microsoft Kinect was then acquired and testing to compare them was performed.

For these tests the accuracy and resolution data was captured using a custom piece of software which accessed the raw data streams of the cameras and saved them to the hard drive of the computer. Mathematical scripts were then used to process this information and graphs and tables were created to visually display the data. The 3D point clouds were captured using RTAB-Map, mentioned in Section 2.3.3, and examined by eye.

3.3 Implementation

3.3.1 Test Designs for Sensor Choice

At the start of the project the sensor which was readily available for the project was an Intel RealSense R200 camera. Although Microsoft's Kinect was used in multiple paper's for room modelling, more than the RealSense, the RealSense was chosen as the initial sensor for this project. Whilst learning how to manually interface with the RealSense, it was discovered that the depth data from the camera was very noisy. The tests that were designed focused on the four areas outlined in the functionality - noise,

accuracy, distance resolution, and 3D room modelling capability.

3.3.2 Floor Plan Creation

Some papers performed their entire project in 3D, like in [9], whereas others processed the data in 2D, [10]. Both of these approaches have their pro's and con's. The advantage of 3D processing is you can extract useful information such as surface normals, as discussed in Section 2.1.1, or downsample the data, as discussed in Section 2.1.2, to pre-process the 3D models and improve overall processing performance.

2D processing has been around for a lot longer than 3D processing, leading to optimised 2D processing algorithms. Therefore, processing 2D data will be a lot faster than 3D processing, but in turn you sacrifice the useful information that the 3D data can provide.

For floor plan creation it was important to find where walls are situated, but additional information such as the height of the walls are not necessary. This is where the 3D information is no longer needed. It was found that it was more efficient to remove outlying points in 3D than it was in 2D as the 3D algorithms can more efficiently determine if a point belongs or is a result of noise.

3D processing is also used to determine the normals of all objects within a 3D model, and surfaces which contain normals pointing up or down were removed as it is safe to say they are not walls. Tools used within these implementations are found in the PCL discussed in Section 2.1. The 3D model, once finished being processed, is outputted at multiple PNG images representing the various horizontal slices of the point cloud.

The 2D slices are stacked together to get an average of the points representing the room layout. Using slices from the higher heights provides better results as there is less interference from objects within the room.

2D processing techniques such as the Hough Transform, covered in Section 2.2.3, are used to find lines from the sparse points produced by the 3D pre-processing software. The lines produced by the Hough Transform were then smoothed in order to average the possible location of the walls. The implementation of this was done using OpenCV, mentioned in Section 2.2, and multiple tools within the library were used.

3.4 Unit Testing and Integration

In order to verify that the functionality mentioned in Section 1.2 has been met the following tests were created. The results of the tests are a pass/fail. If all test for a functionality pass, that functionality is then considered completed. If all functionality is completed for a requirement, that requirement is considered to have been met. The different tests which are to be carried out on the system to verify the functionality are tabulated in Table 3.1. This table gives a description of the test and the question which is asked to determine if the test passes or fails. The association of a test to its functionality and requirement can be seen in Table 3.2.

Table 3.2: The test matrix linking a test to a functionality, to a requirement, and to an objective.

Section	Requirement	Functionality	Test
Floor Plan Creation	R1	F1	T1
	R2	F2	T2
		F3	T3
		F4	T4
	R3	F5	T5
		F6	T6
	R4	F7	T7
			T8
		F8	T9
			T10
	R5	F9	T11
Sensor Testing	R6	F10	T12
			T13
		F11	T14
			T15
		F12	T16
			T17
		F13	T18
			T119
	R7	F14	T20
		F15	T21

Table 3.1: The tests that will be performed in order to verify requirements and functionality.

Test	Test Description	Test Criteria for Pass
T1	Import the two main types of point clouds.	Is the system able to import the two main types of point clouds?
T2	Noise removal.	Does the system adequately remove noise?
T3	Unnecessary data removal.	Does the system remove unnecessary points?
T4	Horizontal surfaces removal.	Does the system remove horizontal surfaces?
T5	Line (wall) detection from point cloud.	Is the system able to find potential lines (walls)?
T6	Smooth lines (walls).	Can the system smooth out the potential lines (walls)?
T7	Determine length of lines (walls).	Is the system able to determine the length of the lines in real-world dimensions?
T8	Accuracy of the lines.	Are these lines accurate to the real-world dimensions? An offset of less than 5% would be deemed a pass.
T9	Display of information.	Is the system able to display the dimensions?
T10	User-friendliness of output.	Are the displayed dimensions readable?
T11	Automating the system.	Can the system import the point cloud and produce a floor plan autonomously?
T12	Accuracy test creation.	Has a method for determining accuracy been found?
T13	Use of accuracy test.	Did the accuracy test produce useful information?
T14	Noise test creation.	Has a method for determining noise been found?
T15	Use of noise test.	Did the noise test produce useful information?
T16	Distance resolution test creation.	Has a method for determining distance resolution been found?
T17	Use of distance resolution test.	Did the distance resolution test produce useful information?
T18	3D room modelling test creation.	Has a method for determining 3D room modelling capabilities been found?
T19	Use of 3D room modelling test.	Did the 3D room modelling test produce useful information?
T20	Usefulness of recorded data.	Have the results been processed to produce usable data?
T21	Creation of plots and tables.	Are the tables and graphical plots useful and don't conflict data in previous tests?

Chapter 4

Design

The two key focuses of the research required two different system designs to be developed, designed in Sections 4.1 and 4.2. The design for the floor plan creation is mostly a software-based approach, with the only real-world interaction being the capturing of the point cloud. The sensor testing made use of sensing real-world objects to see how accurately the system could interpret them in the digital-world. These are detailed in the sections that follow.

4.1 Floor Plan Creation from a Point Cloud

The three main components for creating the floor plan were tackled individually during the development stage. Once a sensor was chosen for the project (the Kinect) it was used to obtain a 3D model of a room. The model was obtained manually as opposed to being downloaded as it is unknown how much pre-processing has been performed on a downloaded 3D room model.

With the 3D room model obtained, processing was performed on this 3D model in the 3D space using the PCL. Once it was deemed 'processed', horizontal slices of the 3D model were outputted as PNGs. The PNGs were then used as input to the 2D processing Python script, as outlined in Section 3.2. With the inputs and outputs of these processing 'blocks' being known they could be designed separately, and with the creation of some simple test data, each module could be tested during their individual design. Once the integration of the various modules was required in order to further development, each integration was focused on separately, but keeping in mind the whole system. The three main components are:

1. Cloud Creation

2. Point Cloud Processing (3D Processing)

3. Image Processing (2D Processing)

The overall block diagram for this design can be seen in Figure 4.1. This block diagram shows the system design from capturing the real-world data and producing a point cloud, through to processing the point cloud and outputting a floor plan. The capturing of the real-world to create a point cloud is covered briefly here as it was necessary for the test section of the project. Figure 4.1 shows a sensor as input to a PC, on which there was ROS and RTAB-MAP (Section 2.3.3) running to produce a point cloud. This point cloud was used as input to the 3D processing algorithm.

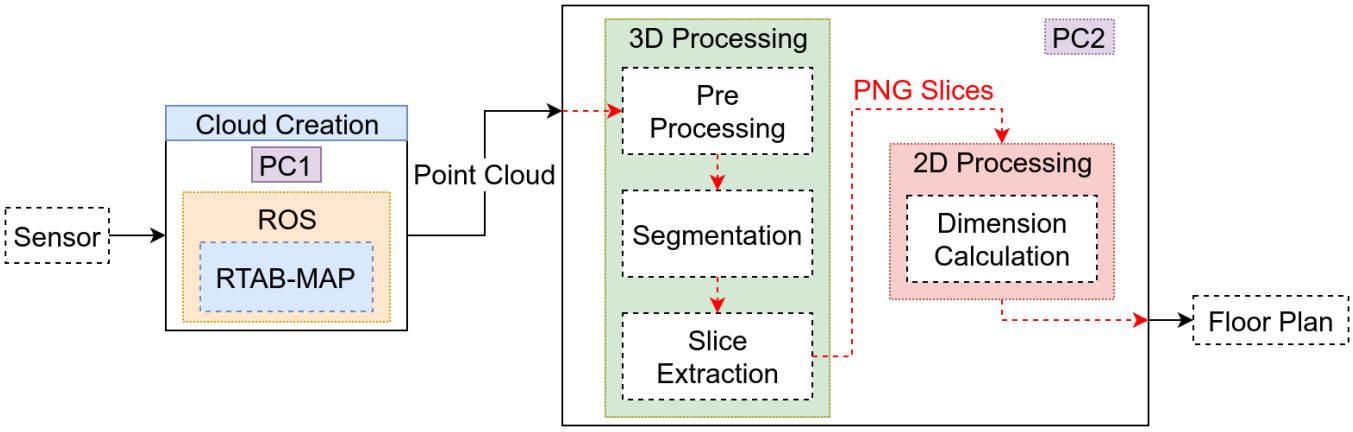


Figure 4.1: A block diagram showing the process from capturing the real-world data to outputting a floor plan.

The 3D processing algorithm made use of algorithms from the PCL for pre-processing the point cloud. Once it was pre-processed it was then processed to produce useful information, finally resulting in horizontal slices of the resultant point cloud being outputted. These PNG slices were then inputted to the 2D image processing algorithm.

The 2D image processing algorithm performed operations on these slices using tools from the OpenCV library. Once completed, the 2D image processing algorithm outputs a floor plan for use. The block diagram depicts two separate PC's involved, but this can all be done on one PC. The three components above will be discussed in greater detail below.

4.1.1 Cloud Creation

The creation of the point clouds used for this dissertation is explained from a very abstracted level as it is not the focus of the dissertation. However, the sensor used is one of the focuses of the dissertation

and is covered in Section 4.2. The sensor was connected to a PC running ROS and RTAB-MAP. The sensor was moved around the room, by hand, in order to map the room and produce a 3D model. Once the room was mapped the point-cloud was exported as a *.ply file, readable by the PCL. The block diagram for this is shown in Figure 4.2.

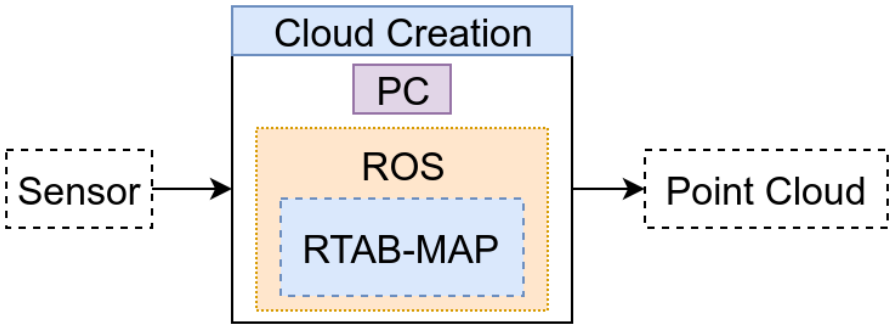


Figure 4.2: A block diagram showing the high-level process of creating a point cloud from a sensor.

The point cloud that was outputted was used as an input to the next system. The output could also be imported into any 3D point cloud visualisation software if there is another need for the 3D model. There is no code for this section as RTAB-Map is a standalone application with its own interface (Figure 4.3). The 3D model produced here was processed in 3D using the PCL, which is discussed in the next section.

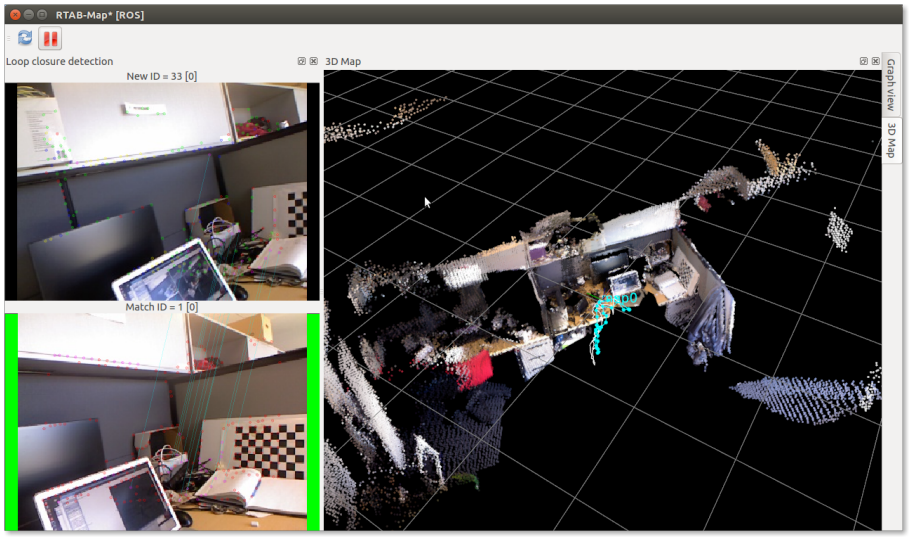


Figure 4.3: An example output of the RTAB-Map interface modelling a room. Upper left of the image is the current frame, bottom left is matching it to a previous frame. The right window is the constructed point cloud so far.

4.1.2 3D Processing

3D cloud processing was used to process the data before creating the floor plan. As mentioned in Section 3.3.2, 2D processing is far quicker but 3D processing provides more ways to manipulate data, such as being able to downsample and remove unnecessary data, in 3D space, for quicker processing. The 3D processing method is represented in the block diagram shown in Figure 4.4. The blocks in the block diagram are discussed below.

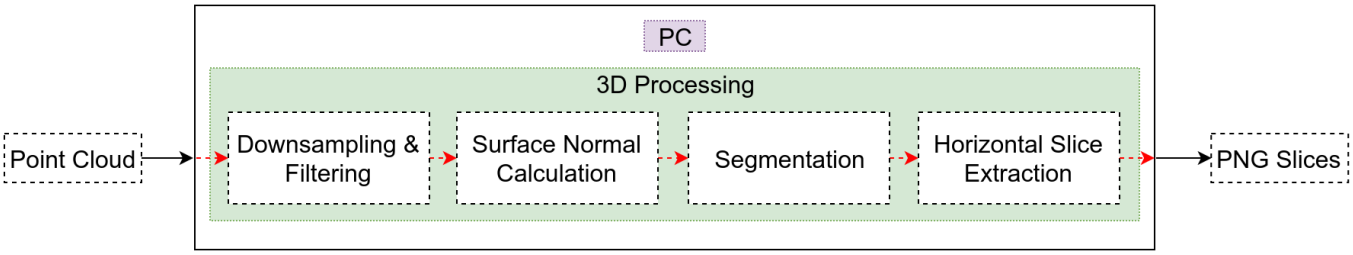


Figure 4.4: A block diagram showing the steps followed for processing the point cloud in 3D. Some of the steps shown here are depicting PCL libraries being used, whereas others are high-level abstractions of algorithms and methods which are not necessary for visualisation. If further explanation is needed, the code can be examined in B

Downsampling & Filtering Downsampling makes use of the Voxel Grid filter with a 3D box size of 2x2x2 cm. This may seem like a large box but its size will in fact assist in smoothing the point cloud and will not cause any important information loss for this system, as explained in Section 2.1.2. Therefore, the necessary surface normals will still be able to be created.

Surface Normal Calculation The surface normal calculations search the nearest 50 neighbours of each point to determine the normals of the point cloud. The value of 50 was chosen through initial trial and error, taking into consideration time and quality. A larger nearest neighbour, or larger radius, means that the normals within an area will not fluctuate from each other as much as with a smaller radius. This is explained visually in Figure 2.6. Having the normals for every point, all those which are horizontal can be removed manually by examining each point. Thus reducing the points that need to be processed for segmentation.

Segmentation The segmentation approach which the final system used is the extraction of planar components and not region growing. Planar extraction is explained in Section 2.1.3. This approach will extract any points which are considered to be on planes perpendicular to an inputted vector. Therefore, the inputted vector is $(0, 0, 1)$, relative to the (x, y, z) coordinate system, which is a vector representing a horizontal surface. Therefore, all vertical planes will be extracted.

Horizontal Slice Extraction The horizontal slice extraction will determine the total height of the scene, and separate the scene into 10 slices. The system will then output the top 5 slices which will be input to the 2D processing system.

The outputs of the downsampling, removing horizontal points, and segmentation are covered in Chapter 5, and shown in Figures 5.2 and 5.3. The code responsible for removing the horizontal slices is included in the appendices, but the pseudo-code is shown in Table 4.1.

The algorithm finds the total height of the scene. It then enters a for loop where it extracts the top half of the scene as slices. When extracting the slices, there's a chance there won't be points at the exact height interval calculated, so there is a tolerance which is used to determine a range of height that if a point falls into, will be extracted.

Table 4.1: Pseudo-code of horizontal slice extraction. The actual code can be seen in Appendix B

Line	Code
1	**RangeOfHeight** = **MaximumHeight** – **MinimumHeight**
2	**Tolerance** = 0.1
3	**ActualTolerance** = **RangeOfHeight***(**Tolerance**/100)
4	*for* (**i** = 0; **i** <11; **i**++)
5	*if* (**i** <6) **continue** *// Do nothing*
6	**HeightToExtract** = **MinimumHeight** + ((**rangeOfHeight**/10)*i)
7	*for every point "**p**" in the cloud*
8	*If the height of point **p** is between **HeightToExtract** +- **ActualTolerance***
9	***Add the point to the PNG***
10	***Save the resultant PNG***

4.1.3 2D Processing

In order to find the walls of the room from the PNG images outputted by the 3D processing block, a script making use of OpenCV will be used. The block diagram of how the system works is seen in Figure 4.5.

The PNG images from the 3D processing step were imported and the images were averaged in order to find the most prominent points, as a wall will be the most common point between the slices, and therefore the most noticeable. From this averaged data, the walls need to be located. The following steps are followed to achieve this:

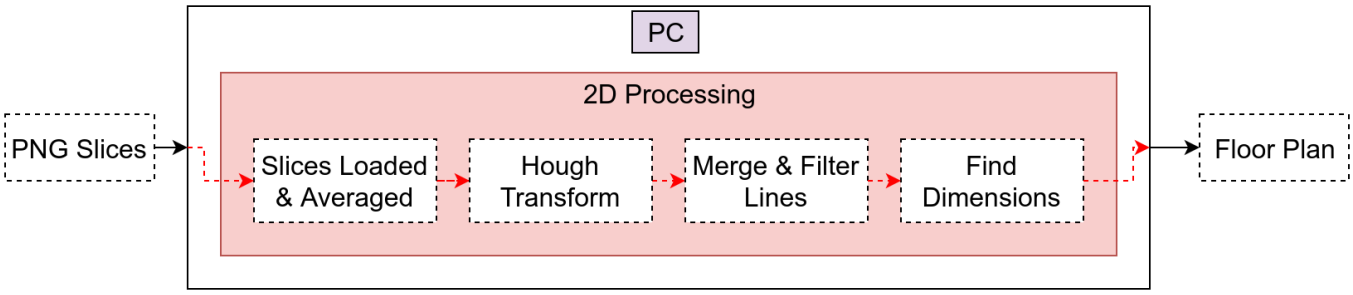


Figure 4.5: A block diagram showing the steps followed for processing the PNG slices in 2D. Some of the steps shown here are depicting OpenCV libraries being used, whereas others are high-level abstractions of algorithms and methods which are not necessary for visualisation. If further explanation is needed the code can be examined in Appendix C.

1. Dilate and Erode the image using OpenCV to get more prominent features and remove noise.
2. Find the Hough Transform of the resultant image.
3. Determine which lines are statistically the same line and merge them.

To complete these tasks multiple methods were created which are present in the Python script. The class of the Python script is shown in Figure 4.6.

2DProcessor

```
import cv2
import numpy
import math
import matplotlib
import glob, os
```

```
+ slope (line:array): gradient:float
+ lineEquation (line:array): gradient:float, yIntercept:float
+ findDistance (point1:array, point2:array): distance:float
+ getLineVariables (lineEquation:array): gradient:float, yIntercept:float
+ checkPerpendicular (lineEquation1:array, lineEquation2:array): boolean
+ lineIntersection (line1:array, line2:array): x:int, y:int
+ lineExtend ({xminU, yminU, xmaxU, ymaxU, x1U, y1U, x2U, y2U}:int) : line:array
+ proximityCheck ({line1, line2, extendedLine1, extendedLine2}:array, offsetGradient:float, {radiusToSearch, parallelLineDistance, imageHeight, imageWidth}:int): int
+ findLargestMinimumDistance (line1: array, line2:array, largeNotMin:boolean): line:array
```

Figure 4.6: 2D processing script class.

The class in Figure 4.6 shows which libraries were used and the functions (in orange) that were created. These functions take in various variables of different types (in blue). Variables of the same type are

grouped together using red braces. This processing script is called from the 3D processing software. Due to the 2D processing and 3D processing being two separate systems, programmed in two different languages, there is no direct linkage of their classes to create a class diagram. The best example of their interactions was shown in Figure 4.2.

The pseudo-code of how lines that are outputted by the Hough Transform are merged is shown in Table 4.2. This is one of the algorithms designed for this dissertation as it is not within a readily available library. The merging of the lines averages the lines and produces a line that is as long as the furthest distance between the lines thought to be the same. All the lines that are outputted by the Hough Transform are compared to each other. A line is chosen as the main line, and it is compared to all other lines one at a time. It does so by passing the two lines to the *proximityCheck* method.

This method will take in the **ParallelLinesDistance**, **OffsetGradient**, **RadiusToSearch** (it is not shown in the pseudo-code due to space constraints). The proximity is checked between the two line by first checking if their gradients are similar, the offset allows for lines with similar gradients to be declared the same. If the gradient is considered similar then their parallel distance is found, this is the closest distance between the lines if they were to extend to infinity. If they are within this offset their actual distance from each other is found, and if they are within the **RadiusToSearch** from each other they are classified as the same line.

If they are classified as the same line they are added to an array which holds all the lines that are considered to be the same line as the line in question. Once all lines have been compared the algorithm averages the lines in the **LinesToMerge** array. This will average their gradient and y-intercept. It will then calculate the minimum and maximum x and y values of all the lines, and use these to draw the new line using a simple line equation of $y = mx + c$ where m is the averaged gradient and c is the averaged y-intercept. This new line will replace the current line at the index **x** and remove all of the lines with which it was merged from the **AllLines** array. This is to prevent these lines from being analysed again and skewing results. This algorithm is looped through multiple times in the code in order to smooth the data as much as possible.

4.2 Sensor Testing

The main tests which are carried out for the comparison of the sensors are shown below:

Noise The uniformity of the depth pixels on the sensors (in the case of RGB-D sensors) were compared in order to determine noise.

Table 4.2: Pseudo-code of finding lines which could be the same and merging them. The actual code can be seen in Appendix C

Line	Code
1	**ParallelLinesDistance** = 30
2	**OffsetGradient** = 1
3	**RadiusToSearch** = 50
4	**AllLines** = *outputOfHoughTransform*
5	**TotalLin**es = *length*(**AllLines**)
6	*while* ***x*** *<* ***TotalLines***
7	**Line1** = **AllLines**[x]
8	**LinesToMerge** = []
9	*for (****y*** *=* ***x****+1;* ***y*** *<* ***TotalLines****;* ***y++****)*
10	**Line2** = **AllLines**[y]
11	**Result** = *proximityCheck*(**Line1**, **Line2**)
12	*if (****Result*** *== TRUE) // The lines are considered close enough*
13	*Add this line to array* ***LinesToMerge***
14	*If there are lines in* ***LinesToMerge***
15	*Average all the lines in this array to produce an estimation of the best fit*
16	*Remove all these lines from* ***AllLines*** *so they are not used again*
17	*Add the averaged Line to* ***AllLines*** *at the current index* ***x***

Accuracy The accuracy of the sensors were tested by comparing the distance reported by the sensor to an actual distance. This was done with and without noise, using the information from the noise test to remove noisy pixels.

Resolution The ability of the sensor to detect small differences in distance between objects of similar distances was tested.

3D Room Modelling Acquisition 3D room scans were taken with both sensors and the quality of the point clouds were compared by eye.

To carry out the sensor tests above, the data for the tests had to be acquired and processed to obtain useful information. For testing accuracy and noise a plain blank wall is all that was needed. However, for testing resolution a scene was created. The relative distances in this scene were recorded for testing. For the 3D room modelling capability the 3D room model was acquired using RTAB-Map and a stand-alone PC.

To obtain the raw data from the sensors for accuracy, noise, and resolution test a piece of software for recording the data was written. A script for Octave, which is a numerical processing language, was

created for processing the raw data. This section describes the data acquisition and data processing.

4.2.1 Data Acquisition

The tests for accuracy and resolution, as mentioned in Section 3.4, required a scene to be set up and a piece of software to capture the information. The software makes used of libraries for each sensor in order to access the raw data streams and initialise the sensors. The block-diagram for data acquisition can be seen in Figure 4.7.

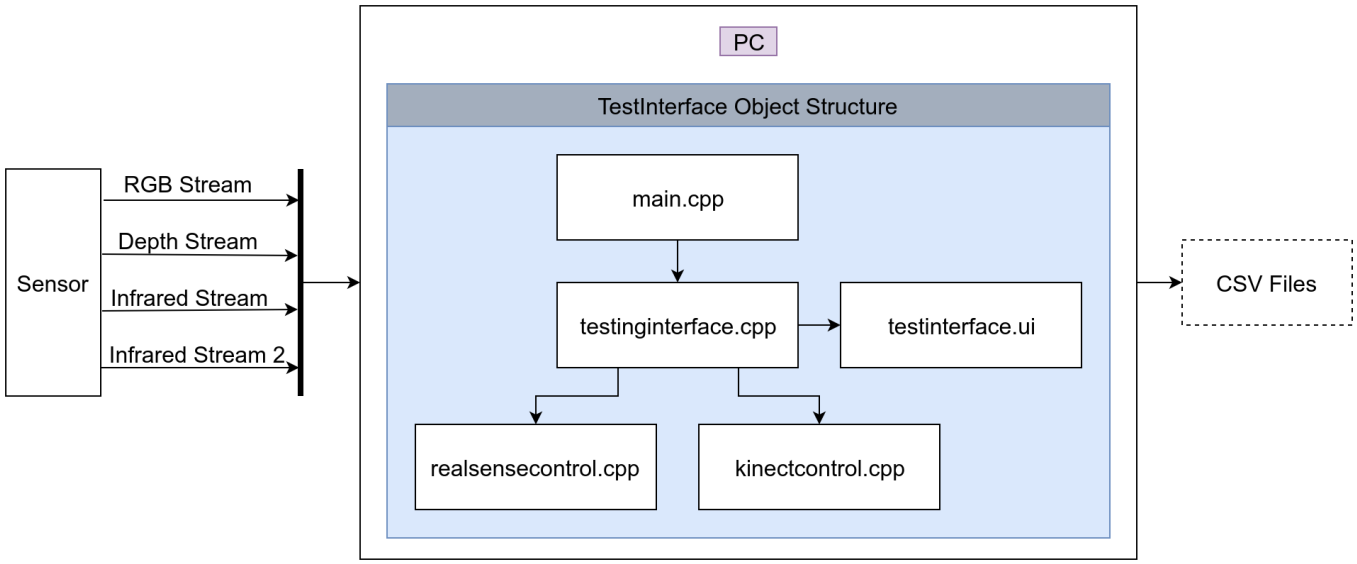


Figure 4.7: Block diagram for sensor testing design.

The system shown in Figure 4.7 is capable of taking in the various data streams shown in the diagram, from a sensor. The system was designed to work with the RealSense and Kinect only, but other sensor controller classes can be created and added. The sensor classes follow the same structure as each other, which allows the **testinginterface.cpp** to perform the same method calls for retrieving information from the sensors. The system is designed to output CSV files of the raw depth information that it is receiving. The software was designed to create the test data needed for the completion of the tests in Section 3.4. The user interface for the software is shown in Figure 4.8. The testing interface provides information using labels which are necessary during debugging but are not necessary for the use of the software and are not explained here. However, how to capture results with the software is explained.

With reference to Figure 4.8, the camera which is to be used was selected from the left, and "Start Camera" was clicked to begin using the sensor. When finished with the software the "Stop Camera" button was clicked and the software closed. Once the sensor had been initialised it was ready for data streaming. The various streams were transmitted to the interface in a separate thread. The images from the different streams can be saved by selecting the streams with the checkboxes. For the tests,

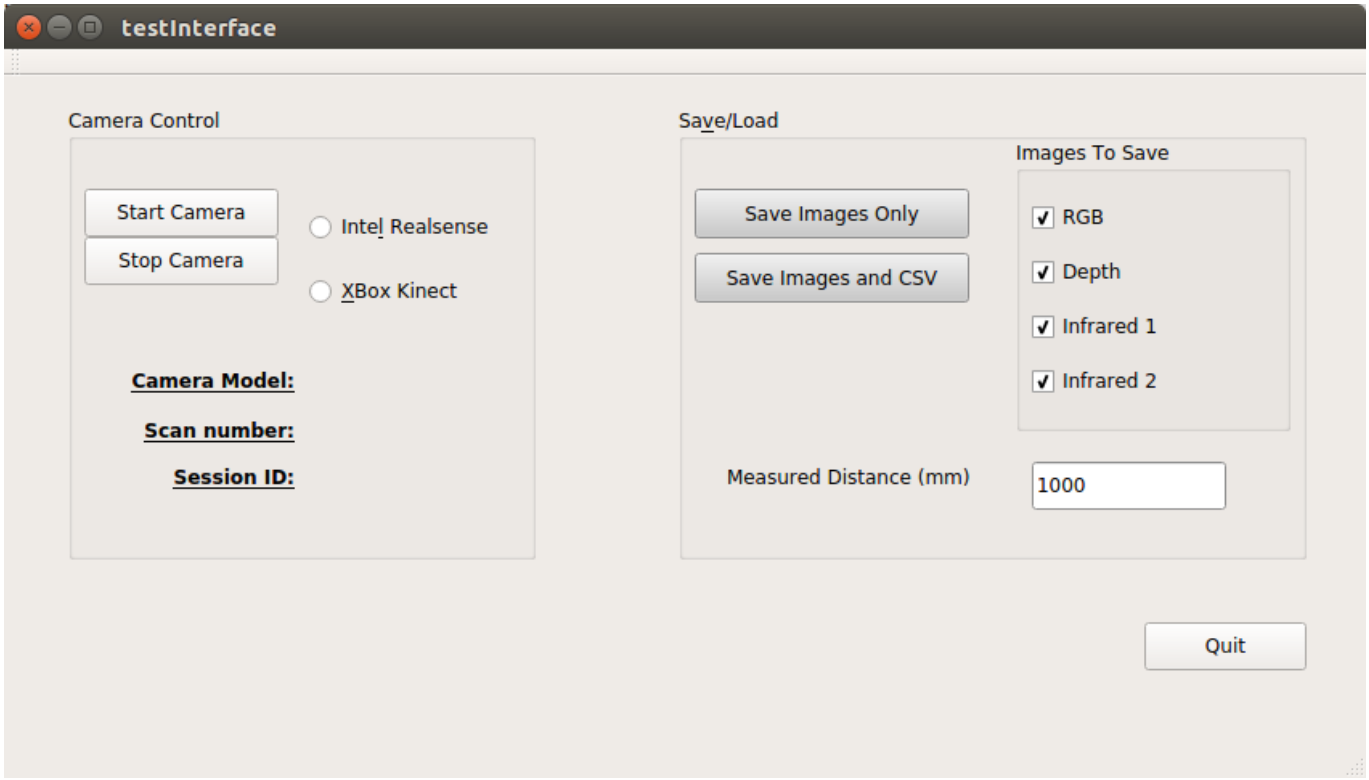


Figure 4.8: Testing software user interface. The testing interface software can be found online[1].

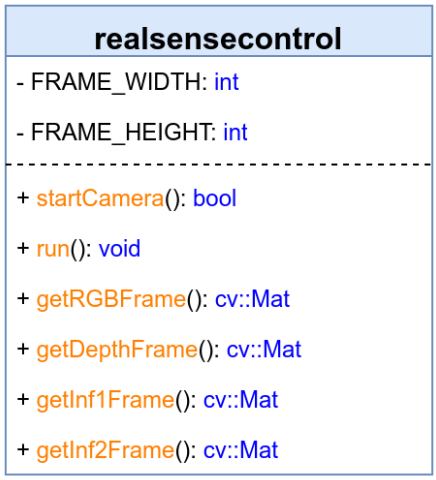
only CSV files were required and no image streams were selected.

For the accuracy and resolution tests the recorded distance from the sensor to the wall was inputted into the textbox labelled "Measured Distance (mm)". The "Save Images and CSV" button was pressed causing the software to record 4 depth frames a second from the sensor, over a 15 second interval, resulting in a total of 60 frames. Each of these frames were saved as an individual CSV file.

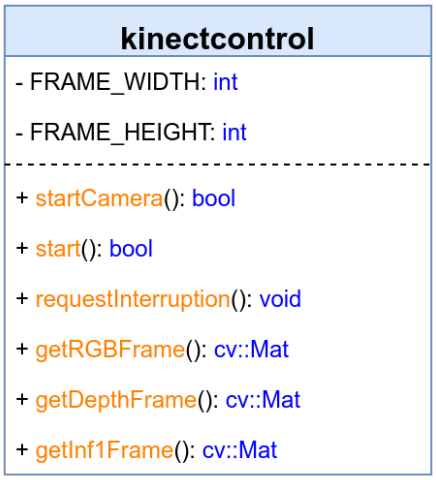
The frames that were captured were done so when the camera is stationary, therefore each frame will be identical to the next except for noise. Therefore, comparing the frames will allow the extraction of the noise from one frame to the next by finding the range of fluctuation of each pixel. By averaging each pixels value it will allow for the accuracy of the sensor to be found, with noise being lost in the averaging. For a broader overview, the classes of the modules are shown in Figure 4.9.

The camera control classes were designed so that their function calls are similar from within the "testinterface". Both camera control classes run threads in order to constantly receive the data streams from the respective cameras, however, their implementation is slightly different. The "realsensecontrol" class has a function to get a second infrared frame, this is because the RealSense camera has two infrared cameras, whereas the Kinect only has one.

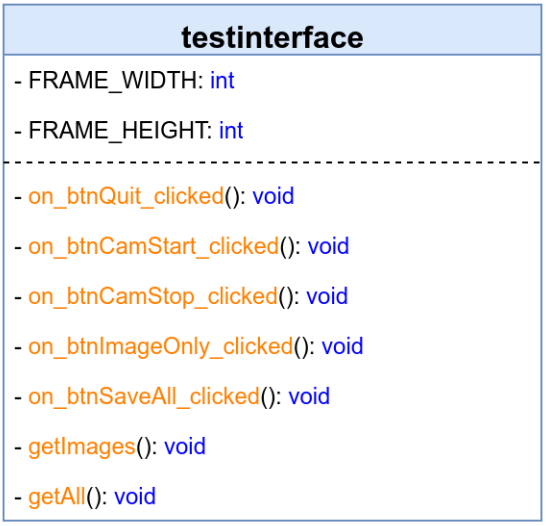
[1] `https://github.com/bkwilsonZA/Masters/tree/master/TestingInterface`



(a) RealSense camera controller class. The class was written for the testing interface.



(b) Kinect camera controller class. The class was written for the testing interface.



(c) Testing interface class.

Figure 4.9: Class related to the testing interface.

The four categories for determining which sensor is the best are: accuracy, noise, resolution, and 3D room modelling ability. The same acquisition software is used for accuracy, noise, and resolution tests. The 3D room modelling test is done by eye with other criteria (Section 3.4) and makes use of RTAB-MAP (Section 2.3.3). The description of the tests and their environments are as follows:

Accuracy and Noise - Acquisition and Test Design

The data captured for the accuracy at a certain distance can also be used to determine the noise at that distance. Noise for the sensor was determined by how much each individual pixel fluctuates between the 60 frames. The sensor being tested was moved to various distances perpendicular to a wall with no

obstructions or features. Each time it was moved a data capture occurred and the actual distance was noted. The recorded CSV files was used to calculate the accuracy of the sensor using data processing techniques in Octave. The physical environment of this setup made use of the following:

- The sensor, placed on a tripod at a fixed height.
- A wall which had no features on it as this is the worst case.
- A tape measure to measure the distance between the wall and the sensor.

Distance Resolution - Acquisition and Test Design

For the resolution the same wall was used but objects were placed in the environment. A drawing of the scene that was used can be seen in Figure 4.10. The scene was made up various objects. The measurements of the scene relative to the wall was taken before scans commenced. The physical environment of this setup made use of the following:

- The sensor, placed on a tripod at a fixed height.
- Two paintings, of which their canvases are different thicknesses resulting in them being slightly different distances from the sensor.
- A chair at an angle.
- A box on the chair.
- Two slightly offset round candles.
- A tape measure to measure the distance between the wall and the sensor.

3D Modelling Ability - Acquisition and Test Design

To test the 3D modelling ability 3D scans were taken with both sensors using RTAB-Map. How the cloud is captured has been discussed in Section 4.1.1, along with the use of Figure 4.2. Due to RTAB-Map being an out-of-the-box package, designed to work with depth data (not necessarily a specific sensor), it will capture relatively unbiased data from each sensor. Due to it also not having been designed for a specific sensor, if the sensor has a lot of noise, or produces bad information for room modelling, the software will struggle to map a room. Therefore, the following things that will be checked for the 3D room modelling test are:

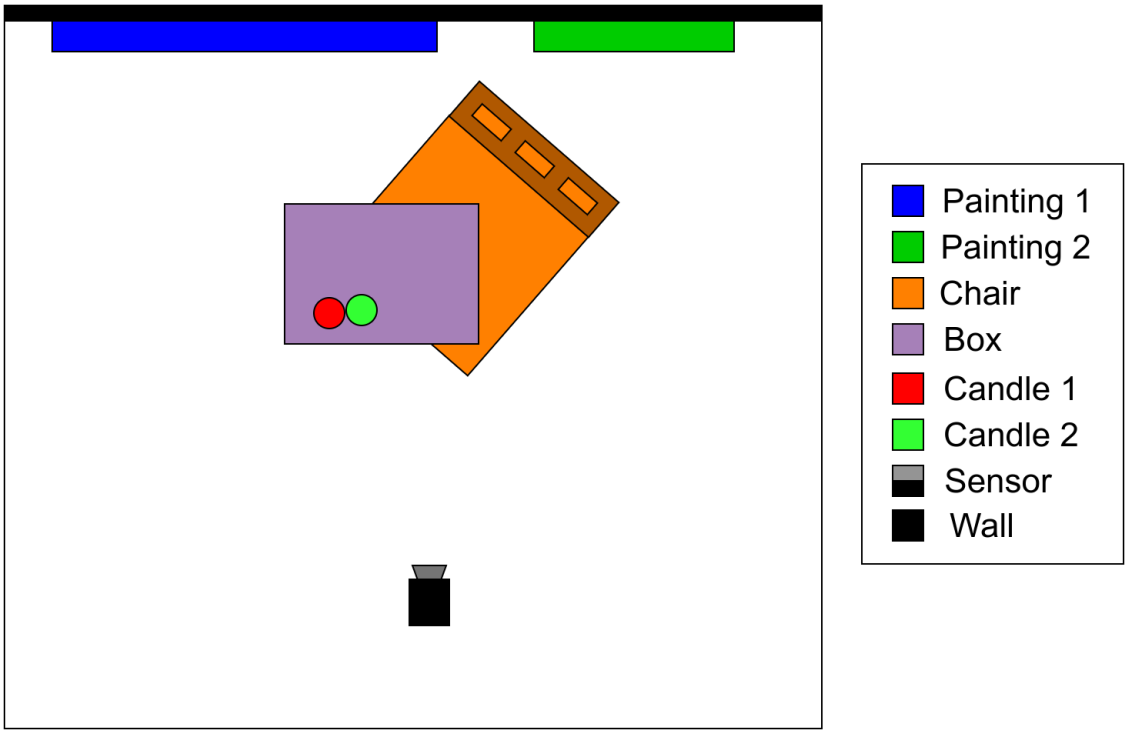


Figure 4.10: Drawing of the scene which will be examined for resolution testing.

1. How well the system can scan an environment:
   (a) Did the sensor get lost in the environment?
   (b) Was the sensor able to map a room out-of-the-box?
2. How good the resultant point cloud looks:
   (a) By eye, does the cloud look like the room it mapped?
   (b) Is the noise in the point cloud noticeable?

These simple questions helped in determining whether the sensors were good for 3D modelling. The reason for performing this test as a final check is due to the fact the sensor might pass the previous tests, but the information it returns might not be as good in practice. The other tests focus on the quality of the data itself, which will be important for the final 3D modelled environment, but the results of the previous tests cannot definitively determine if the sensor can perform 3D room modelling adequately.

Both sets of testing, determining the quality of the data received and the 3D room modelling ability, must be done together. If the sensor excels in 3D room modelling, but the received data is incorrect and always offset by approximately 15 % it will not be worthwhile using the sensor. The same stands for if the sensor produces excellent results for the noise, accuracy, and distance resolution tests, but is unable to work in practice. The reasons for not working in practice are mainly attributed to information loss

(inability to detect distance or produce enough information for SLAM), resulting in the system getting lost whilst mapping an environment.

4.2.2 Data Processing Techniques - Design

The raw data captured by the testing interface was stored as CSV files. Each CSV files represents a single frame captured from the sensor. It is a 2D grid that represents the pixel values of the depth sensor with the amount of rows and columns being equal to the amount of pixels for the height and width of the sensor, respectively. This raw data was imported into Octave and processed to obtain useful information.

To perform the tests required in Section 4.2 the raw data was be processed and results drawn from the outputs. Each distance that was recorded for the sensor will contain 60 frames, which was explained in Section 4.2.1. A diagram depicting how the data was processed for accuracy and noise/range tests can be seen in Figure 4.11.

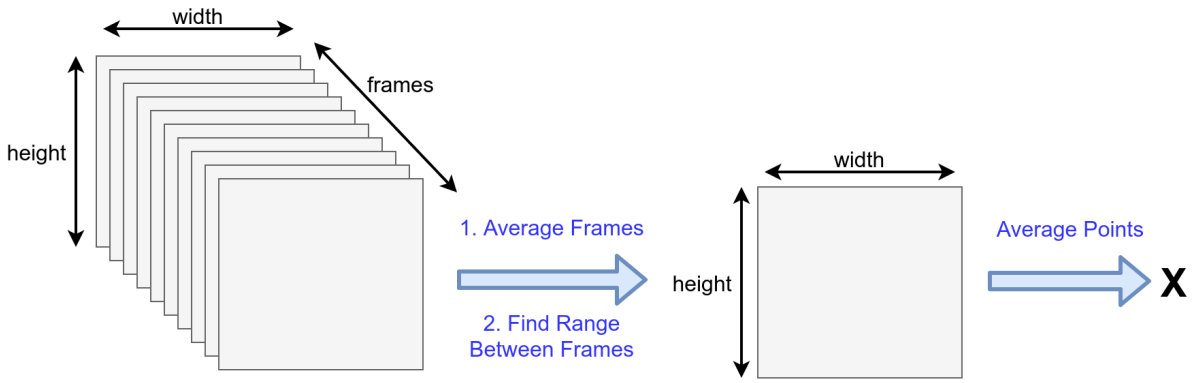
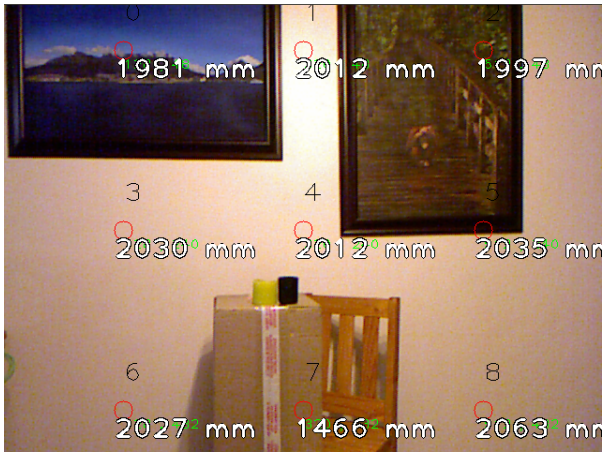


Figure 4.11: An overview of how the raw camera data will be processed in Octave. Each frame is a matrix of points. Due to the similarities between the Average Distance processing and the Range processing, they have been included in the same figure, but in reality output two different matrices as well as different single average variables.

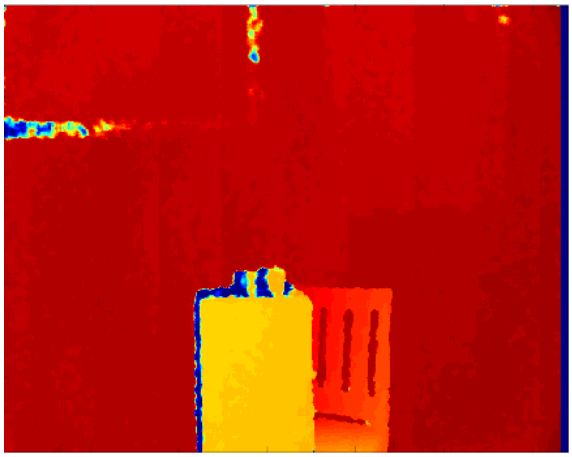
Figure 4.11 imported the different frames for a certain distance into a 3 dimensional matrix. This was so that each frame can be accessed individually. The distance of each pixel was averaged over the 60 frames producing a frame with the average values for each pixel. To determine the noise, the range of fluctuation in distance that each pixel goes through is found. If the range of a pixel across the 60 frames is greater than 25 mm it is considered noise. All points in these matrices were then averaged to produce a single average value for the accuracy and range. The script which was used for this can be found in Appendix D.

For determining resolution accuracy the script also averaged the distance for each pixel over the 60

frames in order to find the average distance and range for each pixel. To determine the resolution accuracy the scene in Figure 4.10 will have to be examined manually. This is done by clicking on the object, in the plot of the depth frame, whose distance must be found. It is not too difficult to determine from the plot of the depth frame the object being tested, this can be seen in Figure 4.12. The expected value was compared to the recorded value and this offset determined the sensors ability for resolution accuracy of multiple objects in a frame.



(a) Picture of the scene as taken by the Kinect.



(b) Heat Map of the depth frame of the Kinect.

Figure 4.12: Comparison of RGB frame and Depth frame from an RGB-D sensor (Microsoft Kinect at 1996 mm from the wall).

4.3 Integration

The integration of each system's sub-systems is explained in this section in more detail. The different systems themselves do not directly integrate with each other. The only form of interaction between the systems is that the sensor testing system is responsible for the sensor choice in the floor plan creation system.

4.3.1 Floor Plan Creation

The floor plan creation system, making use of two pieces of software, will save various interim files and output files in certain directories in order for the two pieces of software to share their information. This can been in Figure 4.13.

Figure 4.13 explains how the systems data flow and integration works. The **FloorPlanCreator.exe** is called with **lounge.ply** as its input. During processing the system outputs the point clouds it is creating

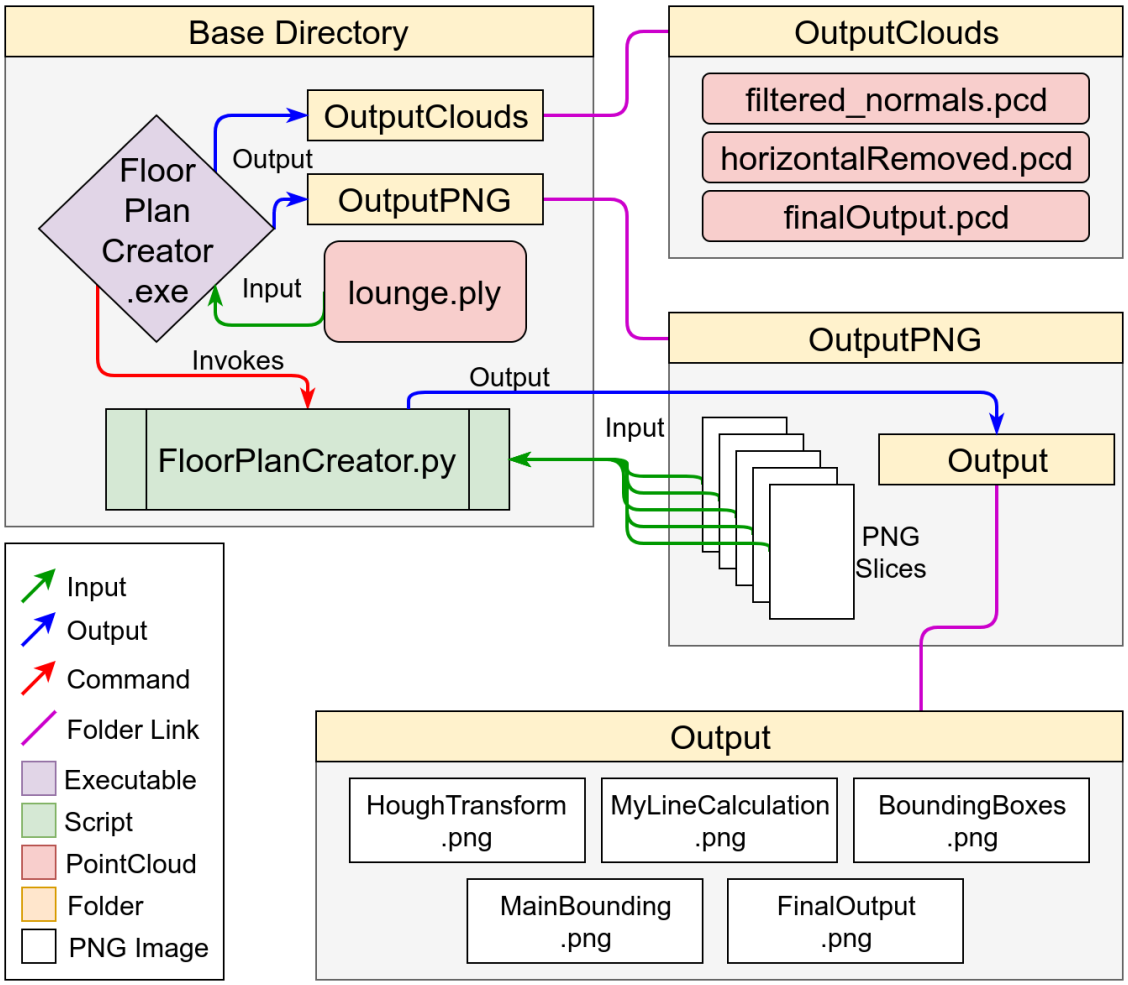


Figure 4.13: An overview of how the two pieces of floor plan creation software integrate with each other even though they are completely separate.

for debugging purpose into the **OutputClouds** folder. The PNG slices which are used for the 2D processing are outputted into the **OutputPNG** folder. Once this is completed the **FloorPlanCreator.py** is called from within the **FloorPlanCreator.exe** program.

The **OutputPNG** folder is where the **FloorPlanCreator.py** script will look for PNG slices. The script will use the PNG slices as input and output PNG images throughout its processing, used for debugging, into the **Output** folder within the **OutputPNG** folder. The main PNG image which represents the final output of the system is names **FinalOutput.png**.

Therefore, the system will take in a point cloud within the base directory, and output the final result in the folder ***OutputPNG/Output/*** relative to the base directory.

4.3.2 Sensor Testing

The integration for the sensor testing software is not as intricate as that for the floor plan creation. The integration of the different classes in the sensor testing software was outline in Figure 4.8. This integration explains the integration between the output of the sensor testing software and the script used to examine the results. This integration can be seen in Figure 4.14.

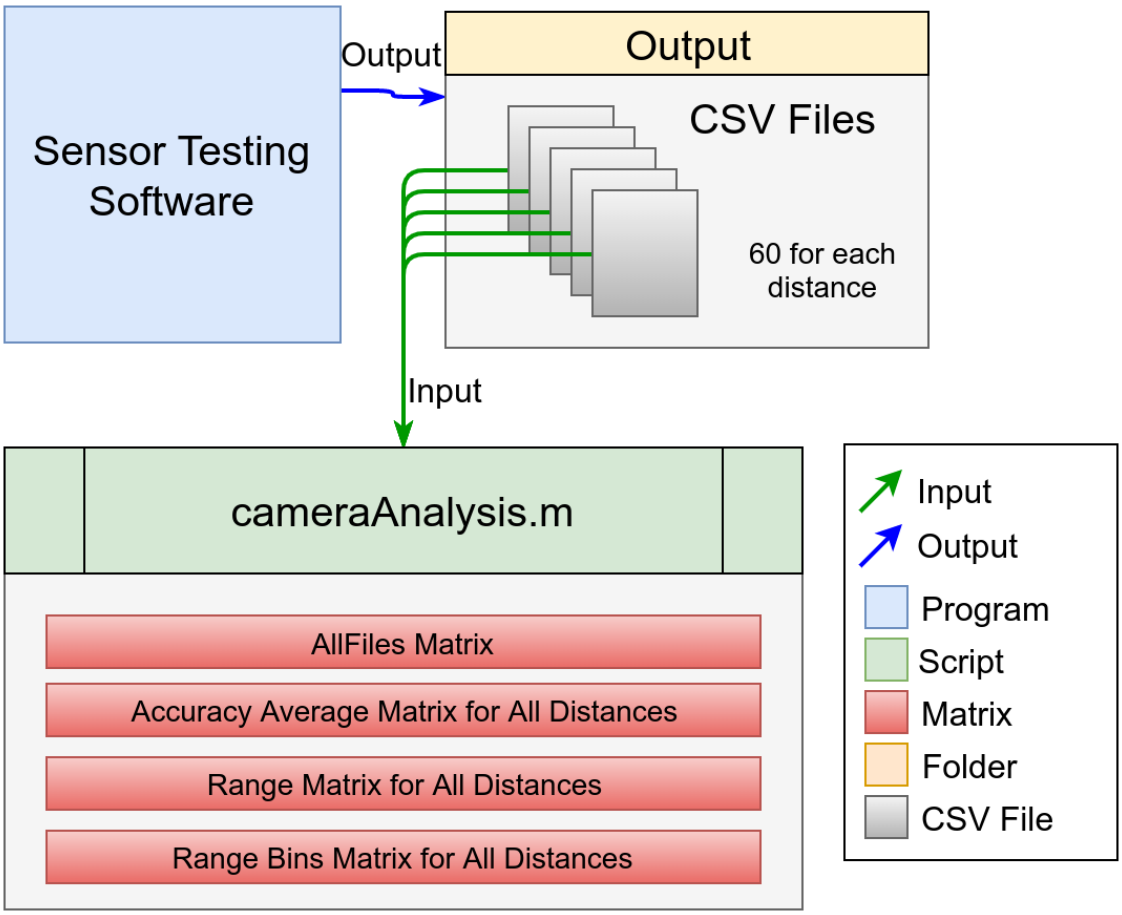


Figure 4.14: An overview of how the sensor testing software's outputs are imported into the Octave scripts, and the data they produce.

Looking at Figure 4.14, the sensor testing software as previously mentioned will capture 60 frames per scan. These frames are saved as CSV files and saved to a directory. The CSV files of multiple distance can be saved in the same directory, as the file names of the CSV files indicate which distance they were recorded at.

The Octave script reads this distance from the file name when importing in order to group all 60 frames together for a specific distance. For each distance it averages the 60 frames, producing an averaged matrix for the distance, which is used to determine the accuracy of the sensors depth finding.

The range of fluctuation of each pixel for the 60 frames were found and saved as a matrix. The pixels

in the range matrix were also sorted into different range bins, representing how many pixels fluctuated by a certain amount of millimetres during the 60 frames. This means that the total amount of pixels in the range matrix that fluctuated by '0 mm' were totalled, by '1 mm', etc. The bins into which they were sorted were: 0, 1, 2, 3, 4, 5, 6-10, 11-15, 16-25, 25+. This gives us information on the precision of the sensors, not just their accuracy. This is better visualised and understood with Tables 5.3 and 5.4 when discussed in Chapter 5.

Chapter 5

Results and Discussions

The results of the floor plan creation system are presented first, followed by those for the sensor testing system and sensor choice tests. The results will be presented in a similar order to that of the requirements, functionality, and tests in order to easily link them to the test matrix in Table 3.2. Discussions on the results will occur in order to explain results clearly and what they mean, if not self evident from the results.

5.1 Floor Plan Creation

The point cloud in Figure 5.1 was used to obtain the results presented here. The point cloud of an apartment's lounge and kitchen were captured using the Microsoft Kinect. The environment was chosen for multiple reasons, such as the strong presence of clutter within the environment as well as it being a confined area, making obstructions (demonstrated in Figure 2.17) more common; therefore, making this the ideal scene for creating and testing the algorithms. The below sections follow a similar structure to Chapter 4.

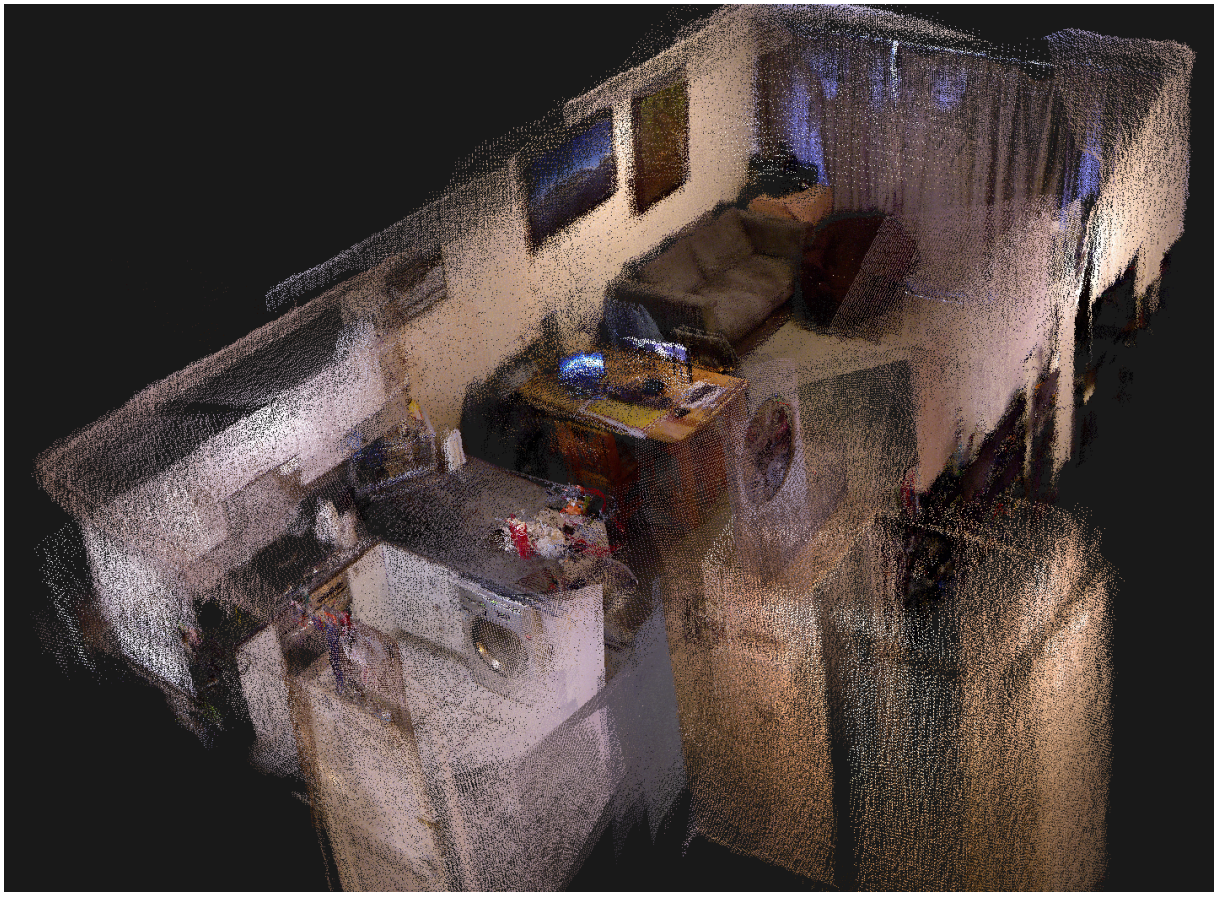


Figure 5.1: A point cloud of the room used for system development. Taken with the Microsoft Kinect and RTAB-Map (Section 2.3.3).

3D Processing

The 3D processing structure can be seen in Figure 4.4. For pre-processing the image is first down-sampled. Due to the size and number of points of the point cloud the effects of down-sampling will speed up the performance of subsequent processes. The surface normals are calculated for the point cloud and each point which has a normal facing upwards or downwards is removed, therefore removing horizontal planes. The output of this pre-processing can be seen in Figure 5.2.

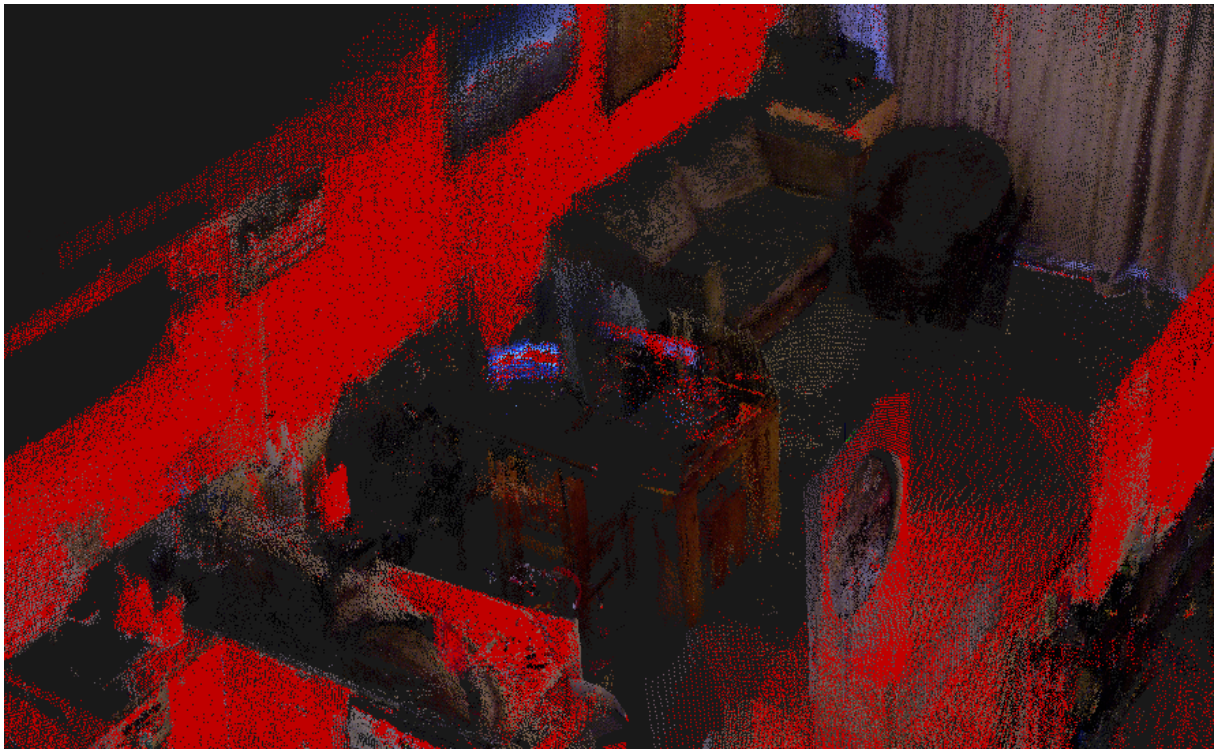


Figure 5.2: The output of the system after pre-processing has occurred

It can be seen from Figure 5.2 that the table and counter top have been removed, as well as the top of the couch cushions. The red dots represent vertical points to aid in the distinction that horizontal points were removed. Points with colour simply have surface normals which are not horizontal nor vertical. In order to determine if these points belong on planes the point cloud is processed using the planar segmentation library, as mentioned in Section 2.1.3. Using this library the system was able to remove all points that were not on vertical surfaces. The output is shown in Figure 5.3.

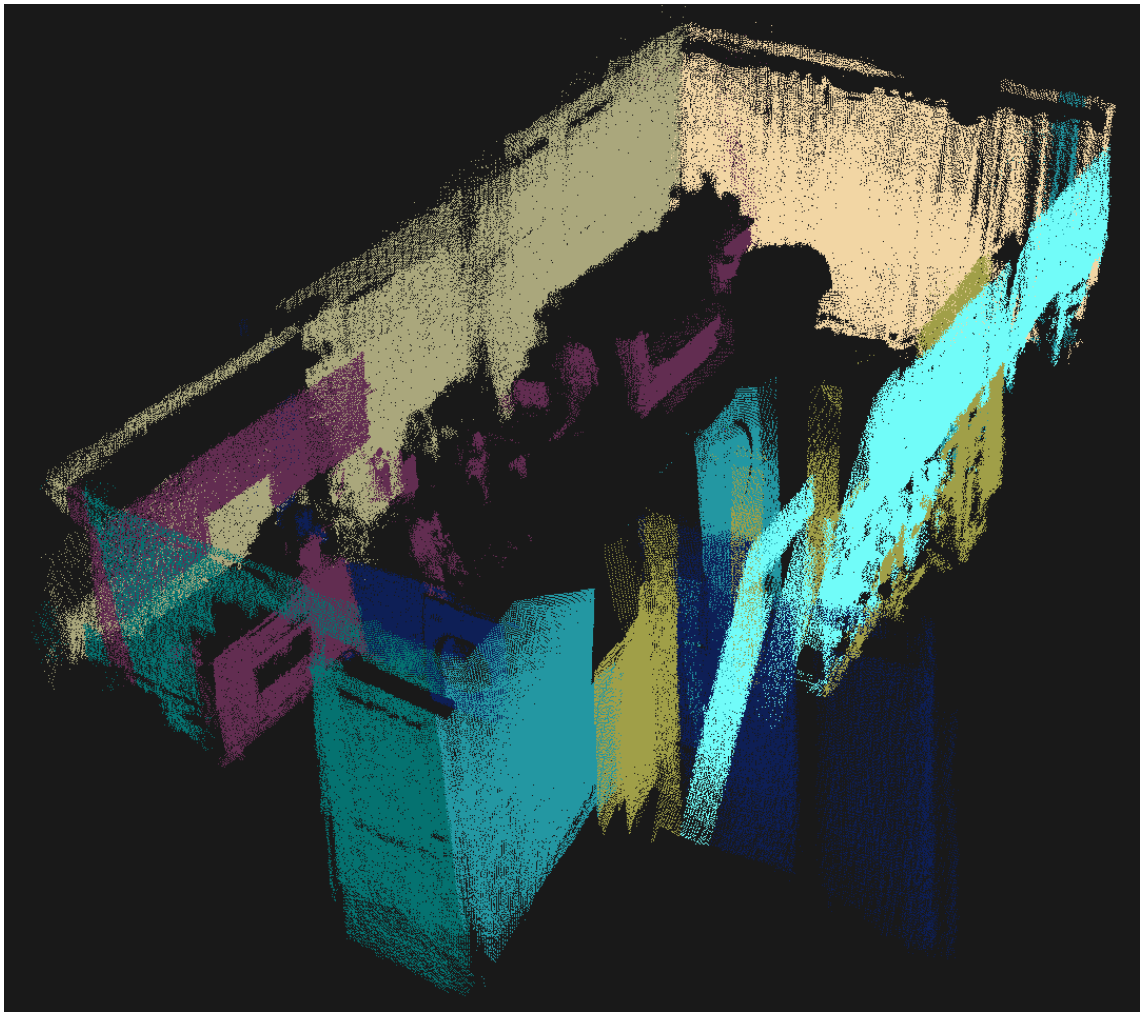
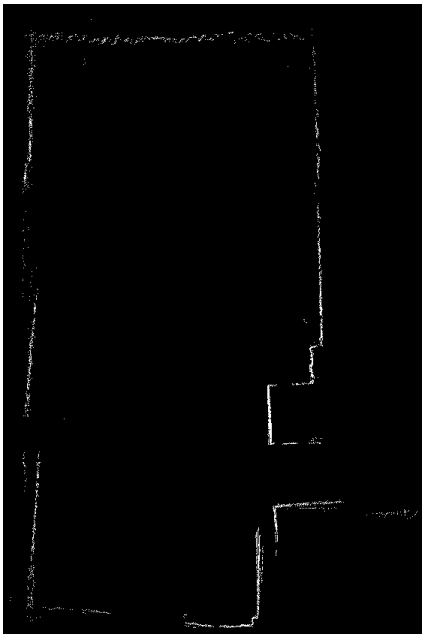
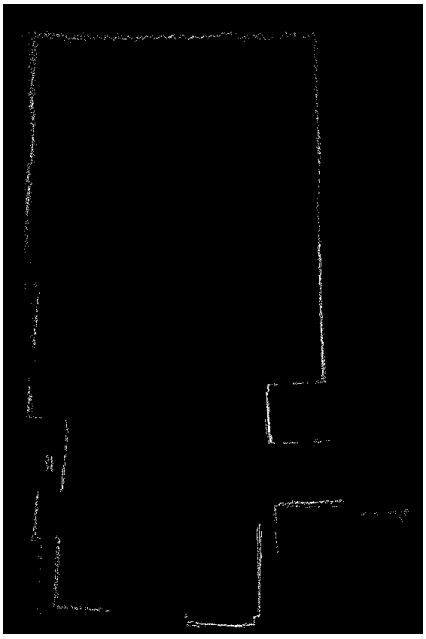


Figure 5.3: The output of the system after finding vertical planes.

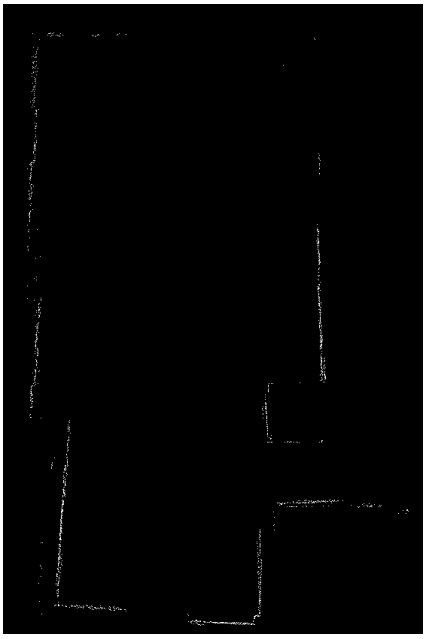
Figure 5.3 still contains some unwanted points, but there are fewer unnecessary points within the figure compared to Figure 5.1. The next step was to extract horizontal slices from the environment. Most of the clutter is from objects which are placed on the ground such as couches, tables, chairs, etc. Therefore, it is useful to only extract information from the top half of the point cloud. The purple plane on the left of Figure 5.3, represents in the real-world: (from left to right) cupboards, an oven, a microwave, sides of a table, sides of a couch, along with some other points. It is obvious by eye that only taking the top half of the point cloud will minimize error introduced by these points, as well as any other planes from counters or shelves. The assumption made about the rooms being scanned is that the walls will be constant from the ground to the ceiling. Therefore, by only taking the top half of the point cloud it will not cause any loss of useful information. A few horizontal slices which have been extracted can be seen in Figure 5.4.



(a) PNG representing 6/10 of the total height



(b) PNG representing 7/10 of the total height



(c) PNG representing 8/10 of the total height

Figure 5.4: PNG images representing different horizontal slices.

5.1.1 2D Processing

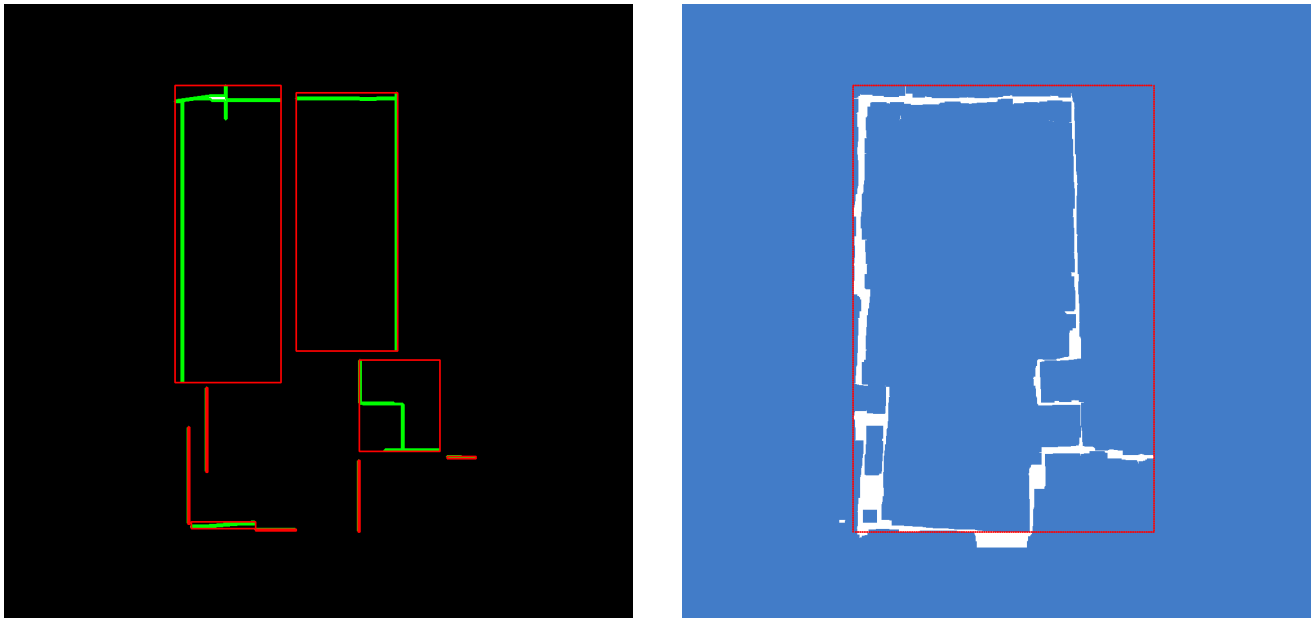
The current setup will result in 5 PNG images being imported into a Python script. The Python script follows the block diagram shown in Figure 4.5. The script imports the images and averages the pixel data of each image producing an averaged image. This image undergoes erosion and dilation, explained in Section 2.2.1, in order to join the dots. The potential lines from the image are found using the Hough Transform seen in Figure 5.5(a).

(a) Hough Transform of averaged image. The red lines are estimated lines from the transform.

(b) Walls having undergone smoothing technique.

Figure 5.5: Steps showing finding walls on the imageDetermining walls on the image.

The red lines in Figure 5.5(a) are the output of the Hough Transform which attempts to find lines in the image. The lines were smoothed using an algorithm created for this project, explained with pseudo-code in Table 4.2. The algorithm works by examining each line and determining if it is meant to be joined on to a nearby line or not. The output of this algorithm can be seen in Figure 5.5(b). These identified lines then undergo OpenCV's function for finding contours. This algorithm determines if these lines are on the same contour (touching) or not, and classifies them into a list of available contours. The bounding boxes for the contours were found and can be seen in Figure 5.6(a).



(a) Bounding box for each contour.

(b) Bounding box for entire cloud.

Figure 5.6: The bounding boxes of the point cloud which are used for displaying dimensions.

The bounding box for each contour and the entire point cloud is shown in Figures 5.6(a) and 5.6(b). The outer bounding box of the point cloud is used to show overall dimensions, and the dimensions of each contour is then displayed outside of this bounding box to avoid overlaying information, making it unreadable. The final output produced by the script can be seen in Figure 5.7.

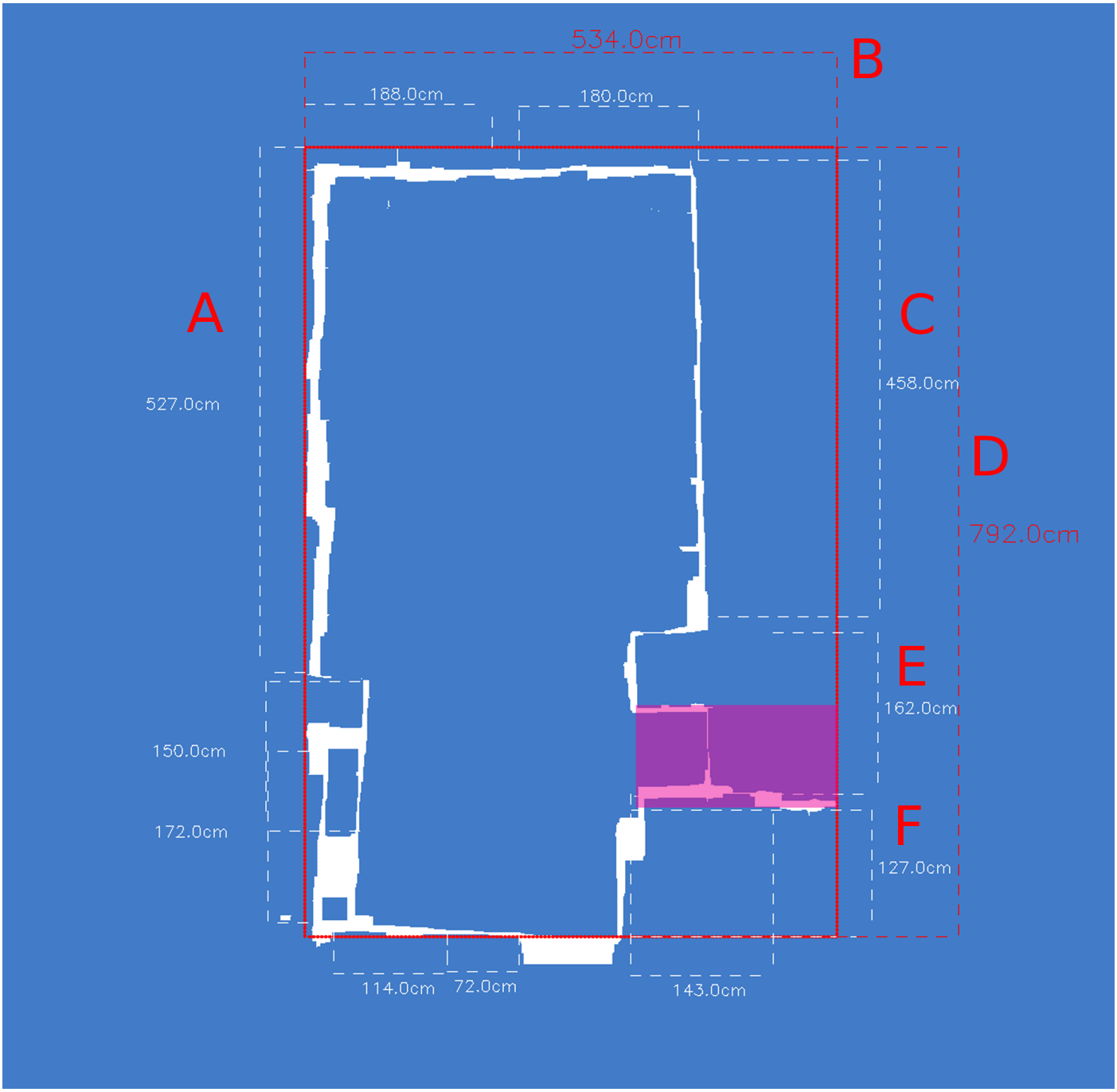


Figure 5.7: The outputted floor plan of the system.

Looking at Figure 5.7, it can be seen that the bounding box does extend further up than the room itself, this is due to noise in the system having caused an unwanted contour. This unwanted contour can be seen in Figure 5.6(a) in the top left of the Figure. It is the short, vertical, contour which must have been created from a few points being present in the PNG. Even though noise removal took place in multiple areas throughout this process, noise will always be present in a system. Therefore, the accuracy of this system will still be examined using this output in order to determine the effectiveness of the system. Table 5.1 tabulates the differences in the outputted floor plans dimensions against the

real-world dimensions.

Table 5.1: Comparison of the real-world measurements of a room against the recorded measurements from a sensor.

Line	Real-World Distance (cm)	Recorded Distance (cm)	Offset
A	488	527	7.4%
B	485	527	7.9%
C	460	458	0.4%
D	763	792	5.5%
E	159	162	1.9%
F	131	127	3.1%

The average offset is 4.36%, which is within the allowable offset determined by Test 8 in Table 3.2. Examining Table 5.1 it can be seen that lines A, B, and D are less accurate compared to the others. This is due to a number of factors. For lines A and D, this is attributed to the bounding box not lining up exactly with the room. Due to the unwanted contour described above, the bounding box of the room extends upwards by an unknown amount, which will add on to the measurement obtained for the floor plan. For line B, this offset is due to the accuracy of the camera measuring the region coloured in purple. During capture, the sensor moved down the middle of the apartment, rotating whilst moving to see all the walls.

The purple region was simply captured as extra information and was not intended to be part of the main model, but was left in for added points and noise. As it was not actively scanned the sensor did not adequately capture data within this region, and the region remained further from the sensor than the other points of the apartment. This is similar to an example of detecting information through a window. Therefore, it is a mixture of inadequately scanned data as well as accuracy deterioration with distance that resulted in this offset of the width of the apartment.

The final functionality of the floor plan creation system was making the system take in a point cloud, and output a floor plan, without any additional interaction from the user. The user would run a single command launching the application. The process was automated by making the main C++ program, the one responsible for 3D processing, call the Python script upon completion of its tasks.

5.1.2 Test In An Unknown Environment

The system was designed using the scene shown in Figure 5.1, therefore making the results here useful but not deterministic of whether the system can work with an unknown environment. The cloud shown in Figure 5.8 was not used as an input into the system until the system was completed, nor has the system undergone any changes after the processing of this unknown environment in order to improve

results.

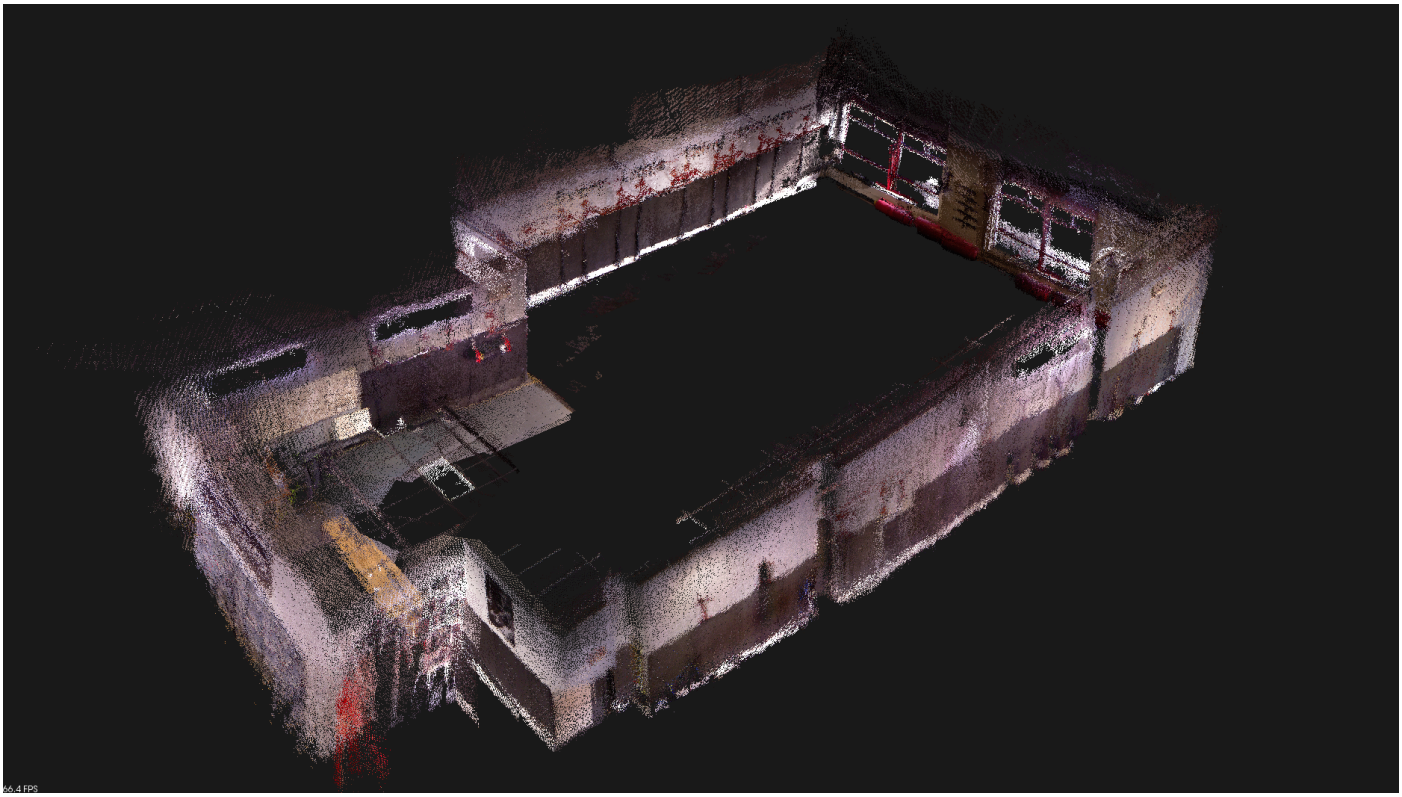


Figure 5.8: The cloud of an unknown environment taken at a training studio using Microsoft's Kinect.

Examining Figure 5.10 it can be seen that there are many different measurements that have been detected thinking they are relevant features, like wall sections. This is due to the system not merging all wall lines from the Hough Transform accurately. Instead of having one continuous contour for each large wall section, there exist many smaller contours representing it, which has led to the many measurements appearing on the floor plan. To test the accuracy for this environment, the bounding box will be compared to the real-world measurements of the studio in question. These results can be seen in Table 5.2.

Table 5.2: Comparison of the real-world measurements of an unknown environment against the recorded measurements, from a sensor.

Side	Real-World Distance (cm)	Recorded Distance (cm)	Offset
Width	1306	1394	6.31%
Length	2283	2209	3.34%

The average offset for the unknown environment is 4.83 % which is still within the 5 % offset deemed necessary to pass the accuracy test. There are multiple reasons as to why this offset is so large (given that it is almost 1 meter in size). The obvious reason is the skewness of the 4 walls in comparison to the bounding box rendered around the studio. Due to the size of the studio, the error accumulated during

the scan may not have been able to be corrected with just loop closure - resulting in not so accurate results.

The wall finding algorithm, in the 2D processing script, could have started with a bad seed point, which may have resulted in incorrect line smoothing. The wall finding algorithm begins by taking a line, comparing it with its neighbours, and any neighbours that are considered to be the same line are merged. During the merging process the gradient changes, therefore, if there is a line which is merged that skews the results just enough, the newly found line will then no longer merge with any other lines, even if they are the same line. Therefore, this line will get its own bounding box and its own measurements.

Whilst examining these results it was discovered that the 2D processing script hadn't accounted for a rotated room. A rotated room in this context refers to one in which the general '4 walls' do not run parallel to the x and y axes on a monitor. The scans which are outputted by RTAB-Map attempt to rotate and display the environment such that one of the walls is parallel to the screen orientation, when viewed from above. This was helpful in outputting the PNG slices which are shown in Figures 5.5 and 5.6 as no manipulation was required to rotate them to this orientation. However, it appears that RTAB-Map struggled to display the environment 'nicely' and therefore, outputted a rotated environment, which the 2D processing script could not handle.

Another possibility for the accuracy error could be due to the height of the studio. The height of the studio that was scanned is higher than a normal room. Therefore, holding the camera will result in it being located lower, relative to the ceiling, than in a normal room with a lower ceiling. This idea is better explained with the aid of Figure 5.9.

With the height of the studio scene being higher than normal, the depth information returned from the higher points will be less accurate than the lower points because the higher points are further away. This assumption is backed up by the results in Section 5.2.2. Therefore, the top of this point cloud will be less accurate than the bottom in terms of distance accuracy, and the PNG slices which are taken from the cloud retain this error.

Although there are errors from the mapping software, the depth information from the higher ceiling, and the 2D processing algorithm not being able to account for a rotated environment, the system was still able to produce a floor plan. Although the offsets were not ideal, they are within an acceptable offset of 5 %.

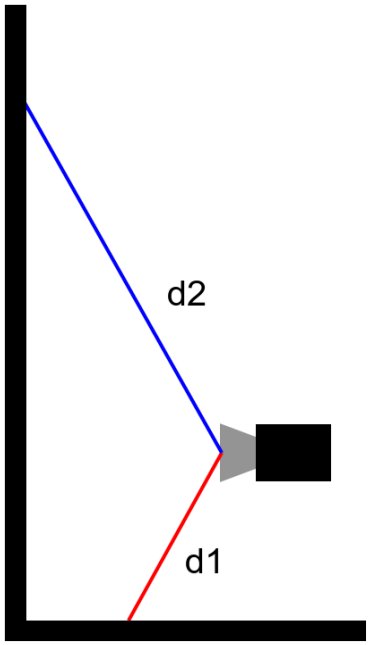


Figure 5.9: Explanation of a theory as to why the distance was not as accurate as hoped for the unknown environment.

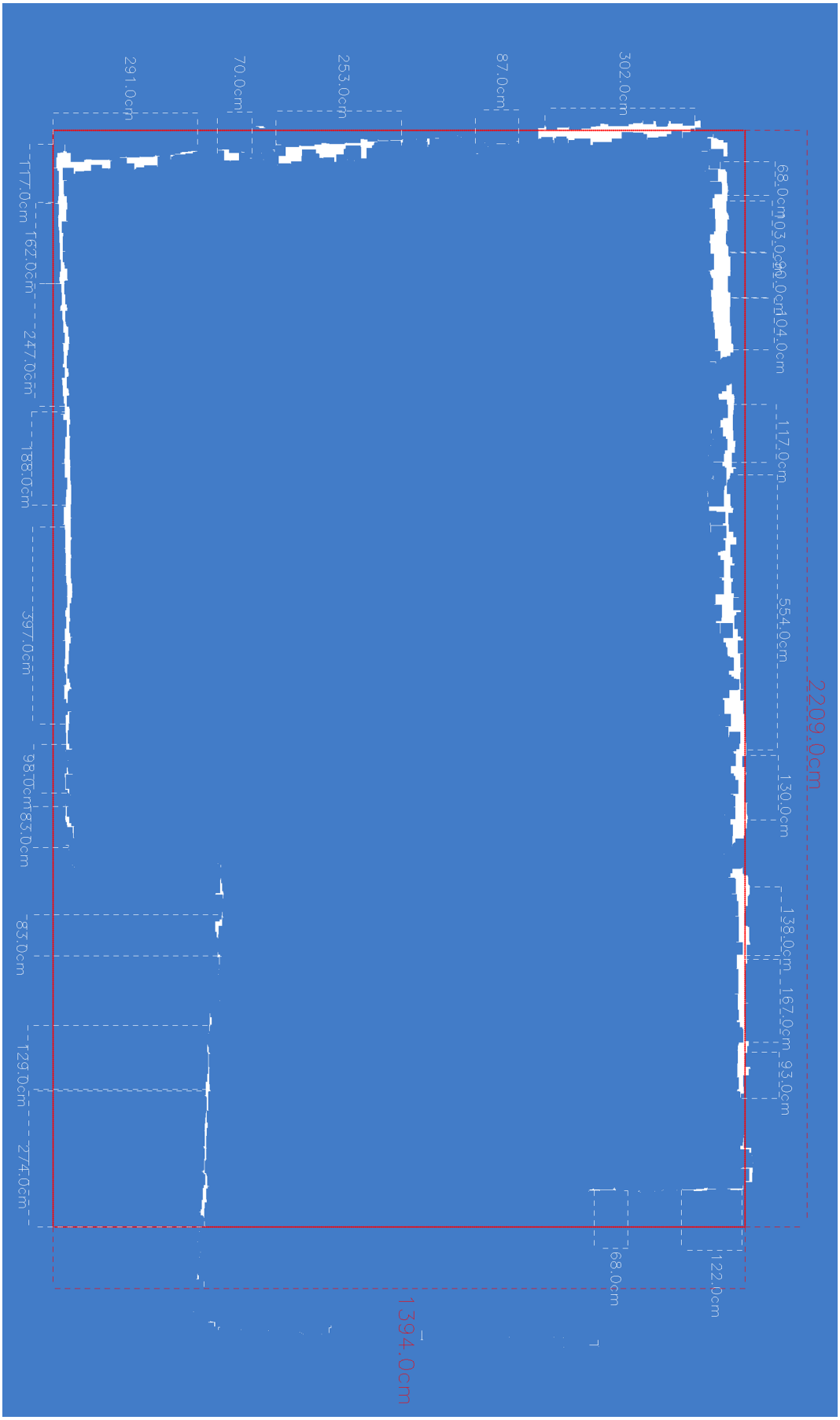


Figure 5.10: The floor plan of the unknown environment.

5.2 Sensor Testing

This sections displays the results obtained from the tests explained in Section 4.2. The results will be presented in four categories: range (noise), accuracy, distance resolution and 3D room modelling capabilities. The results for the Intel RealSense and the Microsoft Kinect will be displayed in each category and discussed.

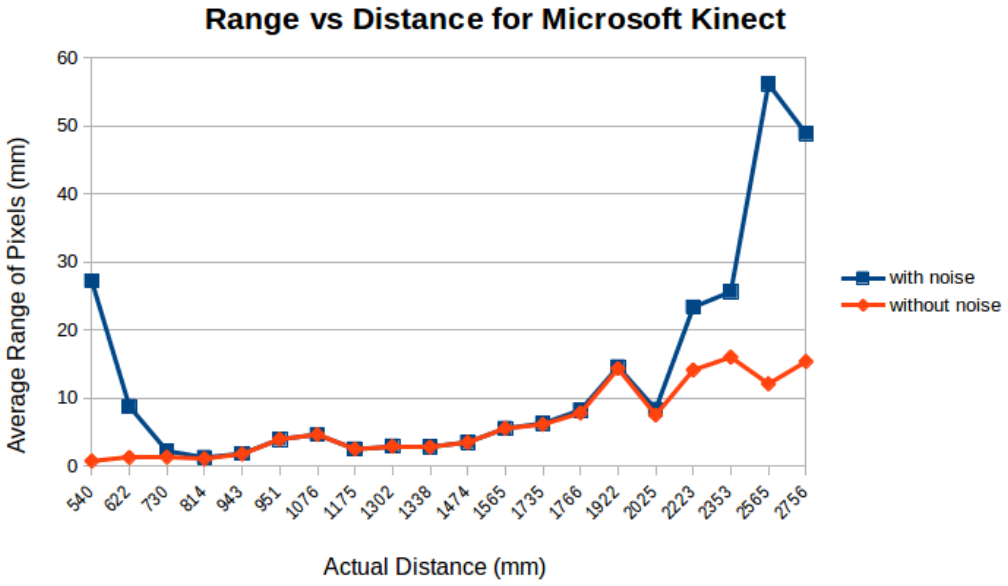
5.2.1 Range

The raw data from each sensor was imported and processed as outlined in Section 4.2. The range of the sensor was found in order to determine how much each pixel fluctuated between subsequent frames. The average range of each sensor at a certain distance is seen below in Figure 5.11.

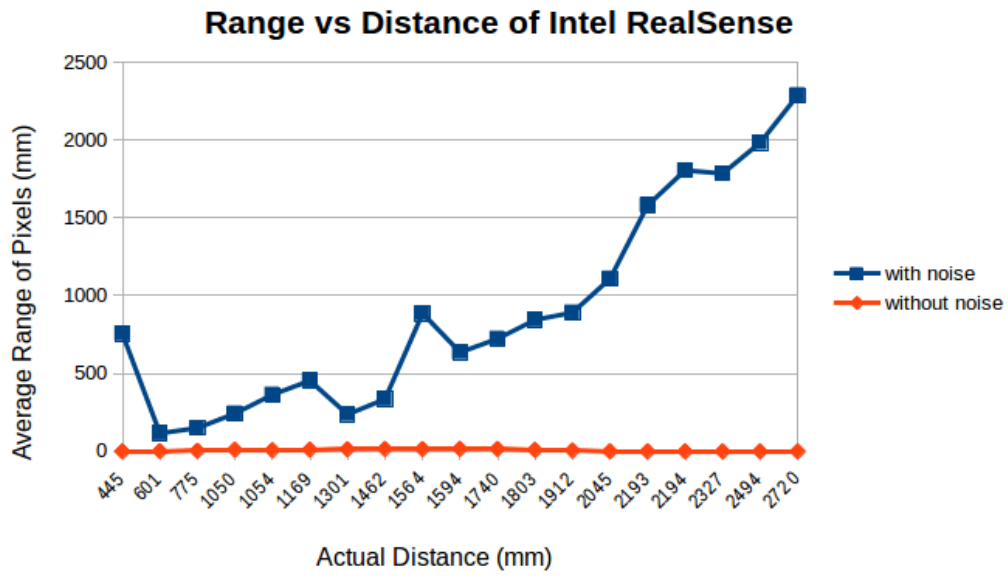
The cause for the discrepancy in Figure 5.11(b) is due to the large amount of noise that is present in the output from the RealSense sensor. The range is calculated by determining the amount of fluctuation of each pixel between 60 frames taken over 15 seconds. If a pixel fluctuates more than 25mm it is considered noise, and the pixel is not used in calculations. Although it may appear that the range improves once noise is removed from the frame, it is not quite the case. By examining Tables 5.3 and 5.4, which represent the range fluctuation of the Intel RealSense and Microsoft Kinect respectively, this can be explained.

Tables 5.3 and 5.4 tabulate the distance the sensors were from the wall at the time of capture versus the percentage of how many pixels fall into a certain range's bin. The total amount of pixels for the RealSense is 172800 pixels (480 x 360) and 307200 pixels (640 x 480) for the Kinect.

Therefore, the percentage of pixels under bin 0 shows how many of the pixels did not change their value over the 15 seconds. The ideal outcome for each sensor is for most of the pixels to end up in the lower bins, meaning a higher accuracy, or for the pixels to be tightly grouped across adjacent bins, as this indicates a uniform range discrepancy, and a higher precision (as explained in Section 2.4.3). If the pixels fluctuate more than 25 mm they are considered to be noise. If a lot of pixels fall under the '26+' bin it is not as bad as them being spread across the bins, as they can simply be removed from the data set when they are in the 26+ bin.



(a) Range plot for Microsoft Kinect with noise and without noise.



(b) Range plot for Intel RealSense with noise. The data without noise is much smaller than with noise, which is why it is not noticeable on the plot.

Figure 5.11: Range vs Distance of the Microsoft Kinect and Intel RealSense.

It was found that when the sensors are unable to calculate the distance for a pixel it sets the value of that pixel to 0. If the sensor is unable to find the distance for a pixel it is generally due to obstructions or other noise in the system. Therefore, the pixel will remain 0 for all frames, calculating a range of 0, and falling into the 0 bin. Therefore, a low range once the noise has been filtered out may not indicate that the remaining points are accurate or useful.

The reason pixels with a distance of 0 were not removed as noise is, firstly, because they do not fluctuate by 25 mm, and secondly, because a 0 distance helps indicate the ability of the sensor to find distance information. Pixels which remained constant were considered accurate pixels - or pixels that should be accurate. The distance accuracy tests and the range (noise) tests took place on a flat wall with no obstructions, therefore, if there are pixels reporting a distance of 0 it means the sensor is struggling to calculate the distance for those pixels. The RealSense was found to struggle on dark surfaces if there are no features on the surface itself, as its infrared scatter pattern gets absorbed on the darker surfaces.

However, in the scene which it was tested the wall was not a dark colour, and there were no obstructions to hamper feature detection. Therefore, the 0 values were indicative of it being unable to find distances for a large amount of its pixels. This became more prominent as the sensor moved further from the wall - meaning the infrared scatter pattern was unable to project bright enough on the wall for the sensor to determine features. As the scatter pattern is light, it will reduce in intensity the further the sensor gets from an object. This indicates the RealSense was not necessarily designed for 3D room modelling, but 3D object modelling, wherein the object would generally be right in front of the sensor during capture.

Table 5.3: Percentage of pixels in various Range Bins of Intel's RealSense at multiple distances.

Distance	**Range Bins - Percentage of Total Pixels (172800)**									
(mm)	**0**	**1**	**2**	**3**	**4**	**5**	**6-10**	**11-15**	**16-25**	**26+**
445	71.06	0.27	0.63	0.41	0.22	0.12	0.13	0.03	0.03	27.10
601	11.98	10.42	27.92	18.93	8.23	2.83	1.25	0.00	0.00	18.44
775	8.82	0.40	4.65	14.70	15.13	12.88	23.52	1.56	0.05	18.29
1050	6.13	0.00	0.00	0.01	0.22	1.08	31.25	28.91	10.41	21.98
1054	7.59	0.00	0.02	0.55	2.59	5.47	36.68	11.18	2.96	32.96
1169	6.03	0.00	0.00	0.00	0.09	0.58	22.01	22.17	11.10	38.02
1301	5.14	0.00	0.00	0.00	0.00	0.08	12.75	33.08	29.02	19.93
1462	4.53	0.00	0.00	0.00	0.00	0.00	2.79	23.17	34.40	35.11
1564	4.28	0.00	0.00	0.00	0.00	0.00	0.03	1.25	12.63	81.82
1594	4.31	0.00	0.00	0.00	0.00	0.00	0.99	9.99	32.56	52.16
1740	4.21	0.00	0.00	0.00	0.00	0.00	0.01	0.45	18.10	77.22
1803	4.25	0.00	0.00	0.00	0.00	0.00	0.00	0.01	3.44	92.30
1912	4.27	0.00	0.00	0.00	0.00	0.00	0.00	0.03	1.73	93.97
2045	4.68	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.37	94.96
2193	5.95	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.01	94.05
2194	5.52	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	94.48
2327	7.70	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	92.30
2494	9.44	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	90.56
2720	13.85	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	86.15

It is evident from looking at Table 5.3 that the RealSense has a maximum range of approximately 2 meters. The pixel ranges for the Intel RealSense are mainly split between bin 0 and bin 26+. The data in Table 5.4 is mainly spread over 3 to 4 bins per distance, with the 3 to 4 bins progressively

Table 5.4: Percentage of pixels in various Range Bins of the Microsoft Kinect at multiple distances.

Distance	**Range Bins - Percentage of Total Pixels (307200)**									
(mm)	**0**	**1**	**2**	**3**	**4**	**5**	**6-10**	**11-15**	**16-25**	**26+**
540	36.81	46.58	9.44	1.80	0.35	0.09	0.08	0.00	0.00	4.86
622	14.93	52.26	22.13	6.76	1.99	0.55	0.19	0.00	0.00	1.19
730	28.05	23.11	39.21	6.84	2.13	0.48	0.06	0.00	0.00	0.12
814	46.46	1.60	48.77	0.23	2.67	0.02	0.23	0.00	0.00	0.02
943	38.29	0.00	22.93	31.36	0.00	4.92	2.50	0.01	0.00	0.00
951	1.25	0.00	13.49	39.03	0.00	28.62	17.52	0.10	0.00	0.00
1076	2.14	0.00	0.00	35.80	24.85	0.00	37.09	0.13	0.00	0.00
1175	41.48	0.00	0.00	0.00	54.22	0.04	4.12	0.12	0.01	0.00
1302	44.27	0.00	0.00	0.00	0.00	52.72	2.88	0.11	0.01	0.00
1338	48.96	0.00	0.00	0.00	0.00	35.70	13.39	1.82	0.13	0.00
1474	48.71	0.00	0.00	0.00	0.00	0.00	48.60	2.51	0.17	0.01
1565	36.76	0.00	0.00	0.00	0.00	0.00	51.24	11.42	0.55	0.03
1735	38.16	0.00	0.00	0.00	0.00	0.00	54.73	0.00	6.52	0.58
1766	30.98	0.00	0.00	0.00	0.00	0.00	50.95	0.00	16.20	1.86
1922	1.48	0.00	0.00	0.00	0.00	0.00	0.50	65.62	31.48	0.93
2025	38.65	0.00	0.00	0.00	0.00	0.00	0.00	55.73	5.12	0.50
2223	1.25	0.00	0.00	0.00	0.00	0.00	0.01	43.38	2.42	52.95
2353	1.36	0.00	0.00	0.00	0.00	0.00	0.00	0.06	49.43	49.16
2565	29.61	0.00	0.00	0.00	0.00	0.00	5.74	1.85	48.60	14.21
2756	25.63	0.00	0.00	0.00	0.00	0.00	0.26	1.07	55.87	17.17

becoming the larger bins (like a moving window). This goes to show the Microsoft Kinect is definitely more precise than the Intel RealSense. The Kinect is shown to see up to approximately 2.8 meters here with most points falling in the 16 to 25 mm bin. This shows that the pixels in the image could be off by up to 25 mm at a distance of 2756 mm. At this distance, the offset is approximately $\frac{25}{2756} = 0.91\%$ which is well within acceptable limits. The average fluctuation range of the distance the pixels detected for each of the sensors are shown in Table 5.5. The average range for the RealSense is only averaged up to approximately 2 meters as this is how far the sensor is able to detect, averaging further would be unfair and would skew the results below.

Table 5.5: The average range in distance detected in mm, per pixel, over 60 frames, for the Intel RealSense and Microsoft Kinect.

	Intel RealSense	**Microsoft Kinect**
Average Range in Distance (mm) - With Noise	510.83	12.19
Average Range in Distance (mm) - Without Noise	9.76	6.58

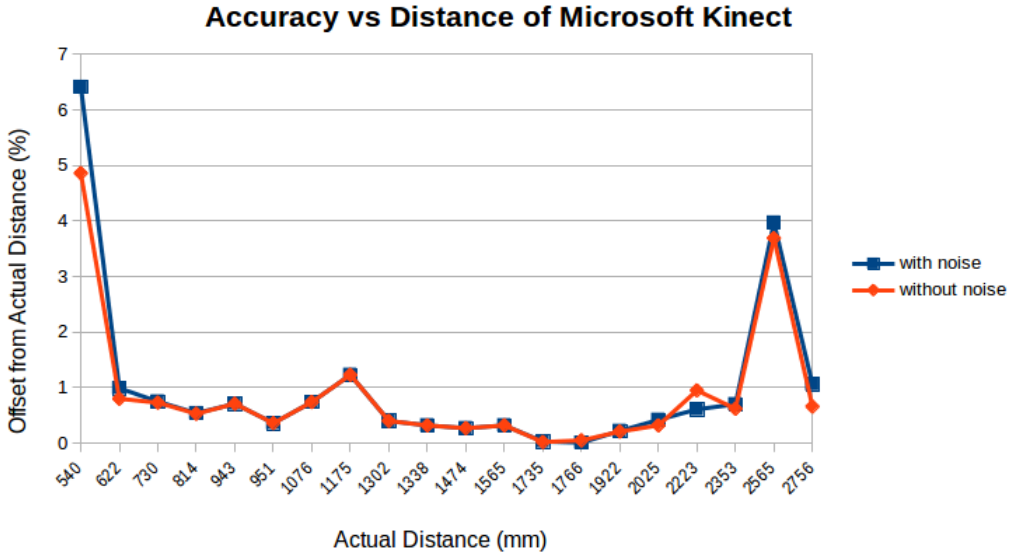
The average range with noise for the Kinect is similar to the results found by K. Khoshelham *et al.* [47] as well as M. Andersen *et al.* [52]. The average offset by Khoshelham was between 110 mm and 120 mm, whereas the results in this dissertation reported 121.9 mm on average, and 65.8 mm if noise is removed. The difference between the results here and those by Khoshelham is that the range in distance is calculated per pixel over 60 frames, and then averaged. Whereas Khoshelham averaged the range of all the pixels from a single frame.

The approach of this section is not necessarily to list the technical specifications of a specific sensor, but to instead use this information to help provide an approach to use for all types of depth sensors to determine if they are capable of accurate 3D room modelling.

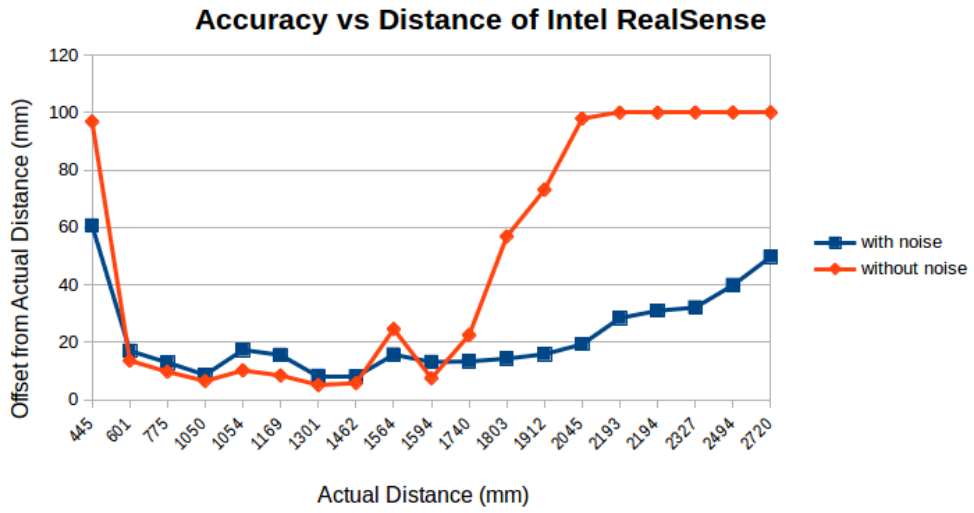
Although both methods returned similar results, the method performed in this dissertation is more relevant to 3D room modelling. This is because a modelling algorithm should average results between frames to smooth data. Testing only one frame may result in more error caused by a bad frame being captured. By averaging frames the noise is reduced in the system.

5.2.2 Accuracy

The range matrix for each distance is kept to help remove noise from the accuracy results. The results show the percentage offset of the sensor at various ranges, with and without noise. These results are shown in Figure 5.12.



(a) Accuracy plot for Microsoft Kinect with noise and without noise.



(b) Accuracy plot for Intel RealSense with noise. The data without noise fluctuates a lot more due to the average distance tending towards 0 as the main points left are 0 after noise removal.

Figure 5.12: Accuracy vs Distance of the Microsoft Kinect and Intel RealSense.

The values used for the plots in Figure 5.12(a) and Figure 5.12(b) can be seen in Tables A.1 and A.2 respectively. It can be noted from the plot of the Kinect's accuracy with and without noise, that the Kinect contains little to no noise in its results. For both sensors their closest distance recorded was at approximately the minimum rated distance each sensor could measure. For the accuracy results these first values were not included in the averaging shown in Table 5.6.

The reason for the distance offset increasing without noise for the RealSense is due to the amount of 0's that are present in the data, bringing the average distance down and further from the actual distance.

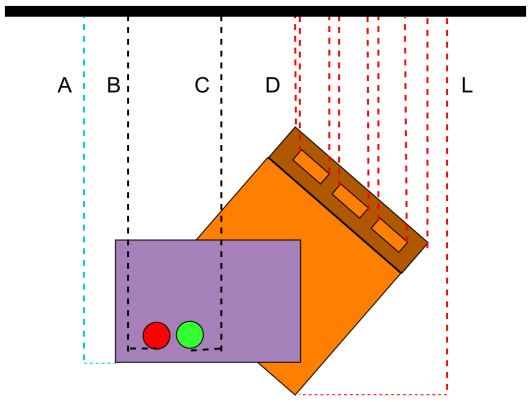
Table 5.6: Average distance accuracy offset as a percentage for the Intel RealSense and Microsoft Kinect.

	Intel RealSense	**Microsoft Kinect**
Offset (%) - With Noise	13.32	1.00
Offset (%) - Without Noise	20.30	0.89

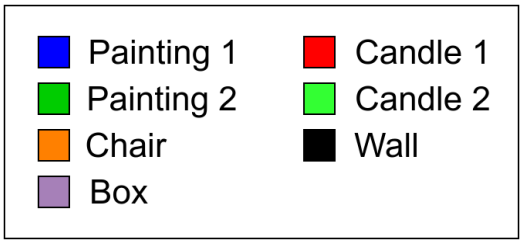
For the average result shown in Table 5.6, the offset was only calculated up to the maximum distance it appears the RealSense can measure, around 2 meters.

5.2.3 Resolution

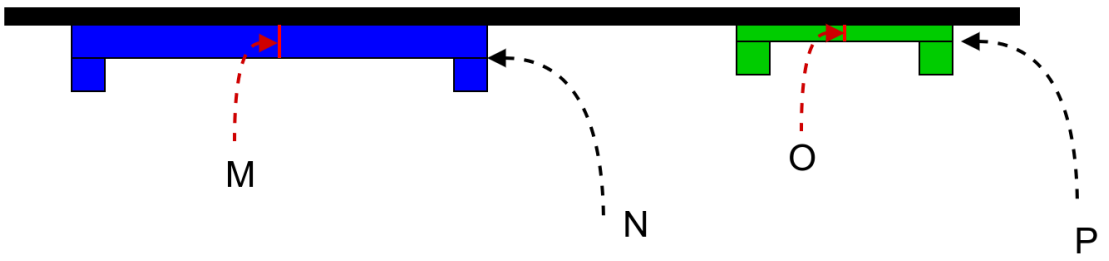
The resolution tests for the sensors used the scene shown in Figure 4.10. The scene was captured at different times for each sensor, therefore, the actual distances of the objects for the tests will vary between sensor results, but each sensor was compared to its individual scene setup. The measurements which were to be found from the scene are shown in Figure 5.13.



(a) Layout of part of the scene with labels for distances measured.



(b) Legend for objects in figure.



(c) Layout of part of the scene with labels for distances measured.

Figure 5.13: Scene being measured for resolution test with labels.

The results of the scenes are shown in Tables 5.7 and 5.8 for the Kinect and RealSense, respectively.

If the recorded distance has an 'NA' it means that the distance was not obtained due to obstructions. If the recorded distance has a 0, however, it means that the sensor could obtain the value as there was no obstruction, but it did not. This was only the case with the Intel RealSense, where the dark frames of the paintings did not produce any features for the RealSense to detect. The built in infrared scatter pattern did not provide any features as it does not show up on dark surfaces. This was discovered during the tests.

Table 5.9 displays the average offset of the distance from the sensor to the various objects in the scene. This average offset was compared to the distance at which the camera was placed. Although the Kinect's performance is much better, both fall within 5% of the distance they were taken at, which can be considered acceptable.

Table 5.7: Results of scene tests for resolution for the Kinect, placed 1483 mm away from the wall.

Point	**Actual Distance from Wall (mm)**	**Recorded Distance from Wall (mm)**	**Difference in mm Average: 13.75**
A	576	571	5
B	534	NA	-
C	526	518	8
D	72	NA	-
E	124	NA	-
F	158	124	34
G	174	157	17
H	205	182	23
I	228	211	17
J	256	229	27
K	283	273	10
L	684	NA	-
M	10	13	3
N	25	22	3
O	25	21	4
P	35	21	14

5.2.4 3D Modelling Ability

To perform this test the two sensors were used to model a room. The room has plain walls with almost no features on them, which as discussed earlier, is one of the more challenging scenarios for an RGB-D sensor to calculate the distance of. If the depth data is not sufficient the modelling software, RTAB-Map, it will not be able to successfully create an entire map of the room, and will get 'lost' in the environment. The output of the Kinect for another apartment is shown in Figure 5.14.

Table 5.8: Results of scene tests for resolution for the RealSense, placed 1216 mm away from the wall.

Point	**Actual Distance from Wall (mm)**	**Recorded Distance from Wall (mm)**	**Difference in mm Average: 40.14**
A	565	565	0
B	506	667	161
C	488	506	18
D	82	101	19
E	120	158	38
F	156	165	9
G	186	216	30
H	223	241	18
I	252	288	36
J	290	309	19
K	326	368	42
L	513	NA	-
M	10	17	7
N	25	0	-
O	25	190	165
P	35	0	-

Table 5.9: Average Offset in mm for each object in the scene used for distance resolution for the Intel RealSense and Microsoft Kinect. As well as their offset relative to the distance at which the cameras were placed.

	Intel RealSense	**Microsoft Kinect**
Offset in mm	40.14	13.75
Offset relative to distance of camera (%)	3.3	0.93

Figure 5.14: The cloud of another apartment taken with the Kinect.

Using the RealSense, this same apartment was modelled. However, the noise from the RealSense, as well as its inability to find depth on dark objects, did not allow for it to complete modelling an entire room. The noise on the sensor seemed to increase on plain objects, such as the wall, which is where it would often lose its orientation. This can be seen in Figures 5.15 and 5.16.

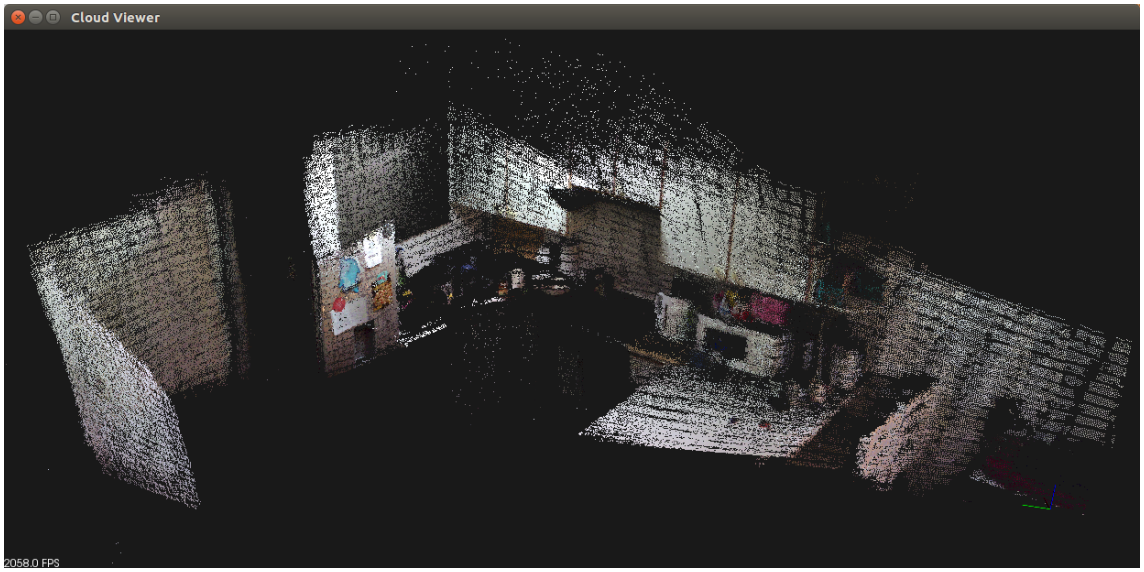


Figure 5.15: An attempted room modelling with the RealSense (kitchen of the room in Figure 5.14).

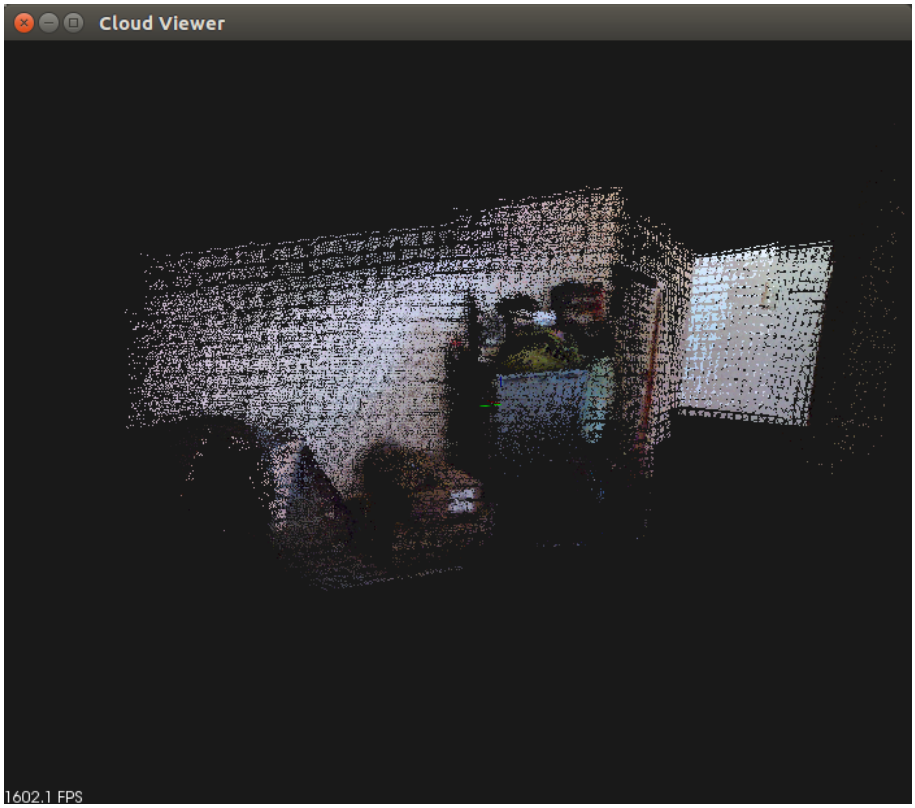


Figure 5.16: An attempted room modelling with the RealSense (wall opposite the kitchen area in Figure 5.14).

The reason why these room models are incomplete is because the modelling software could not locate itself in the environment when the RealSense passed over a wall. It can also be seen that the cloud produced by the Kinect appears to be more dense, and clearer, than the RealSense. The results of the test are shown in Table 5.10. The results for this test are a binary pass or fail, and cannot be represented by a percentage because either the sensor was able to model the room in 3D or not.

Table 5.10: Results of the Intel RealSense and the Microsoft Kinect for their 3D Room Modelling ability. The test was done using an out-of-the-box piece of software to model the room.

Intel RealSense	**Microsoft Kinect**
FAIL	PASS

5.3 Summary

The results of the floor plan creation system are summarised in Table 5.11. The average offset of the two scenes that were tested is 4.6 %, which is within the offset defined by Test 8 in Table 3.2. The offset is

a mixture of not enough pre-processing to remove unwanted points, along with using an external piece of software to perform the room modelling. The error within this software is unknown and could be introducing noise to the system.

The 2D processing software's algorithm also introduces noise by not being able to adequately find all the walls in the room, or join all the potential lines it finds correctly. This lead to the many features being measured and displayed on Figure 5.10. If these lines were averaged better the results of the final bounding box may have been better, resulting in more accurate results overall.

Table 5.11: Summary of floor plan creation results.

Scene	**Offset**
Testing Scene (%)	4.36
Unknown Scene (%)	4.83
Average Offset (%)	4.60

The image tests that were created for this project provided interesting results when used to test the Intel RealSense and the Microsoft Kinect. The summary of these results can be seen in Table 5.12. The RealSense did not perform as well as the Kinect in the tests, this is largely attributed to its difficulty scanning far distances, as well as detecting features on smooth surfaces (with no visual textures). The RealSense is a stereoscopic camera, as defined in Section 2.4.2, meaning it requires visual features to work. The scatter pattern it produces is an attempt to create features, but due to the low intensity of the scatter pattern - resulting in reduced accuracy on dark surfaces or far distances - it fell short here.

The accuracy and range (noise) tests were done on a plain wall, which is the worst case for the RealSense, as it is relying on its scatter pattern to create the features for it. The scatter pattern, being light, will decrease in intensity as the sensor moves further from the object it is sensing, which is why the sensor was unable to see further than 2 meters during these scans, but was able to see further during the 3D modelling scans in Section 5.2.4, this can be inferred by eye. However, whenever the sensor reached a plain white wall in these tests it lost its position and the modelling was aborted, causing it to fail this test. However, the distance resolution tests show that the sensor is accurate to within 5 % like the Kinect.

The average distance resolution offset for the RealSense, although more than the Kinect, is still quite good. This is because the distance is being found for an object, a single point in the data. The distance that is found is not being averaged with any of the pixels around it, therefore returning a non-zero result. This shows that the RealSense, when able to detect features adequately, is an accurate sensor.

Table 5.12: Summary of sensor testing results.

	Intel RealSense	**Microsoft Kinect**
Average Range Offset with noise (mm)	510.83	12.19
Average Range Offset without noise (mm)	9.76	6.58
Average Accuracy Offset with noise (%)	13.32	1
Average Accuracy Offset without noise (%)	20.3	0.89
Average Distance Resolution Offset (%)	3.3	0.93
3D Modelling Ability (Pass/Fail)	Fail	Pass

Chapter 6

Conclusions and Future Work

The following conclusions have been drawn for each area of focus of this dissertation. The conclusions and recommendations will relate back to the results in the previous chapter, as well as to the tests mentioned in section 3.4. The results of the project summarising all the tests can be seen in Table 6.1. Future work leading from this dissertation has been explored towards the end of this chapter.

Table 6.1: Results of tests, linking back to Requirements and Functions in section 3.1.

Section	Requirement	Functionality	Test	Pass/Fail
Floor Plan Creation	R1	F1	T1	PASS
	R2	F2	T2	PASS
		F3	T3	PASS
		F4	T4	PASS
	R3	F5	T5	PASS
		F6	T6	PASS
	R4	F7	T7	PASS
			T8	PASS
		F8	T9	PASS
			T10	PASS
	R5	F9	T11	PASS
Sensor Testing	R6	F10	T12	PASS
			T13	PASS
		F11	T14	PASS
			T15	PASS
		F12	T16	PASS
			T17	PASS
		F13	T18	PASS
			T19	PASS
	R7	F14	T20	PASS
		F15	T21	PASS

6.1 Floor Plan Creation

The floor plan creation system was successfully able to import a point cloud and produce an accurate floor plan from this data, as was demonstrated in Section 5.1. The approach of using the contours of lines to find the dimensions worked well in a single room environment.

However, the use of contours as features provides unnecessary information when used on a larger environment (Figure 5.10). The information received by the sensor consisted of extraneous features resulting in the system believing that there were multiple points to measure, when it was in fact a straight wall. These additional measurements that were labelled on the outputted floor plan, although still useful, hampers readability. With improved noise reduction techniques of the imported data, as well as being able to account for rotated rooms, this may have been avoided.

The inaccuracy incurred in the system is due to things out of the scope of the project, such as the modelling software or the noise that is present in the sensor. Other aspects were excluded during design, such as the inability to find accurate dimensions of a rotated environment, in order to maintain a consistent focus on the project, as this inability was only discovered towards the end of the project timeline.

The system meets its requirements, especially the accuracy requirement of the system. The results of the accuracy can be seen in Table 5.11 whereby the overall average offset of the system is below 5 %. This meets requirement R4 and functionality F8 in Section 3.1. The floor plan creation software also meets all of its other functionalities and requirements - all together deeming the system successful.

6.2 Sensor Tests

The results that were tabulated after processing the raw data from the sensors turned out to be indicative of the results obtained when attempting 3D modelling. Although the results from testing accuracy, noise, and range resolution could be considered all that is necessary for choosing a sensor, it is not all that is needed for choosing a sensor for 3D room modelling. It does however, help in removing a sensor early on if its noise, accuracy, or range resolution is considered undesirable for 3D room modelling.

A sensor might perform well in these tests, but in practice, may not perform well in creating a 3D model. The raw data tests do provide a good basis for choosing the sensor, especially the noise tests, but the 3D modelling test is the final test to determine if a sensor is capable for 3D room modelling. On the

contrary, the raw data tests are needed because a sensor may model a room well by visual inspection, but if the modelled information about the scene is inaccurate, the 3D model is also inaccurate. In conclusion, both quality testing and 3D modelling tests should be undertaken on the sensor.

6.3 Future Work

During the process of developing the system, and in response to the insights gained from the results acquired, a number of ideas were identified for further refinements to the systems. These ideas are presented in this section as recommendations for how this system could be extended should future researchers wish to take this system further, as well as to provide suggestions for researchers involved in the development of similar systems.

6.3.1 Rotated Room Handling

As mentioned in Section 5.1.2, the system is unable to work with a rotated room. A room is considered rotated if the walls of the outputted PNG are not parallel to the x and y axes. A solution to this could be to make use of something similar to [29] whereby the rooms from the 2D image are found using region growing. This will determine the outer limits of any environment regardless of shape and orientation. This approach would also allow for a multiple room scene, which was not explored in this project as it was out of the specified scope.

This approach of region growing was initially tried, but due to the breaks in the contours produced (Figure 5.5(b)) the algorithm struggled to perform its region growing. The implementation of joining the contours to best fit the environment in question, creating a solid contour with no breaks, was taking too long for the project's expected timeline. Therefore, due to time constraints for this investigation, it was not pursued further. However, the current method of using the contours allowed for some features to be detected in the environment, thus allowing for useful information to be seen in the environment.

6.3.2 Robotic Automation

The system could be incorporated into an automated robotic system. Whereby the steps of capturing the point cloud would be done by a robot instead of by a human moving a sensor. This would allow an additional step to be automated and added on to the processing chain. This would enable the block diagram in Figure 4.1 to become one system, as illustrated in Figure 6.1.

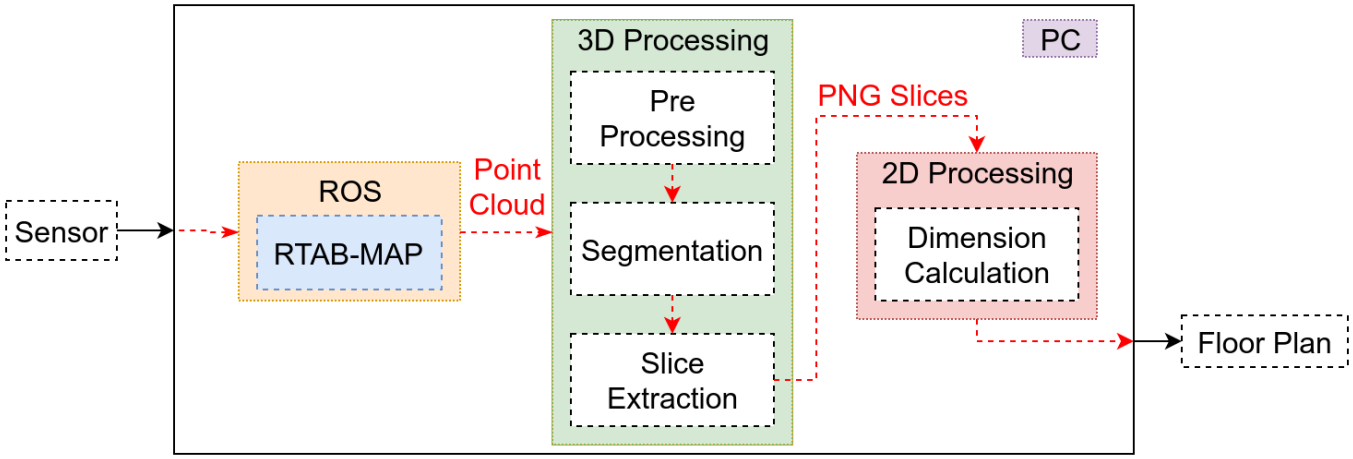


Figure 6.1: Proposed future approach for a fully-automated system to capture the real-world and produce a floor plan.

In the proposed design in Figure 6.1, ROS, running RTAB-Map, will handle all the navigation and robot-related operations necessary to capture the point cloud. These future work ideas will definitely aid the system in capturing the point cloud, however, the floor plan creation can still be performed using the approach presented in this dissertation.

6.3.3 Use of an Improved Sensor

The Kinect V2 (Section 2.4.2), or a sensor of equivalent design, would have been an ideal sensor for this project. The reason this sensor would be the best for 3D room modelling applications, in comparison to the Microsoft Kinect V1 and the Intel RealSense, is because the sensor does not use feature detection to determine distance. The sensor instead uses TOF to determine the distance for each pixel.

This method combines the advantages of LiDAR with the advantages of an RGB-D sensor. The Kinect V2 sensor can go further than an ordinary RGB-D sensor as it is not limited by the pixel resolution of the camera for determining depth. The sensor can scan an entire section of a room at once, unlike the way that a LiDAR sensor operates (Section 2.4.1).

Unfortunately, there was no access to a Kinect V2 for testing. In terms of future work on this aspect, a suggested investigation could involve getting this sensor and performing the same 3D room modelling tests outlined in this dissertation. 3D room models could also be captured with the Kinect V2 to test the accuracy of the floor plan creation software - which will also be indicative of how well the results of the tests correlate to a produced 3D model.

6.3.4 Improved Testing Interface

Another suggestion for future work on this topic is the creation of an all-in-one software bundle. The software would be able to interface to almost any sensor. The software would then perform all the sensor tests mentioned in this paper (noise, accuracy, range resolution), and tabulate the data effectively for the end-user to make informed decisions. The purpose of this software would be a simple way for a user to determine if a sensor is accurate for 3D room modelling.

6.3.5 Creation of Custom Modelling Software

A further recommendation is the addition of a custom software component. The software produced will be responsible for the creation of the point cloud from the sensor, potentially building on top of RTAB-Map. This would be done to improve the integration of the modelling software with the testing interface software mentioned above. When capturing a room, frame-by-frame accuracy reports can be created, and used to improve the creation of the 3D room model. This may allow for decreased noise in the outputted point cloud.

6.4 Summary

The project presented in this dissertation involved the development and testing of two systems, both of which followed a literature-based approach. Theories and techniques from the literature were researched and summarised in order to determine the best approach for achieving the requirements and functionality mentioned in Section 1.2. The requirements and functionality were refined in Section 3.1 using the information gained from the literature review of Chapter 2.

The design of the systems was reported on in Chapter 4, using Chapter 3 as a guideline. The designed systems were implemented using multiple coding languages and open-source libraries, and involved the creation of customised algorithms where existing libraries were not able to provide the necessary services or characteristics.

The results from the created systems were displayed and discussed in Chapter 5. These results were tested against the test matrix (Table 3.2) in order to determine if the systems created satisfied the desired requirements and functionality.

The outcome of the sensor tests provided detailed insight to determine which sensor was the optimum

choice for 3D room modelling. This sensor was then used to model a known and unknown single-room environment in 3D. This 3D modelled room was used as input into the floor plan creation system, from which a 2D floor plan was created. The floor plan's accuracy was within acceptable limits. The project developed solutions that satisfactorily passed all the tests, meeting all the functionality and requirements as discussed in Section 1.2. Therefore, this project can be considered a success.

Appendices

Appendix A

Tables From Sensor Testing

The tables within this appendix relate to the information in Section 5.2. These tables are placed here due to space constraints, and because their information is summarised in the aforementioned section.

Table A.1: Accuracy of Microsoft's Kinect at multiple distances, with and without noise removed.

Actual Distance (mm)	**Measured Distance With Noise (mm)**	**Offset (%) Average: 1.00**	**Measured Distance Without Noise (mm)**	**Offset (%) Average: 0.89**
540	505.39	6.41	513.75	4.86
622	615.81	0.99	617.02	0.80
730	724.51	0.75	724.68	0.73
814	809.64	0.54	809.66	0.53
943	936.33	0.71	936.34	0.71
951	947.57	0.36	947.57	0.36
1076	1067.99	0.74	1067.99	0.74
1175	1160.55	1.23	1160.55	1.23
1302	1296.85	0.40	1296.85	0.40
1338	1333.71	0.32	1333.71	0.32
1474	1470.05	0.27	1470.04	0.27
1565	1559.98	0.32	1559.98	0.32
1735	1735.53	0.03	1735.38	0.02
1766	1765.76	0.01	1765.16	0.05
1922	1926.18	0.22	1925.96	0.21
2025	2033.49	0.42	2031.54	0.32
2223	2236.64	0.61	2201.98	0.95
2353	2369.36	0.70	2338.46	0.62
2565	2463.21	3.97	2470.30	3.69
2756	2726.70	1.06	2737.85	0.66

Table A.2: Accuracy of Intel's RealSense at multiple distances, with and without noise removed.

Actual Distance (mm)	**Measured Distance With Noise (mm)**	**Offset (%) Average: 13.32**	**Measured Distance Without Noise (mm)**	**Offset (%) Average: 20.30**
445	175.25	60.62	14.3	96.79
601	498.95	16.98	519.51	13.56
775	675.06	12.9	699.72	9.71
1050	959.33	8.63	982.09	6.47
1054	871.61	17.3	946.72	10.18
1169	987.31	15.54	1070.75	8.40
1301	1193.57	8.26	1234.72	5.09
1462	1343.66	8.09	1377.95	5.75
1564	1318.86	15.67	1179.79	24.57
1594	1385.24	13.1	1474.95	7.47
1740	1508.59	13.3	1348.78	22.48
1803	1545.68	14.27	778.23	56.84
1912	1609.52	15.82	514.65	73.08
2045	1650.44	19.29	43.64	97.87
2193	1570.06	28.41	2.37	99.89
2194	1514.86	30.95	1.86	99.92
2327	1580.75	32.07	0	100.00
2494	1499.55	39.87	0	100.00
2720	1367.5	49.72	0	100.00

Table A.3: Range fluctuation of Microsoft's Kinect at multiple distances, with and without noise removed.

Actual Distance (mm)	**Average Pixel Range With Noise**	**Average Pixel Range Without Noise**
540	27.22	0.77
622	8.78	1.30
730	2.24	1.34
814	1.30	1.12
943	1.84	1.80
951	3.98	3.97
1076	4.65	4.65
1175	2.52	2.52
1302	2.95	2.93
1338	2.84	2.82
1474	3.52	3.51
1565	5.57	5.57
1735	6.29	6.14
1766	8.25	7.82
1922	14.60	14.35
2025	8.37	7.53
2223	23.37	14.14
2353	25.65	16.03
2565	56.17	12.10
2756	48.89	15.39

Table A.4: Range fluctuation of Intel's RealSense at multiple distances, with and without noise removed.

Actual Distance (mm)	**Average Pixel Range With Noise**	**Average Pixel Range Without Noise**
445	755.68	0.08
601	116.11	2.18
775	150.13	4.48
1050	243.16	10.58
1054	364.61	7.70
1169	454.39	10.96
1301	237.42	13.64
1462	336.07	15.61
1564	886.55	14.72
1594	636.51	16.45
1740	723.58	15.55
1803	844.67	9.29
1912	891.88	5.68
2045	1110.34	0.44
2193	1581.31	0.03
2194	1805.77	0.02
2327	1785.64	0.00
2494	1983.76	0.00
2720	2288.28	0.00

Appendix B

3D Processing Code (C++)

Description	The 3D processing code was written in C++ and makes use of the Point Cloud Library (PCL) which will need to be installed to run the software, as well as an additional library for creating the PNG slices. This can be found at `http://pngwriter.sourceforge.net/`.
Input 1	A point cloud of type *.ply or *.pcd
Input 2	Any value must be inputted here if the cloud is of type *.pcd, otherwise do not use if *.ply.
Output	PNG Slices representing horizontal slices of the point cloud.

The code below is from the **main.cpp** file of the project. ***The full code for this program can be found at:*** `https://github.com/bkwilsonZA/Masters/tree/master/FloorPlanCreation`

```
/*
 ------------------------------------------------------
   3D Processing for My Masters - 2017 - FloorPlanCreator
 ------------------------------------------------------

    This code module was developed by:

      Bradlee Wilson
      Student

      V1.0

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with this program.  If not, see <http://www.gnu.org/licenses/>.
```

```
25
26  ----------------------------------------------------
27  */
28
29 // Initialise the variables representing the minimum and maximum ranges.
30     double minPointZ = 0;
31     double maxPointZ = 0;
32     double minPointX = 0;
33     double maxPointX = 0;
34     double minPointY = 0;
35     double maxPointY = 0;
36
37     // Find the minimum and maximum values for each axis (x, y ,z)
38     for (size_t i = 0; i < interimNewFilter->points.size (); ++i) {
39         if (interimNewFilter->points[i].z > maxPointZ)
40             maxPointZ = interimNewFilter->points[i].z;
41         else if (interimNewFilter->points[i].z < minPointZ)
42             minPointZ = interimNewFilter->points[i].z;
43         if (interimNewFilter->points[i].x > maxPointX)
44             maxPointX = interimNewFilter->points[i].x;
45         else if (interimNewFilter->points[i].x < minPointX)
46             minPointX = interimNewFilter->points[i].x;
47         if (interimNewFilter->points[i].y > maxPointY)
48             maxPointY = interimNewFilter->points[i].y;
49         else if (interimNewFilter->points[i].y < minPointY)
50             minPointY = interimNewFilter->points[i].y;
51     }
52
53
54     pcl::PointCloud<pcl::PointXYZRGB>::Ptr cloudFromSlices(new pcl::PointCloud<pcl::PointXYZRGB>);
55
56     // Determine the range of the height
57     double rangeOfHeight = maxPointZ - minPointZ;
58     double tollerancePercentage = 0.1; // Tollerance dependant on the range, in cm
59
60     // 100 is a magic number here becasue:
61     // * The position points of the point cloud are in relation to real-world metric
62     // * The value is stored as a float using decimal places.
63     // * A point located 1 meter up will have a Z value of 1.000
64     // * Therefore, to get it to cm we will multiply by 100
65
66     // The tollerance is divided by 100 because entering a tollerance of 0.1 would mean
67     // 10 cm, which is quite large. Therefore, dividing it by 100 will relate it to the data
68     double actualTollerance = (maxPointZ - minPointZ)*(tollerancePercentage/100);
69     for (int m = 0; m < 11; m++) {
70         if (m < 6) continue; // If this is slice 5 or below do nothing. This was left in during debugging
71                              // A TODO: Make the for loop range from 6 to < 11
72         double heightToExtract = minPointZ + ((rangeOfHeight/10) * m);
73         for (size_t i = 0; i < interimNewFilter->points.size (); ++i) {
74             double currentHeight = interimNewFilter->points[i].z;
75             if ((currentHeight < (heightToExtract + actualTollerance)) && (currentHeight > (heightToExtract -
                    actualTollerance))) {
76                 cloudFromSlices->points.push_back(interimNewFilter->points[i]); // Push the points to put on the PNG
                        onto a cloud
77             }
78         }
79
80         // Create the PNG slice for this height
81         int i;
82         int y;
83
84         // Multiplying the pixel locations by 100 will result in a PNG image where 1 pixel
85         // represents 1cm. This will ease processing conversion in 2D
86
87         int width = ceil((maxPointX - minPointX)*100); // Determined by the min/max x, multiplied by 100
88         int height = ceil((maxPointY - minPointY)*100); // Determined by the min/max y, multiplied by 100
89         std::stringstream ss;
90         ss << "OutputPNG/slice_" << m << ".png";
91         std::string s(ss.str());
92
93         // http://pngwriter.sourceforge.net/ is used for the PNG creations
94
95         pngwriter png(width,height,0,s.c_str());
96         std::cout << "Plotting height: " << heightToExtract << std::endl;
97
98         for (size_t i = 0; i < cloudFromSlices->points.size (); ++i) {
99             int xValue = ceil((cloudFromSlices->points[i].x - minPointX) * 100);
100            int yValue = ceil((cloudFromSlices->points[i].y - minPointY) * 100);
101            // Create the PNG plot
102            png.plot(xValue, yValue, 1.0, 1.0, 1.0);
```

```
103             }
104
105             png.close();
106
107             cloudFromSlices->clear();
108         }
109
110         // Runs the pythong script to commence creating the floorplan from the script
111
112         // TODO: Make the python script accessible from the path, therefore not needing
113         // it to be present in the same directory as this program.
114         std::string command = "python FloorPlanCreator.py ";
115         system(command.c_str());
116
117         // Used for debugging purposes. In production leave commented or else the program will not end
118         /* ------------------------------
119          * Begin Viewer Loop For Display
120          * ------------------------------ */
121         // PCL_WARN("Begin Display...\n");
122         // while (!viewer->wasStopped())
123         // {
124         //     viewer->spinOnce(100);
125         // }
126
127          return (0);
```

Appendix C

2D Processing Code (Python)

Description	The 2D processing code was written in Python and makes use of the OpenCV library which will need to be installed to run the software, as well as Python itself if it is not on the system.
Input	PNGs representing a horizontal slices of a point cloud at different heights.
Output	A labelled floor plan.

The code below is from the **FloorPlanCreator.py** file of the project. ***The full code can be found at:*** `https://github.com/bkwilsonZA/Masters/blob/master/FloorPlanCreation/build/FloorPlanCreator.py`

```
1  # ------------------------------------------------
2  #    Floor Plan Script for My Masters Project - 2017
3  # ------------------------------------------------
4
5  #     This code module was developed by:
6
7  #       Bradlee Wilson
8  #       Student
9
10 #       V1.0
11
12 #     This program is free software: you can redistribute it and/or modify
13 #     it under the terms of the GNU General Public License as published by
14 #     the Free Software Foundation, either version 3 of the License, or
15 #     (at your option) any later version.
16
17 #     This program is distributed in the hope that it will be useful,
18 #     but WITHOUT ANY WARRANTY; without even the implied warranty of
19 #     MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
20 #     GNU General Public License for more details.
21
22 #     You should have received a copy of the GNU General Public License
23 #     along with this program.  If not, see <http://www.gnu.org/licenses/>.
24
25 # ------------------------------------------------
26
27 # Determines if lines are the same and merges them
28
29 for reps in range(0,repeat):
```

```
30      print(reps)
31      totalX = len(lines)
32      x = 0
33      # The parallelLinesDistance decreases with each scan as they should be getting closer
34      parallelLineDistance = parallelLineDistance / (reps/2+1)
35      # The radius to search for the lines is increasing through as they should be getting
36      # more accurate, less likely to be incorrect.abs
37      radiusToSearch = radiusToSearch + (reps*5)
38      while x < totalX:
39          totalY = len(lines)
40          matchingArray = []
41          for y in range(x, totalY): # The lines which are compared are none before the current index
42              if (len(lines) != len(extendedLinesArray)):
43                  print("Length Mismatch") # If this occurs there's a memory management error
44              if (x != y) and (y < len(lines)):
45                  result = proximityCheck(lines[x], lines[y], extendedLinesArray[x], extendedLinesArray[y],
                        offsetGradient, radiusToSearch, parallelLineDistance, imageHeightY, imageWidthX)
46                  # Whether the lines are close enough is returned into results.
47                  if result == 1: # This means it is not a vertical line
48
49                      # This line is close enough to be considered the same
50
51                      line1X1, line1Y1, line1X2, line1Y2 = lines[x][0]
52                      line1Ends = ((line1X1, line1Y1),(line1X2, line1Y2))
53                      line1Equation = lineEquation(line1Ends)
54
55                      line2X1, line2Y1, line2X2, line2Y2 = lines[y][0]
56                      line2Ends = ((line2X1, line2Y1),(line2X2, line2Y2))
57                      line2Equation = lineEquation(line2Ends)
58
59                      # This line determines which are the furthest points between these two lines.
60                      largeX1, largeY1, largeX2, largeY2 = findLargestMinimumDistance(lines[x][0],lines[1][0], True)
61
62                      # The points that result in the largest distance are added to the array
63                      # The index of the line that matches the current line is also included for later
64                      matchingArray.append([line2Equation,y,[largeX1, largeY1, largeX2, largeY2]])
65                  # This means it is a vertical line, handled separately incase other functionality was needed
66                  elif result == 2:
67                      line1X1, line1Y1, line1X2, line1Y2 = lines[x][0]
68                      line1Ends = ((line1X1, line1Y1),(line1X2, line1Y2))
69                      line1Equation = lineEquation(line1Ends)
70
71                      line2X1, line2Y1, line2X2, line2Y2 = lines[y][0]
72                      line2Ends = ((line2X1, line2Y1),(line2X2, line2Y2))
73                      line2Equation = lineEquation(line2Ends)
74                      # This line determines which are the furthest points between these two lines.
75                      largeX1, largeY1, largeX2, largeY2 = findLargestMinimumDistance(lines[x][0],lines[1][0], True)
76
77                      # The points that result in the largest distance are added to the array
78                      # The index of the line that matches the current line is also included for later
79                      matchingArray.append([line2Equation,y,[largeX1, largeY1, largeX2, largeY2]])
80
81              y = y + 1
82
83          # There were lines which are considered to be on the same line as the line in question
84          if (len(matchingArray) != 0):
85              # There were lines that matched this one
86
87              toDelete = []
88              averageM = 0
89              averageC = 0
90              allX = []
91              allY = []
92
93              # This loop adds the lines to be averaged
94              for z in range(0,len(matchingArray)):
95                  index = matchingArray[z][1]
96                  vertical = False
97                  if (len(matchingArray[z][0]) == 1):
98                      vertical = True
99                  # Add the points to be deleted to an array
100                 # Do not delete now as the array is still
101                 # being used and will disrupt index locations
102                 toDelete.append(index)
103
104                 if vertical == False:
105                     averageM = averageM + matchingArray[z][0][0]
106                     averageC = averageC + matchingArray[z][0][1]
107                 else:
108                     averageM = float("inf")
```

```
                    averageC = averageC + matchingArray[z][0][0]

                # Take the found lines and add their points
                # to the following arrays

                allX.append(matchingArray[z][2][0])
                allX.append(matchingArray[z][2][2])
                allY.append(matchingArray[z][2][1])
                allY.append(matchingArray[z][2][3])

            # This is where the merging actually happens
            if averageM != float("inf"):
                averageM = averageM/len(matchingArray)
            averageC = averageC/len(matchingArray)

            chosenX1 = max(allX)
            chosenX2 = min(allX)

            # Choose the minimum and maximum x values
            # from all the lines which were used

            # Now calculate the corresponding Y values
            # for these X values from the average line
            # equation calculated above

            # If it is a vertical line a different approach is taken
            if averageM != float("inf"):
                calcY1 = averageM*chosenX1 + averageC
                calcY2 = averageM*chosenX2 + averageC
            else:
                xTot = 0
                for h in range(0,len(allX)):
                    xTot = allX[h] + xTot
                chosenX1 = xTot/len(allX)
                chosenX2 = chosenX1
                calcY1 = max(allY)
                calcY2 = min(allY)
            toDelete.sort(reverse=True)

            # The information of the new line is added to the current data

            # Find the extended line of this line
            newExtendedLine = lineExtend(0,0,imageWidthX,imageHeightY,chosenX1, calcY1, chosenX2, calcY2)

            # Add the new lines to the arrays at the current index
            lines[x] = np.array([[chosenX1, calcY1, chosenX2, calcY2]])
            extendedLinesArray[x] = np.array([[newExtendedLine[0],newExtendedLine[1],newExtendedLine[2],newExtendedLine
                [3]]])
            # The old lines are removed
            for i in range(0, len(toDelete)):
                lines = np.delete(lines, toDelete[i], axis=0)
                extendedLinesArray = np.delete(extendedLinesArray, toDelete[i], axis=0)

            totalX = len(lines)
        x = x+1

# The merged lines have been found
```

Appendix D

Octave Script

Description	The numerical script responsible for outputting the results tabulated in Section 5.2. The scripting language is Octave.
Input	CSV files representing different scans at different distances.
Output	CSV files representing useful information about the inputted CSV files.

The code below is from the **cameraAnalysis.m** file of the project. ***This code can be found at:*** `https://github.com/bkwilsonZA/Masters/tree/master/Octave`

```
1  # ------------------------------------------------------
2  #    Octave processing script for My Masters Project - 2017
3  # ------------------------------------------------------
4
5  #     This code module was developed by:
6
7  #       Bradlee Wilson
8  #       Student
9
10 #       V1.0
11
12 #     This program is free software: you can redistribute it and/or modify
13 #     it under the terms of the GNU General Public License as published by
14 #     the Free Software Foundation, either version 3 of the License, or
15 #     (at your option) any later version.
16
17 #     This program is distributed in the hope that it will be useful,
18 #     but WITHOUT ANY WARRANTY; without even the implied warranty of
19 #     MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
20 #     GNU General Public License for more details.
21
22 #     You should have received a copy of the GNU General Public License
23 #     along with this program.  If not, see <http://www.gnu.org/licenses/>.
24
25 # ------------------------------------------------------
26
27 clear
28 # This location is changed depending which sensor is
29 # being used and which tests are being done
30 # accuract or resolution
31
32 # Uncomment the location needed
33
```

```
34 kinectLocation = '/home/bradlee/Documents/Repos/masters-2017/Data/CameraScans/Realsense/Resolution/RAW/*.csv';
35 # kinectLocation = '/home/bradlee/Documents/Repos/masters-2017/Data/CameraScans/Kinect/Resolution/RAW/*.csv';
36 # kinectLocation = '/home/bradlee/Documents/Repos/masters-2017/Data/CameraScans/Realsense/Accuracy/RAW/*.csv';
37 # kinectLocation = '/home/bradlee/Documents/Repos/masters-2017/Data/CameraScans/Kinect/Accuracy/RAW/*.csv';
38
39 files = glob(kinectLocation);
40 AllFiles = cell(1,2);
41 Distances = cell(1,1);
42 numEl = 1;
43
44 for i=1:numel(files)
45   [dir,name,ext] = fileparts (files{i});
46   curM = csvread(fullfile(dir,strcat(name,ext)));
47
48   # The distance of the file is found in the file name
49   if (numEl == 1)
50     currDist = str2num(strsplit(name,'_'){2});
51     Distances(numEl) = currDist;
52     AllFiles{numEl,1} = currDist;
53     temp{1} = curM;
54     AllFiles{numEl,2} = temp;
55     numEl++;
56   else
57     found = 0;
58
59     for j=1:numel(Distances)
60       currDist = str2num(strsplit(name,'_'){2});
61       prevDist = Distances{j};
62       if (currDist == prevDist)
63         AllFiles{j,2}{numel(AllFiles{j,2})+1} = curM;
64         found = 1;
65         break;
66       endif
67     endfor
68
69     if (found == 0)
70       currDist = str2num(strsplit(name,'_'){2});
71       Distances(numEl) = currDist;
72       AllFiles{numEl,1} = currDist;
73       temp{1} = curM;
74       AllFiles{numEl,2} = temp;
75       numEl++;
76     endif
77   endif
78 endfor
79
80 # ************** THE DATA HAS BEEN IMPORTED ************
81
82 # Display the distances that were found
83
84 for i=1:numel(Distances)
85   disp(AllFiles{i,1})
86 endfor
87
88 ## Now we have read them all in
89 # Find average for each Distance and store
90
91 AllAverages = cell(1,2);
92
93 for i=1:numel(Distances)
94   AllAverages(i,1) = AllFiles{i,1};
95   TotalMat = [];
96   for j=1:numel(AllFiles{i,2})
97     if (j == 1)
98       TotalMat = AllFiles{i,2}{j};
99     else
100      TotalMat = TotalMat + AllFiles{i,2}{j};
101    endif
102  endfor
103
104  AllAverages(i,2) = TotalMat/(numel(AllFiles{i,2}));
105 endfor
106
107 ## Now we have all the averages
108 # Find the range
109
110 AllRanges = cell(1,2);
111
112 # For all the distances find the range of the pixels
113
```

```
for i=1:numel(Distances)
  AllRanges(i,1) = AllFiles{i,1};
  [Height, Width] = size(AllFiles{i,2}{1});
  MaxMat = zeros(Height,Width);
  MinMat = zeros(Height,Width);
  MinMat(:,:) = 3000;

  # Determines the minimum/maximum value for each pixel in the two
  # matrices in the min/max functions.

  for j=1:numel(AllFiles{i,2})
    MinMat = min(MinMat, AllFiles{i,2}{j});
    MaxMat = max(MaxMat, AllFiles{i,2}{j});
  endfor

  #Substracting these two matrices gives the range of each pixel
  AllRanges{i,2} = MaxMat - MinMat;
endfor

# This loop is responsible for binning the pixels
# with certain ranges by totalling how many fit
# into a certain bin

RangeDiscrepency = cell(1,1);
for i=1:numel(Distances)
  RangeDiscrepency{i,1} = AllFiles{i,1};
  Ranges = cell(1,1);
  Ranges{1,1} = "0";
  Ranges{1,2} = "1";
  Ranges{1,3} = "2";
  Ranges{1,4} = "3";
  Ranges{1,5} = "4";
  Ranges{1,6} = "5";
  Ranges{1,7} = "6-10";
  Ranges{1,8} = "11-15";
  Ranges{1,9} = "16-25";
  Ranges{1,10} = "26+";
  Values = cell(1,1);
  Values{1,1} = 0;
  Values{1,2} = 0;
  Values{1,3} = 0;
  Values{1,4} = 0;
  Values{1,5} = 0;
  Values{1,6} = 0;
  Values{1,7} = 0;
  Values{1,8} = 0;
  Values{1,9} = 0;
  Values{1,10} = 0;
  RangeDiscrepency{i,2} = Ranges;
  RangeDiscrepency{i,3} = Values;
  [Height, Width] = size(AllRanges{i,2});
  for y=1:Height
    for x=1:Width
      currRange = AllRanges{i,2}(y,x);
      if (currRange == 0)
        RangeDiscrepency{i,3}{1,1}++;
      elseif (currRange == 1)
        RangeDiscrepency{i,3}{1,2}++;
      elseif (currRange == 2)
        RangeDiscrepency{i,3}{1,3}++;
      elseif (currRange == 3)
        RangeDiscrepency{i,3}{1,4}++;
      elseif (currRange == 4)
        RangeDiscrepency{i,3}{1,5}++;
      elseif (currRange == 5)
        RangeDiscrepency{i,3}{1,6}++;
      elseif (currRange <= 10)
        RangeDiscrepency{i,3}{1,7}++;
      elseif (currRange <= 15)
        RangeDiscrepency{i,3}{1,8}++;
      elseif (currRange <= 25)
        RangeDiscrepency{i,3}{1,9}++;
      elseif (currRange > 25)
        RangeDiscrepency{i,3}{1,10}++;
      endif
    endfor
  endfor
endfor

# Here the offset allowed is 25 (for determining what is noise)
```

```
# Here the range average is found for all the distances in
# an attempt to normalise the data. This will produce two
# matrices representing the average range with and without noise

OffsetAllowed = 25;
RangeAverage = cell(1,1);
for i=1:numel(Distances)
  RangeAverage{i,1} = AllFiles{i,1};
  total=0;
  counter = 0;
  total2=0;
  counter2 = 0;
  [Height, Width] = size(AllRanges{i,2});
  for y=1:Height
    for x=1:Width
      total = total + AllRanges{i,2}(y,x);
      counter++;
      if (AllRanges{i,2}(y,x) =< OffsetAllowed)
        total2 = total2 + AllRanges{i,2}(y,x);
        counter2++;
      endif
    endfor
  endfor
  RangeAverage{i,2} = (total/counter);
  RangeAverage{i,3} = (total2/counter2);
endfor

# Here the offset used is 25 from above (for determining what is noise)
# Here the accuracy average is found for all the distances in (i.e. the average distance)
# an attempt to normalise the data. This will produce two
# matrices representing the average accuracy (distance) with and without noise

AccuracyAverage = cell(1,1);

for i=1:numel(Distances)
  AccuracyAverage{i,1} = AllFiles{i,1};
  DistanceInQuestion = AllFiles{i,1};
  total=0;
  counter = 0;
  total2=0;
  counter2 = 0;
  [Height, Width] = size(AllAverages{i,2});
  for y=1:Height
    for x=1:Width
      total = total + AllAverages{i,2}(y,x);
      counter++;
      if (AllRanges{i,2}(y,x) =< OffsetAllowed)
        total2 = total2 + AllAverages{i,2}(y,x);
        counter2++;
      endif
    endfor
  endfor
  AccuracyAverage{i,2} = (total/counter);
  AccuracyAverage{i,3} = (total2/counter2);
endfor

Accuracy1Simple = [];
Accuracy2Simple = [];
DistanceSimple = [];
Range1Simple = [];
Range2Simple = [];

## The data above is trasnfered from vectors to matrices
## Vectors were used because they can hold object
## That way text could be included, and an enture matrix
## could be stored as an element.

for i=1:numel(Distances)
  Accuracy1Simple = [Accuracy1Simple AccuracyAverage{i,2}];
  Accuracy2Simple = [Accuracy2Simple AccuracyAverage{i,3}];
  Range1Simple = [Range1Simple RangeAverage{i,2}];
  Range2Simple = [Range2Simple RangeAverage{i,3}];
  DistanceSimple = [DistanceSimple Distances{i}];
endfor

RangeDiscrepencySimple = []

for i=1:numel(Distances)
  TempArray = [];
  for j=1:numel(Ranges)
```

```
274       TempArray = [TempArray RangeDiscrepency{i,3}{j}];
275     endfor
276     RangeDiscrepencySimple = [RangeDiscrepencySimple; TempArray];
277   endfor
278
279   ## ******* THE CSV FILES ARE OUTPUTTED ************
280   ##
281   ## These CSV files are then used to tabulate the
282   ## results of the sensor tests
283   ##
284   ## ********************************************
285
286   csvwrite("Accuracy1.csv",Accuracy1Simple);
287   csvwrite("Accuracy2.csv",Accuracy2Simple);
288   csvwrite("Range1.csv",Range1Simple);
289   csvwrite("Range2.csv",Range2Simple);
290   csvwrite("DistancesSimple.csv",DistanceSimple);
291   csvwrite("RangeDiscr.csv",RangeDiscrepencySimple);
```

Bibliography

[1] D. Takahashi. (2013). Beyond kinect, primesense wants to drive 3d sensing into more everyday consumer gear, [Online]. Available: https://venturebeat.com/2013/01/20/beyond-kinect-primesense-wants-to-drive-3d-sensing-into-more-everyday-consumer-gear/.

[2] S. Ochmann, R. Vock, R. Wessel, M. Tamke, and R. Klein, "Automatic Generation of Structural Building Descriptions from 3D Point Cloud Scans," *Proceedings of GRAPP 2014 - International Conference on Computer Graphics Theory and Applications*, vol. January, 2014.

[3] Autodesk. (2017). Autocad overview, [Online]. Available: https://www.autodesk.co.za/campaigns/inspired-by-autocad/compare-autocad.

[4] A. Oliver, S. Kang, B. C. Wünsche, and B. MacDonald, "Using the Kinect as a navigation sensor for mobile robotics," *Proceedings of the 27th Conference on Image and Vision Computing New Zealand - IVCNZ '12*, pp. 509–514, 2012. DOI: 10.1145/2425836.2425932. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2425836.2425932.

[5] C. Mura, O. Mattausch, A. Jaspe Villanueva, E. Gobbetti, and R. Pajarola, "Automatic room detection and reconstruction in cluttered indoor environments with complex room layouts," *Computers and Graphics (Pergamon)*, vol. 44, no. 1, pp. 20–32, 2014, ISSN: 00978493. DOI: 10.1016/j.cag.2014.07.005. [Online]. Available: http://dx.doi.org/10.1016/j.cag.2014.07.005.

[6] Intel. (2016). Intel realsense r200, [Online]. Available: https://software.intel.com/en-us/articles/realsense-r200-camera.

[7] Scanse. (2017). Scanse — afforable scanning lidar for everyone, [Online]. Available: http://scanse.io/.

[8] Microsoft. (2017). Kinect for windows hardware, [Online]. Available: https://msdn.microsoft.com/en-us/library/jj131033.aspx.

[9] S. Ochmann, R. Vock, R. Wessel, and R. Klein, "Automatic reconstruction of parametric building models from indoor point clouds," *Computers and Graphics (Pergamon)*, vol. 54, pp. 94–103, 2016, ISSN: 00978493. DOI: 10.1016/j.cag.2015.07.008. [Online]. Available: http://dx.doi.org/10.1016/j.cag.2015.07.008.

[10] S. Oesau, F. Lafarge, and P. Alliez, "Indoor scene reconstruction using feature sensitive primitive extraction and graph-cut," *ISPRS Journal of Photogrammetry and Remote Sensing*, vol. 90, pp. 68–82, 2014, ISSN: 09242716. DOI: 10.1016/j.isprsjprs.2014.02.004.

[11] R. Marks. (2010). Eyetoy - innovation and beyond, [Online]. Available: https://blog.us.playstation.com/2010/11/03/eyetoy-innovation-and-beyond/.

[12] Introlab. (2017). Overview, [Online]. Available: http://introlab.github.io/rtabmap/.

[13] R. B. Rusu and S. Cousins, "3D is here: Point Cloud Library (PCL)," in *IEEE International Conference on Robotics and Automation (ICRA)*, Shanghai, China, 2011.

[14] P. C. Library. (2017). Normal estimation - documentation - point cloud library, [Online]. Available: http://pointclouds.org/documentation/tutorials/normal_estimation.php.

[15] ——, (2017). Statistical outlier removal - documentation - point cloud library, [Online]. Available: http://pointclouds.org/documentation/tutorials/statistical_outlier.php#statistical-outlier-removal.

[16] ——, (2017). Downsampling - documentation - point cloud library, [Online]. Available: http://pointclouds.org/documentation/tutorials/voxel_grid.php.

[17] ——, (2017). Region growing segmentation - documentation - point cloud library, [Online]. Available: http://pointclouds.org/documentation/tutorials/region_growing_segmentation.php#region-growing-segmentation.

[18] L. Outcomes and L. Style, "Curvature," *Engineering Mathematics: Open Learning Unit Level 1*, p. 14, [Online]. Available: http://www3.ul.ie/{~}mlc/support/Loughboroughwebsite/book.html.

[19] P. C. Library. (2017). Extract indices, [Online]. Available: http://pointclouds.org/documentation/tutorials/extract_indices.php#extract-indices.

[20] ——, (2017). Difference of normals based segmentation, [Online]. Available: http://pointclouds.org/documentation/tutorials/don_segmentation.php#don-segmentation.

[21] OpenCV. (2017). About - opencv library, [Online]. Available: https://opencv.org/about.html.

[22] ——, (2017). Eroding and dilating, [Online]. Available: https://docs.opencv.org/2.4/doc/tutorials/imgproc/erosion_dilatation/erosion_dilatation.html.

[23] sansuiso. (2013). Difference between "edge detection" and "image contours" - stack overflow, [Online]. Available: https://stackoverflow.com/questions/17103735/difference-between-edge-detection-and-image-contours.

[24] OpenCV. (2017). Find contours - documentation, [Online]. Available: https://docs.opencv.org/2.4/modules/imgproc/doc/structural_analysis_and_shape_descriptors.html?highlight=findcontours#findcontours.

[25] J. Canny, "A Computational Approach to Edge Detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-8, no. 6, pp. 679–698, 1986, ISSN: 01628828. DOI: 10.1109/TPAMI.1986.4767851.

[26] OpenCV. (2017). Canny detector - tutorials, [Online]. Available: https://docs.opencv.org/2.4/doc/tutorials/imgproc/imgtrans/canny_detector/canny_detector.html.

[27] R. O. Duda and P. E. Hart, "Use of the Hough transformation to detect lines and curves in pictures," *Communications of the ACM*, vol. 15, no. 1, pp. 11–15, 1972, ISSN: 00010782. DOI: 10.1145/361237.361242. [Online]. Available: http://portal.acm.org/citation.cfm?doid=361237.361242.

[28] A. W. R. Fisher S. Perkins and E. Wolfart. (2003). Hough transform, [Online]. Available: http://homepages.inf.ed.ac.uk/rbf/HIPR2/hough.htm.

[29] N. Estner. (2013). Image processing: Floor plan, [Online]. Available: https://mathematica.stackexchange.com/questions/19546/image-processing-floor-plan-detecting-rooms-borders-area-and-room-names-t.

[30] A. Rosebrock. (2016). Measuring size of objects in an image with opencv, [Online]. Available: https://www.pyimagesearch.com/2016/03/28/measuring-size-of-objects-in-an-image-with-opencv/.

[31] R. A. Brooks, "Symbolic Error Analysis and Robot Planning," *The International Journal of Robotics Research*, vol. 1, no. 4, pp. 29–78, 1982, ISSN: 17413176. DOI: 10.1177/0278-3649-82001-00403.

[32] R. C. Smith and P. Cheeseman, "On the Representation and Estimation of Spatial Uncertainty," *The International Journal of Robotics Research*, vol. 5, no. 4, pp. 56–68, 1986, ISSN: 17413176. DOI: 10.1177/027-836-498-6005-00404.

[33] O. S. R. Foundation. (2017). Out ros, [Online]. Available: http://www.ros.org/about-ros/.

[34] S. Kohlbrecher, O. Von Stryk, J. Meyer, and U. Klingauf, "A flexible and scalable SLAM system with full 3D motion estimation," *9th IEEE International Symposium on Safety, Security, and Rescue Robotics, SSRR 2011*, pp. 155–160, 2011, ISSN: 2374-3247. DOI: 10.1109/SSRR.2011.6106777.

[35] J. Zhang and S. Singh, "LOAM : Lidar Odometry and Mapping in Real-time," *Robotics: Science and Systems*, 2014, ISSN: 0929-5593. DOI: 10.1007/s10514-016-9548-2.

[36] T. Optech. (2017). Lynx hs-600 mobile survey system, [Online]. Available: http://www.teledyneoptech.com/wp-content/uploads/Lynx_HS-600_specsheet_161110_LR.pdf.

[37] J. Tan, J. Li, X. An, and H. He, "Robust curb detection with fusion of 3d-Lidar and camera data," *Sensors (Switzerland)*, vol. 14, no. 5, pp. 9046–9073, 2014, ISSN: 14248220. DOI: 10.3390/s140509046.

[38] J. Mrovlje and D. Vran, "Distance measuring based on stereoscopic pictures," *9th International PhD Workshop on Systems and Control: Young Generation Viewpoint*, vol. 2, no. October, pp. 1–6, 2008.

[39] L. Keselman, J. I. Woodfill, A. Grunnet-Jepsen, and A. Bhowmik, "Intel RealSense Stereoscopic Depth Cameras," 2017, ISSN: 21607516. DOI: 10.1109/CVPRW.2017.167. arXiv: 1705.05548.

[40] M. Carfagni, R. Furferi, L. Governi, M. Servi, F. Uccheddu, and Y. Volpe, "On the Performance of the Intel SR300 Depth Camera: Metrological and Critical Characterization," *IEEE Sensors Journal*, vol. 17, no. 14, pp. 4508–4519, 2017, ISSN: 1530437X. DOI: 10.1109/JSEN.2017.2703829.

[41] Wikipedia. (2018). Intel realsense generations and specifications, [Online]. Available: https://en.wikipedia.org/wiki/Intel_RealSense.

[42] O. Demetz, "Correspondence Problems in Computer Vision," 2014.

[43] CuriousInventor. (2013). How the kinect depth sensor works in 2 minutes, [Online]. Available: https://www.youtube.com/watch?v=uq9SEJxZiUg.

[44] A. Nag and S. Deshmukh, "Real Time Tracking System using 3D Vision," no. December 2015, 2016. DOI: 10.13140/RG.2.1.3513.4489.

[45] O. Ťupa, A. Procházka, O. Vyšata, M. Schätz, J. Mareš, M. Vališ, and V. Mařík, "Motion tracking and gait feature estimation for recognising Parkinson's disease using MS Kinect," *BioMedical Engineering Online*, vol. 14, no. 1, pp. 1–20, 2015, ISSN: 1475925X. DOI: 10.1186/s12938-015-0092-7.

[46] A. M. Ahmad, Z. Bazzal, and H. A. Youssef, "Kinect-Based Moving Human Tracking System with Obstacle Avoidance," *Advances in Science, Technology and Engineering Systems Journal*, vol. 2, no. 3, pp. 191–197, 2017.

[47] K. Khoshelham and S. O. Elberink, "Accuracy and resolution of kinect depth data for indoor mapping applications," *Sensors*, vol. 12, no. 2, pp. 1437–1454, 2012, ISSN: 14248220. DOI: 10.3390/s120201437. arXiv: arXiv:1505.0193.

[48] M. Hansard, S. Lee, O. Choi, R. Horaud, M. Hansard, S. Lee, O. Choi, R. Horaud, and F. Cameras, *Time of Flight Cameras : Principles , Methods , and Applications To cite this version : HAL Id : hal-00725654*. 2012, ISBN: 9781447146582.

[49] Microsoft. (2017). Kinect hardware, [Online]. Available: https://developer.microsoft.com/en-us/windows/kinect/hardware.

[50] D. Lau. (2013). The science behind kinects or kinect 1.0 versus 2.0, [Online]. Available: https://www.gamasutra.com/blogs/DanielLau/20131127/205820/The_Science_Behind_Kinects_or_Kinect_10_versus_20.php.

[51] A. Note, *Radar Measurements*. 2012, ISBN: 9781891121524.

[52] M. Andersen, T. Jensen, P. Lisouski, A. Mortensen, M. Hansen, T. Gregersen, and P. Ahrendt, "Kinect Depth Sensor Evaluation for Computer Vision Applications," *Electrical and Computer Engineering*, vol. Technical, p. 37, 2012, ISSN: 2245-2087. DOI: TechnicalReportECE-TR-6. arXiv: f47766f5-9ed1-46e2-a773-f5c319507790.