Electronics and Computer Science
Faculty of Physical Sciences and Engineering
University of Southampton

Adam Melvin

April 2018

Towards Presence in Telerobotics: Real-time Image Abstraction for Virtual Reality

Project supervisor: Klaus-Peter Zauner
Second examiner: Sheng Chen

A project report submitted for the award of
MEng Electronic Engineering

UNIVERSITY OF SOUTHAMPTON

Abstract

Faculty of Physical Sciences and Engineering
Electronics and Computer Science

MEng Electronic Engineering

by Adam Melvin

The application of Virtual Reality (VR) to telerobotics is a current area of study due to a desire for the increased spacial awareness VR provides. However, attempts to implement such a system using standard teleoperation techniques result in sub-par performance and an uncomfortable experience for the user; the benefits of a VR based system are entirely eliminated. The system proposed by this project incorporates elements of data abstraction to produce abstractions of the environment. This process reduces the performance requirements of providing the user presence within the telerobot's space such that it can be achieved under reasonable budgetary constraints.

Contents

1 Introduction **1**

2 Background **5**
2.1 Virtual Reality . . . 5
2.2 Telerobotics . . . 6
2.2.1 VR in Telerobotics . . . 6
2.3 Data Abstraction . . . 8
2.4 Sobol Sequences . . . 8
2.5 Computer Vision . . . 9
2.5.1 Edge Detection . . . 9
2.5.2 Flood Filling . . . 10
2.5.3 Depth Mapping . . . 11

3 System Overview **13**

4 Abstraction Algorithm Development **15**
4.1 Edge Detection Process . . . 17
4.2 Colour Averaging . . . 17
4.2.1 Seed Points . . . 18
4.2.2 Fill Colour . . . 19

5 Rover Implementation **23**
5.1 Gimbal Design . . . 25
5.2 Image Pipeline . . . 27
5.3 Control System . . . 32

6 Server-Side Implementation **35**
6.1 Depth Map and Abstraction Construction . . . 36
6.2 3D Model Generation . . . 41

7 Testing and Evaluation **43**
7.1 3D Model Quality . . . 43
7.2 Run-Time Performance . . . 47
7.3 Comparison to a Non-Abstraction System . . . 47

8 Conclusion **49**

9 Project Management **51**

Bibliography **57**

A Demonstrations of Full Data Abstraction **59**

B Contour Based Seed Point Location **61**

C Images of Algorithm Performance Test Scenes **63**

D Rover Hardware Breakdown **65**

E Vectorization **67**

F Project Brief **69**

G Gantt Chart **71**

H Risk Assessment **73**

I Design Archive Contents **75**

List of Figures

1.1 System Outline . . . 3
1.2 Rover Pictures . . . 4

2.1 HTC Vive . . . 5
2.2 Indirect VR Teleoperations Examples . . . 7
2.3 Data Abstraction Example . . . 8
2.4 Comparison of Pseudo-Random and Quasi-Random Sequences . . . 9
2.5 Canny Edge Detection Example . . . 10
2.6 Flood Filling Example . . . 11
2.7 Depth Mapping Example . . . 12

3.1 System Overview Block Diagram . . . 14

4.1 Data Abstraction Algorithm Process . . . 16
4.2 Performance Comparison of Seed Point Methods . . . 19
4.3 Comparison of Brute Force and Sobol Seed Point Generation . . . 19
4.4 Comparison of Colour Averaging Methods . . . 21

5.1 Rover Internals . . . 24
5.2 Hardware Block Diagram . . . 25
5.3 Gimbal Picture . . . 27
5.4 Raspberry Pi Image Pipeline Threading Block Diagram . . . 28
5.5 Comparison of Image File Formats . . . 30
5.6 Comparison of Laptop and Rover Abstraction . . . 31

6.1 Server Program Structure . . . 36
6.2 Demonstration of Stereo Image Rectification . . . 37
6.3 Comparison of Depth Mapping Algorithm Speed . . . 38
6.4 Comparison of Raw Depth Mapping Algorithm Output . . . 39
6.5 Filtered StereoBM and StereoSGBM Outputs . . . 40
6.6 Mesh Grid Segment . . . 41
6.7 Final Textured 3D Model . . . 42

7.1 Full System Demonstrations . . . 44
7.2 Demonstration of System Errors . . . 46

7.3 Comparison of Packet Size for Reliable Depth Maps 48
7.4 Non-Abstraction Depth Map Comparison at Equal Packet Size 48

A.1 Demonstration of Full Abstraction Process 59
A.2 Additional Examples of the Full Abstraction Process 60

B.1 Demonstration of Contour Centre Point Location 62

C.1 Images of Algorithm Performance Test Scenes 64

E.1 Conversion to Vector Graphics 67
E.2 Comparison of Vector and Bitmap Accuracy 68

G.1 Gantt Chart 72

H.1 Risk Assessment 74

Acknowledgements

I would like to thank Klaus-Peter Zauner for his incredible help and support. I would also like to thank Tom Darlison and Lawrence Gray for their help and for being expert "rubber ducks". Finally, I would like to thank the users of the Building 16 labs for treating my rover driving around with the utmost patience.

Chapter 1

Introduction

Whether it be within a virtual space such as in a video game or training simulation [1], or within a remote real world location as in telerobotics, a feeling of presence [2] allows the user to more naturally and intuitively interact with the presented environment as if they were truly there. The desire for presence within a space that is not your own is therefore one that drives much technological innovation. In telerobotics in particular, where the aim can often be to interact with dangerous or industrial environments, intuitive control is essential to safe and effective operation.

Virtual Reality (VR) is a technology spearheaded by the video games industry for use in immersive gaming applications. Through the use of a tracked headset, giving the user the ability to freely look around a 3D space, it provide unparalleled presence within a virtual world—comparable to presence within a real, physical space [3, 4]. To be able to incorporate VR into teleoperation is therefore desirable, however, sending a video feed to the headset as if it were a normal monitor has been found to lead to motion sickness [5]. This is due to VR's high frame rate and low latency requirements. It's widely accepted that for a VR application to not cause motion sickness and headaches due to frame rate, it must maintain at least 90 frames per second (fps) [6]; a minimum of 60 fps can also be acceptable [7], but generally only for applications with little motion or when used by people with lower susceptibility to motion sickness. Unfortunately, to transmit 90 fps from a stereo camera rig (two images are required to perceive 3D) to the computer running the VR application has

bandwidth requirements too high to be currently implementable outside the most expensive of designs.

Another consideration is the ability to look around the space independently, as this is a major factor in providing presence to the user in VR. This can be achieved by mounting the stereo camera rig on a gimbal, however to build a gimbal that is able to track the angle of the user's head accurately and with low latency is, once again, expensive and challenging [8]; if not implemented perfectly the user would experience significant sickness and dissociation from the space.

The aim of this project is to research the use of data abstraction to minimise the outlined technical issues through the design and implementation of a teleoperations system that incorporates it. To achieve this, the system must reduce each image down to its most essential features, reducing its size and therefore the required bandwidth significantly. Each image pair must then be transmitted to a server and combined into a single 3D model of the space that could be looked around freely through the VR headset. As the camera feed is converted to a 3D model rather than displayed directly as images, the headset could run at its maximum frame rate of 90 fps even if the model is updating at a much slower rate. The use of abstraction has the capacity to allow for high presence systems, as presence in VR is not dependant on the 3D environment being an exact reproduction of the telerobot's space, only that the reproduction is consistent and comfortable to view.

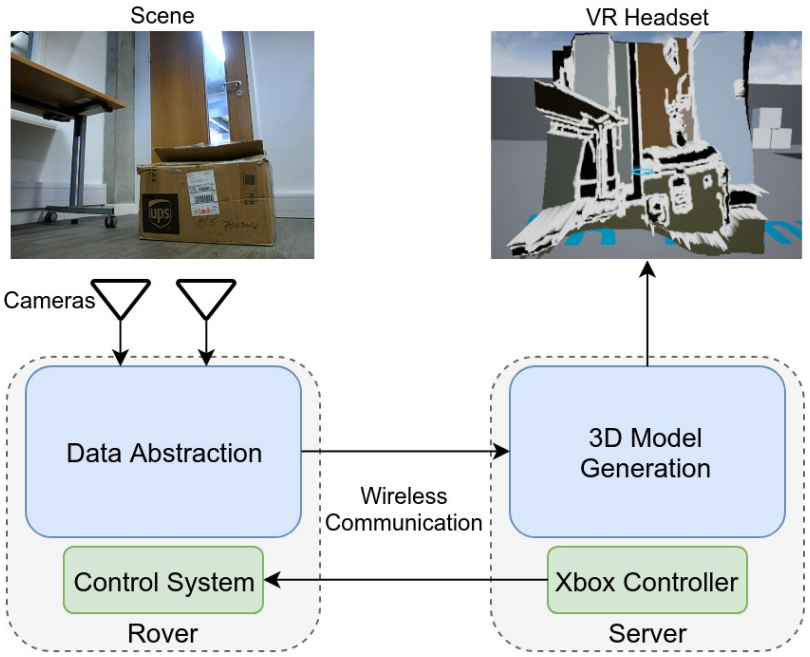


FIGURE 1.1: System Outline. It can be seen that the scene the rover's cameras are observing (top left) is within the VR environment (top right) as a simple abstracted 3D representation.

The system (cf. Figure 1.1) consists of two platforms: a server (Chapter 6) that runs a VR environment and reads user input from an Xbox 360 controller, and a rover platform (Chapter 5) that is controlled from said environment and supplies the abstracted images that the 3D model in the environment is built from. The rover (Figure 1.2) is a simple drivable platform with a stereo camera gimbal mounted on it (adapted from one produced by previous students [9]). The server is a powerful PC running Windows 10 [10] and a HTC Vive [11]. The data abstraction algorithm the system uses is novel, so its design and development is initially discussed in isolation (Chapter 4), and then its implementation within the system explained as part of the rover implementation (Chapter 5).

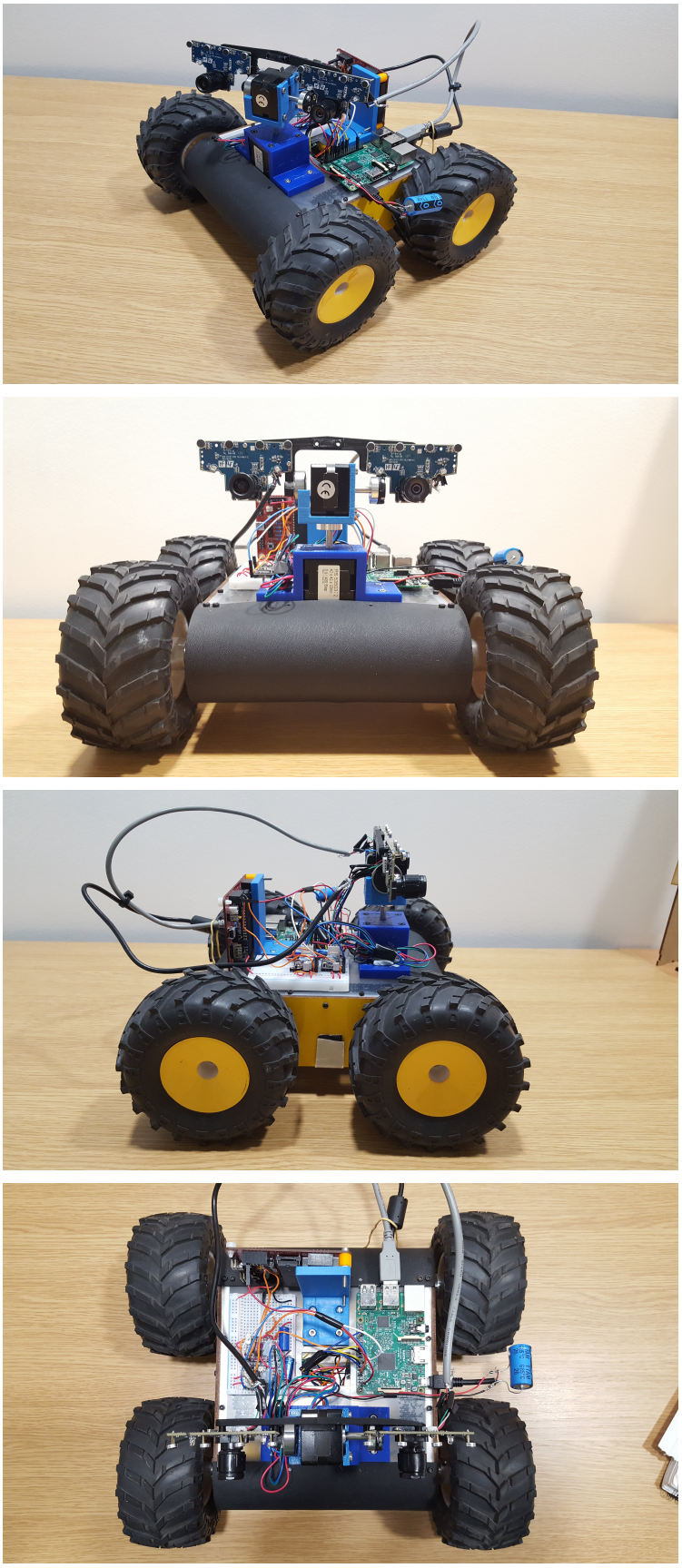


FIGURE 1.2: Rover Pictures.

Chapter 2

Background

2.1 Virtual Reality

The term Virtual Reality (VR) refers to the generation of a 3D environment that can be interacted with by a user in a realistic fashion, with the aim of immersing the user in the environment as if it were the real world [12]. While there are a large array of systems that can be considered VR, whenever the term is used in this report it is only referring to the head-mounted display (HMD) systems that have become popular in recent years with the release of the Oculus Rift [13] and the HTC Vive [11] (Figure 2.1). These are both consumer grade systems that are aimed at the immersive gaming market.

FIGURE 2.1: HTC Vive. Pictures of the Vive headset, reproduced from [11].

HMD based systems display different images for each eye to provide the user with a sense of depth within the 3D environment, making the headset effectively operate like a pair of binoculars into the virtual world. The headset is also tracked in 3D space, and this movement translated into the 3D environment with low latency. These features, among others, are all implemented with the aim of providing the user with presence within the virtual space that is comparable to observing the real world.

Due to its availability at the University of Southampton and in my own home, the HTC Vive was used as the VR device in the development and implementation of the teleoperations system discussed in this report.

2.2 Telerobotics

A telerobot is a robot controlled from a distance by a human operator [14]. Telerobots are typically developed to undertake activities within environments that are too dangerous or costly for humans to work in. Typical fields of telerobotics research include deep-space exploration [15], deep-sea exploration [16], and handling radioactive materials [17].

2.2.1 VR in Telerobotics

The use of HMDs in teleoperations is not a new concept; NASA's Robonaut 2 was sent to the International Space Station in 2011 and can be controlled through a headset that displays the output of the robot's head cameras [18], and flying drones by First Person View (FPV), an analogue video feed transmitted over radio to a HMD, has become popular in recent years [19]. However, these systems are either incredibly expensive (Robonaut 2 is worth millions of dollars) or very limited (FPV systems are only capable of sending one low quality video stream over a short distance), and all suffer from the motion sickness issues discussed in Chapter 1. While stereo camera FPV systems have been developed, so the user has depth perception and better presence in the drone's view, the motion sickness problem remains the major drawback of HMD based teleoperations systems [20].

As previously established, motion sickness in VR is mitigated through high frame rates and low latency. However, VR based teleoperations systems are typically direct VR systems, so they display the video feeds provided by the telerobot's cameras directly in the headset. This entirely ties the frame rate and latency of the headset to the capabilities of the video transmission system, and only the most expensive and complicated systems will meet the strict requirements for comfortable VR and high spacial presence.

An alternate option to a direct system is an indirect system. This is one in which the video feed is abstracted from the headset in some way in the hope of providing improved comfort and awareness. Indirect systems can come in a variety of forms, such as placing the video feed on a virtual screen in the VR environment and controlling the robot using virtual controls laid out infront of the screen [21], or a 3D map generated from a multi-line LiDAR and IMU [22] (Figure 2.2). These examples show the potential of indirect systems as a solution to VR based teleoperations, however neither of them provide true presence within their respective robots' environments. This project aims to contribute to this field of research by demonstrating an alternate approach to the problem that is capable of providing greater presence than these previous attempts.

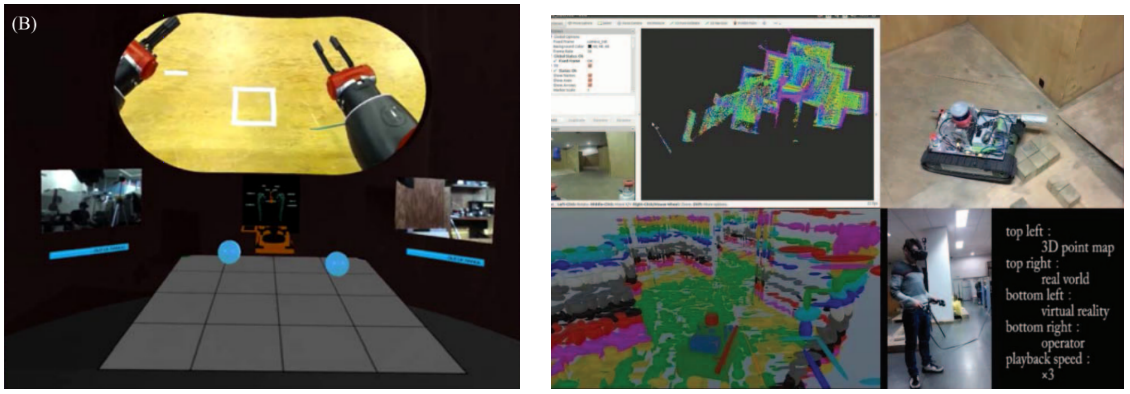


FIGURE 2.2: Indirect VR Teleoperations Examples. A virtual control room based system (left) and a LiDAR 3D map based system (right), reproduced from [21] and [22].

2.3 Data Abstraction

Data abstraction is the phrase that will be used in this report to describe the act of reducing an image down to only its most essential elements. It is similar in concept to an artist sketching a scene instead of attempting a full drawing. Comparing the concept to data compression is apt, as both aim to reduce the file size of the image, however data abstraction takes a very different approach to solving the problem than standard compression algorithms.

Data compression is the storing of information using a more space efficient encoding [23]. While some information is lost during lossy compression, the aim regardless of the algorithm used is to retain as much of the original information as possible. In contrast, the aim when utilising data abstraction is to discard all the information that is unnecessary to fulfilling the image's specific purpose. For example, if all that is required of an image is that basic shapes can be identified, then only the information on the boundaries of the shapes is necessary; the rest of the image can be discarded. An implementation of data abstraction can be seen in Figure 2.3.

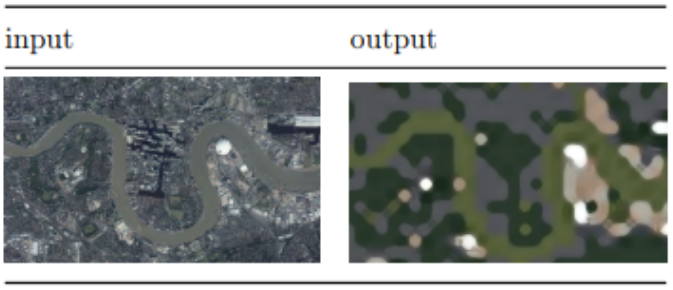


FIGURE 2.3: Data Abstraction Example. This is an abstraction of an aerial photograph of London, reproduced from [24].

2.4 Sobol Sequences

Sobol sequences are quasi-random sequences that were introduced to aid in approximating integrals. The aim is to form a sequence of points that are evenly spread across an S-dimensional unit cube [25], providing a much more even spread of points across the chosen space than can be produced from a pseudo-random number source

(Figure 2.4). The code used in this project to produce these sequences was created by Leonhard Grünschloß [26].

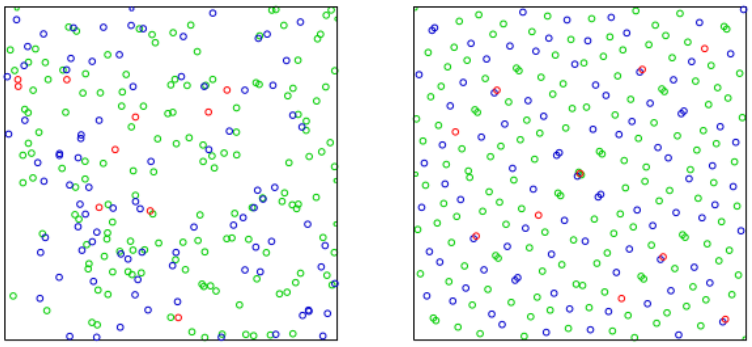


FIGURE 2.4: Comparison of Pseudo-Random and Quasi-Random Sequences. 256 points from a pseudo-random generator (left) and 256 points from a Sobol sequence (right), reproduced from [27].

2.5 Computer Vision

Computer vision is the automatic analysis of images and extraction of the useful information they contain [28]. A raw image is simply a large matrix of colour values, so for a computer to take action based on the contents of an image it must be able to recognise features using analysis of this data. Doing so involves a variety of techniques such as statistical pattern classification and geometric modelling [29]. All computer vision methods in this project are implemented using the OpenCV libraries, and the example programs provided with them used as starting points for development [30].

2.5.1 Edge Detection

When attempting to recognise the features of an image, knowing the locations of the edges of objects within the scene is often valuable. An edge is defined as a significant local change in intensity, usually due to a discontinuity in either the intensity or its first derivative [31]. There are many algorithms available that will detect the edges of an image from the locations of these discontinuities. When the most popular algorithms (Laplacian of Gaussian, Robert, Prewitt, Sobel, and Canny) are compared

[32], the most effective in almost all scenarios is Canny edge detection [33], therefore this is the algorithm utilised in this project. Canny edge detection is demonstrated in Figure 2.5.

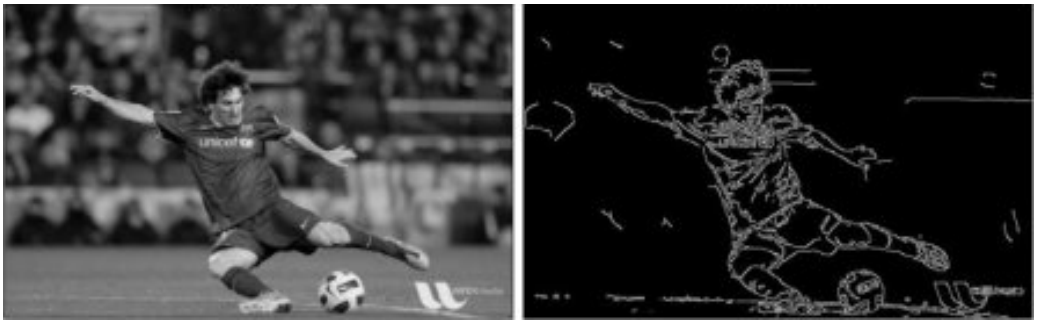


FIGURE 2.5: Canny Edge Detection Example. Simple edge detection program applied to a fairly detailed photo of Lionel Messi, to demonstrate its effectiveness with even complex images. Figure taken from an OpenCV edge detection tutorial [34].

2.5.2 Flood Filling

Flood fill algorithms determine the area connected to a given cell (the seed point) in a multi-dimensional array that have similar intensity values for the purpose of filling them with a chosen colour [35]. This is a technique that is not only useful in image processing, but also for many other fields such as in passive acoustic monitoring, where finding the area connected to a given node can be useful as part of tracking in 4D space (x,y,z,time) [36]. A demonstration of flood fill has been presented in Figure 2.6.

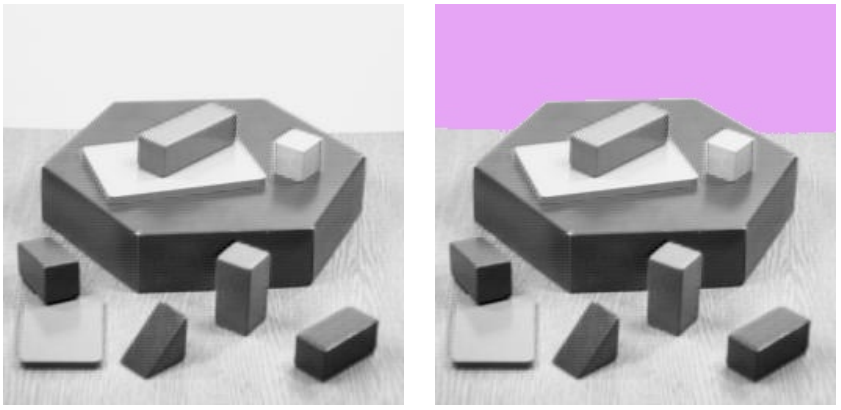


FIGURE 2.6: Flood Filling Example. The original image (left) was provided by OpenCV [30]. The right image is the result of flood filling from the top left corner.

2.5.3 Depth Mapping

The main component in the human brain's perception of 3D is the identification of disparity between the locations of objects in the 2D images being produced by its eyes [37]. The greater the difference in the horizontal placement of an object between the images, the closer the object is to the observer. This technique can be used in computer vision to produce depth/disparity maps. Depth maps display differences in depth as a gradient from white to black (Figure 2.7), and can be produced using a variety of different algorithms. The most common are block matching algorithms, which use simple geometry and the matching of blocks of pixels horizontally in the 2 images to calculate depth [38]. For these algorithms to locate the same object in different places in the 2 images, the cameras taking them must be calibrated to rectify any distortion due to the lenses [39] or discrepancies in the mounting that would cause them to be out of line [40]. Without this image rectification, the similar blocks the algorithm is attempting to locate will not be on the same horizontal row or will be distorted, so the algorithm will produce a mostly black image.

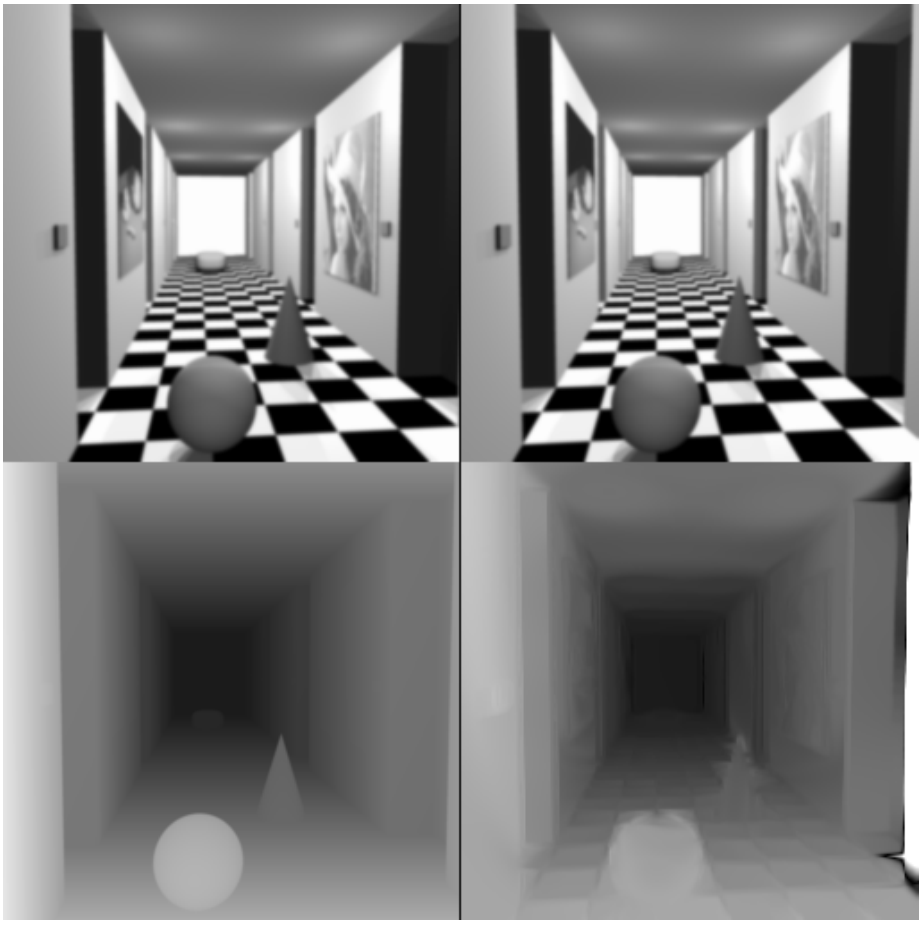


FIGURE 2.7: Depth Mapping Example. A stereo pair of images, with their exact disparity map bottom left and a disparity map produced by a dense disparity estimation algorithm bottom right, reproduced from [41].

Chapter 3

System Overview

The telerobotics system presented in this report consists of a fairly complex image processing pipeline running alongside a simple control system, with little interaction between the two (cf. Figure 3.1). The pipeline starts with the two cameras on the rover taking a picture each as close to simultaneously as possible. These images then get abstracted down to edge detected sets of lines, with a set of colours also generated from *Image 1* to combine with its edge detected version later on. These three pieces of data are then compressed into a single packet and transmitted via UDP [42] to the server. The server splits them up again and produces a depth map from the two edge detected images. This depth map is passed to the 3D Environment to be used to produce a 3D model of the space the cameras were looking at. The server also combines the *Colour Data* and *Edge Detected Image 1* into a full coloured abstraction, which is overlaid onto the 3D model in the 3D environment as a texture. This environment is finally observed in the VR Headset.

The rover is controlled from an Xbox 360 controller [43] that is connected to the server. Control inputs for driving the rover, gimble orientation, and parameters for the data abstraction (the only crossover between the control and image processing components) are transmitted to the rover at a fixed rate over UDP. It should be noted that in this system there is no connection between the orientation of the headset and gimble. The focus of this project was on providing the image processing foundations for future high presence systems, therefore the control system is the minimum required to explore the effectiveness of the chosen approach.

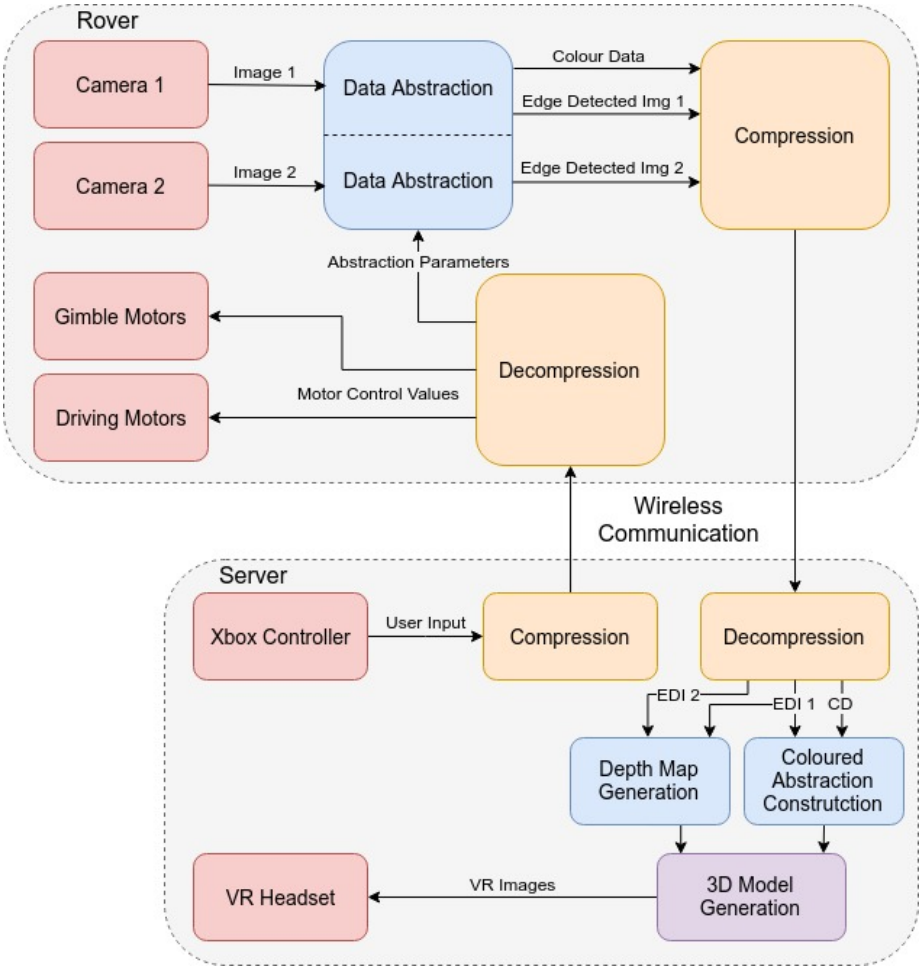


FIGURE 3.1: System Overview Block Diagram. The red blocks represent input/output, the blue blocks represent image processing steps, the yellow blocks represent communications, and the purple block represents the 3D environment.

Chapter 4

Abstraction Algorithm Development

We defined data abstraction as the reduction of an image down to only its most essential elements (Section 2.3). What these essential elements are depends on the application, leading to a large range of possible methods and results. In this implementation of data abstraction the aim is to be able to navigate a telerobot from the resultant images, therefore the most important elements of each image are the size and shape of the objects in it. It is possible to convey the rough shape of an object by presenting its outline, therefore leading to edge detection being chosen as the first step of the algorithm. Identification of objects, while possible using edges alone, is significantly easier when colour is also provided. Most of the colours in an image, however, do not aid in object identification; one colour (preferably the average colour) for each object in the scene is sufficient. This leads to the algorithm presented in Figure 4.1, with examples of the abstractions it produces presented in Appendix A.

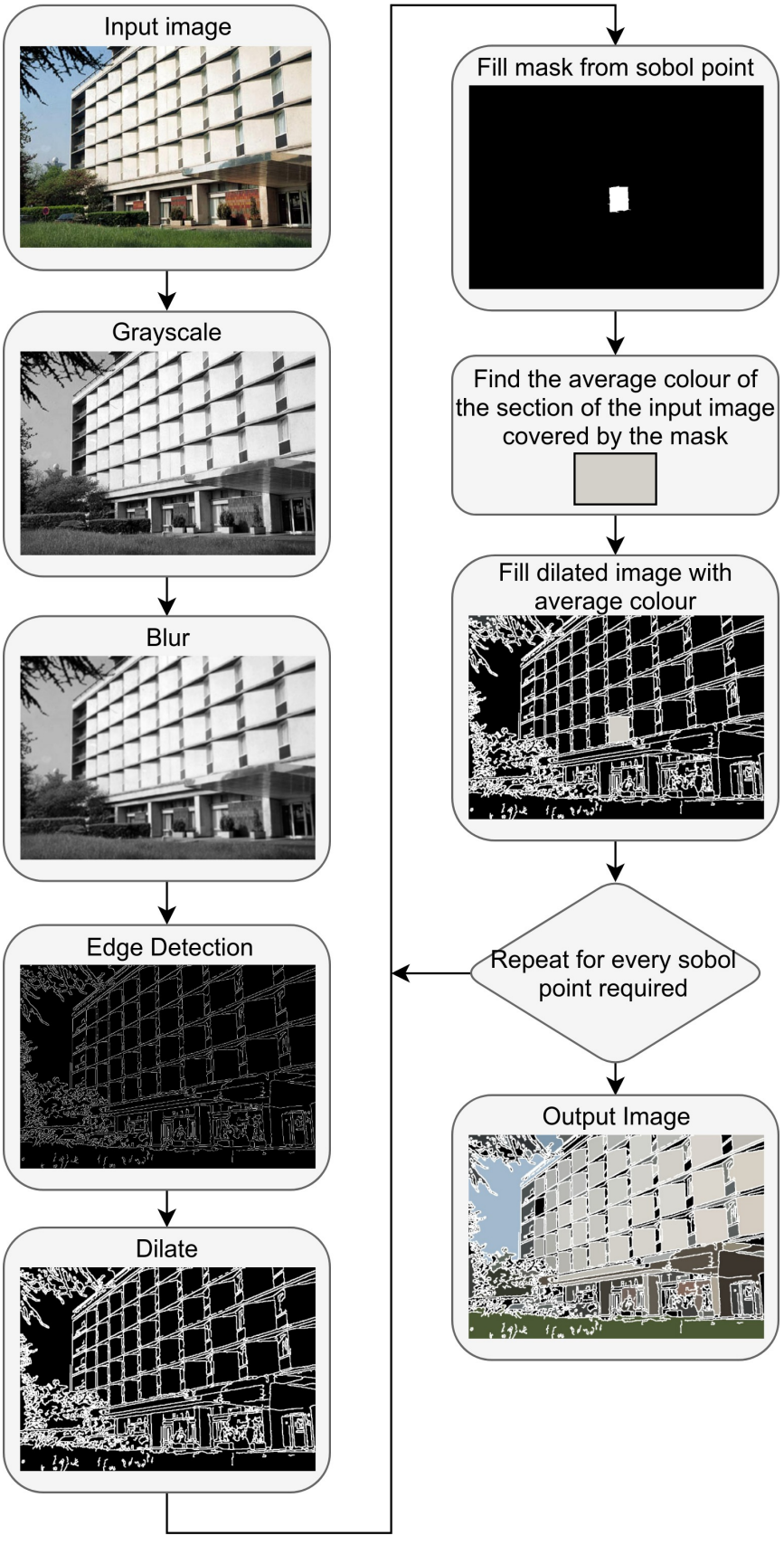


FIGURE 4.1: Data Abstraction Algorithm Process. The left half is explained in the *Edge Detection Process* section and the right half in the *Colour Averaging* section, both located below.

This chapter is presented in the context of the entire process occurring on a single computer; when incorporated into the final system, the whole process is undertaken on the rover apart from the *Fill dilated image with average colour* block, which is implemented on the server in its own loop through every Sobol point (*Coloured Abstraction Construction* on Figure 3.1). It is also worth noting that this chapter is concerned with the algorithm's ability to produce recognisable abstractions under reasonable resource constraints, and not the file sizes of said abstractions. The file sizes it is capable of producing will be covered as part of the system testing and evaluation in Chapter 7.

4.1 Edge Detection Process

Canny edge detection was implemented using the Canny function provided by OpenCV [30]. The image is first converted into a grayscale format, as the edge detector detects large changes in intensity and not colour, then blurred to remove any unnecessary edges and noise. The edge detection is applied, producing white lines representing the edges on a black background. The output of the edge detection is finally dilated to make the lines thicker and bridge the gaps between the lines that are very close together. This is done to reduce the number of lines produced by areas that are dense with detail such as hair and foliage and to increase the likelihood of defined shapes being created that can be easily flood filled later.

4.2 Colour Averaging

Flood filling was chosen as the method for applying colour to the edge detected image, as it is an effective method for filling spaces of unknown size and shape that are defined by high contrast boundaries. The OpenCV flood fill function requires a seed point to start filling from, a colour to fill with, and parameters for the filling itself (unchanged from the defaults provided by the OpenCV documentation [44]).

4.2.1 Seed Points

Finding the points to flood fill from is a challenge, as each image will have a different number of spaces to be filled and the spaces can be of any size and shape. Three different methods were attempted to solve this problem. The first was an attempt to use OpenCV's contour functionality to convert the lines into a set of contours and use the centre of mass of each contour (an approximation of the centre of each space) as the seed point. Unfortunately this was unusable due to high resource requirements. The method is detailed in Appendix B.

Although it would by ideal to aim to flood fill from the centre of each space, it is only necessary if the intention is to be selective about which pixels are being used as seed points. It is possible to instead iterate through the whole image and flood fill from every pixel found that is not part of an edge or an already filled space. This method is effective at filling every space, and is also less resource intensive than the previous method. However, if presented with a complicated environment with many spaces to flood fill it must fill every single one, leading to unacceptable drops in frame rate.

A simple solution to the performance issues caused by complex images would be to set a maximum number of times flood fill can be used per image. However, if this is done then the seed points can no longer be selected by iterating through the whole image, as the presence of many small spaces at the top of an image would lead to larger, more important spaces not being filled at the bottom. The chosen solution to this issue is to select a set number of points quasi-randomly across the image using Sobol sequencing (explained in Section 2.4). Although a certain number of points will land on lines and therefore not be used, if enough points are used then all the important spaces are filled without performance being heavily influenced by the complexity of the image as in the brute force method above (Figure 4.2). However, complicated images will be processed with many of their denser areas unfilled (Figure 4.3). It was decided that this was an adequate trade-off for the performance improvements using Sobol points provides, therefore the seed points are selected using this method in the final design.

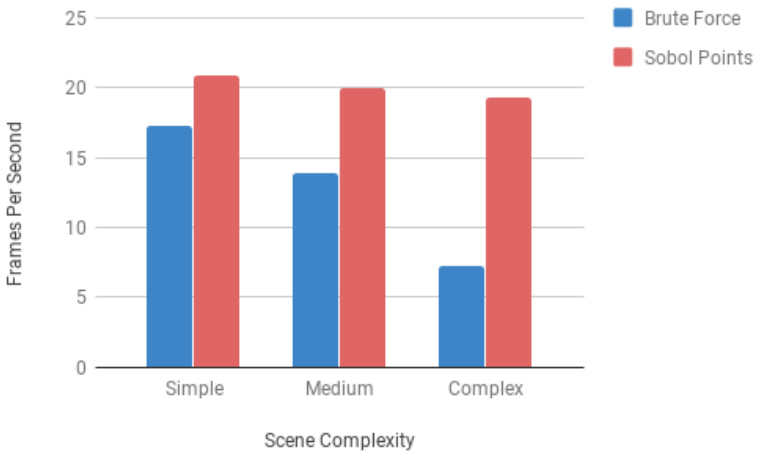


FIGURE 4.2: Performance Comparison of Seed Point Methods. It can be clearly seen that as the complexity of the scene increases, the brute force method results in unacceptable drops in frame rate, whereas the sobol method maintains a much higher degree of consistency. Images of the scenes used in this test can be found in Appendix C.

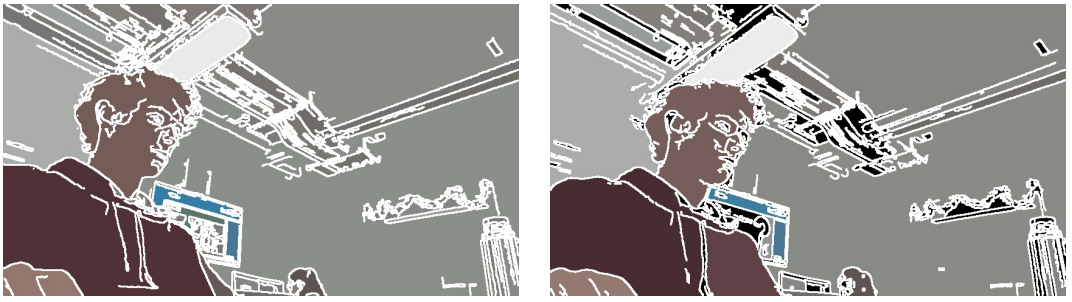


FIGURE 4.3: Comparison of Brute Force and Sobol Seed Point Generation. It can be observed that the brute force method (left) accurately fills every space in the image, whereas using the Sobol method results in many unfilled spaces.

4.2.2 Fill Colour

Three different methods were considered for finding the colours that the edge detected images must be flood filled with. All three methods are valid solutions, but present different ratios between accuracy and resource requirements.

Filling the abstracted image with the average colours of the objects present within the input image would be the clear optimal choice for the purposes of object recognition.

However, the use of average colours is not essential; the objects in the scene can still be recognisable with significantly lower colour accuracy. For this reason it would be acceptable to not find the average colour of the area being flood filled at all, and instead simply use the colour present in the input image at the seed point. This is a very fast method, however produces wildly inconsistent colours between images. This is because there can be a wide spectrum of colour across a single surface even within the threshold of Canny edge detection, and the Sobol sequence will not always sample from the same point in each new frame, producing spaces that flicker between a wide range of colours.

The consistency of the previous method can be improved substantially with minimal impact on performance by taking an average of the colour within the area of a small circle around the seed point, rather than just the colour of that one point. This significantly improves the consistency between images, however introduces the problem of incorporating pixels from outside the space in question. This is due to the quasi-random points often being so close to the edge of the space that the averaging circle crosses the edge slightly and incorporates part of a neighbouring space in the average. Therefore, the size of the circle must be carefully selected to balance the benefits of increasing size (more consistency when the seed point is near the centre of the space) and the benefits of decreasing size (more consistency when the seed point is closer to the edge).

The OpenCV flood fill function provides the ability to fill a blank mask using the boundaries defined by a different image [45]. This makes it possible to create a custom mask with the exact size and shape of the space that is to be filled and use that to find the average colour instead of a predefined circle. This produces the average colour of every space in the image exactly at the expense of adding an extra stage of flood filling before the edge detected image itself is filled (a stage of flood filling that must be undertaken on the rover in the full system). The consistency between images for this is the maximum possible based on colour alone (Figure 4.4), though inconsistency in the edge detection causes certain spaces to combine and divide constantly, leading to a small amount of colour inconsistency to remain regardless. The fact that this method leads to flood filling being undertaken on the rover as well as the server leads to performance concerns, however the consistency it produces led it to be the chosen method for the full system.

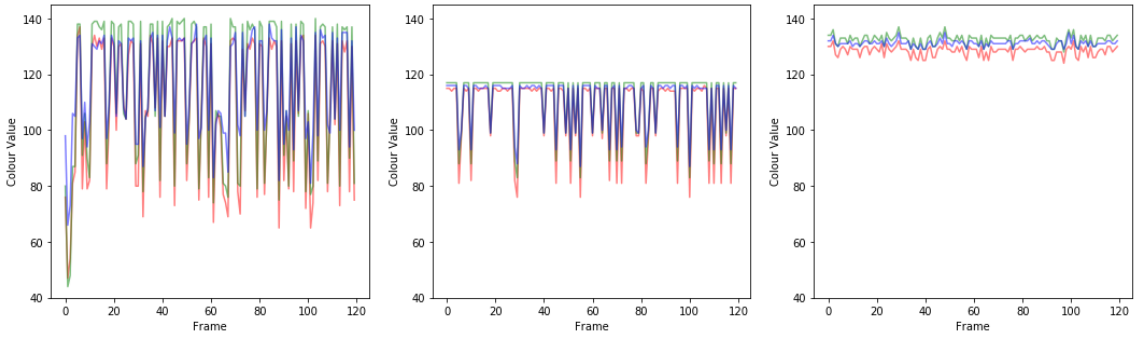


FIGURE 4.4: Comparison of Colour Averaging Methods. These are the RGB values over time produced by flood filling an example area using the colour of the seed point (left), circle average (middle), and flood fill average (right). The increase in consistency from left to right is very apparent.

Chapter 5

Rover Implementation

Due to the rover being no more than a test platform, its hardware is fairly simple. It consists of a chassis containing batteries, a power distribution board, motor drivers and motors used to drive the rover's wheels, with the central computer (Raspberry Pi 3 [46]), gimbal, and gimbal control circuitry all mounted on top (Figure 5.1, 5.2). A list of the specific hardware used can be found in Appendix D. The application of computer vision techniques in an embedded system has high processing requirements, leading to the selection of a Raspberry Pi 3 (R-Pi) as the core of the system (it is widely available and well supported by OpenCV).

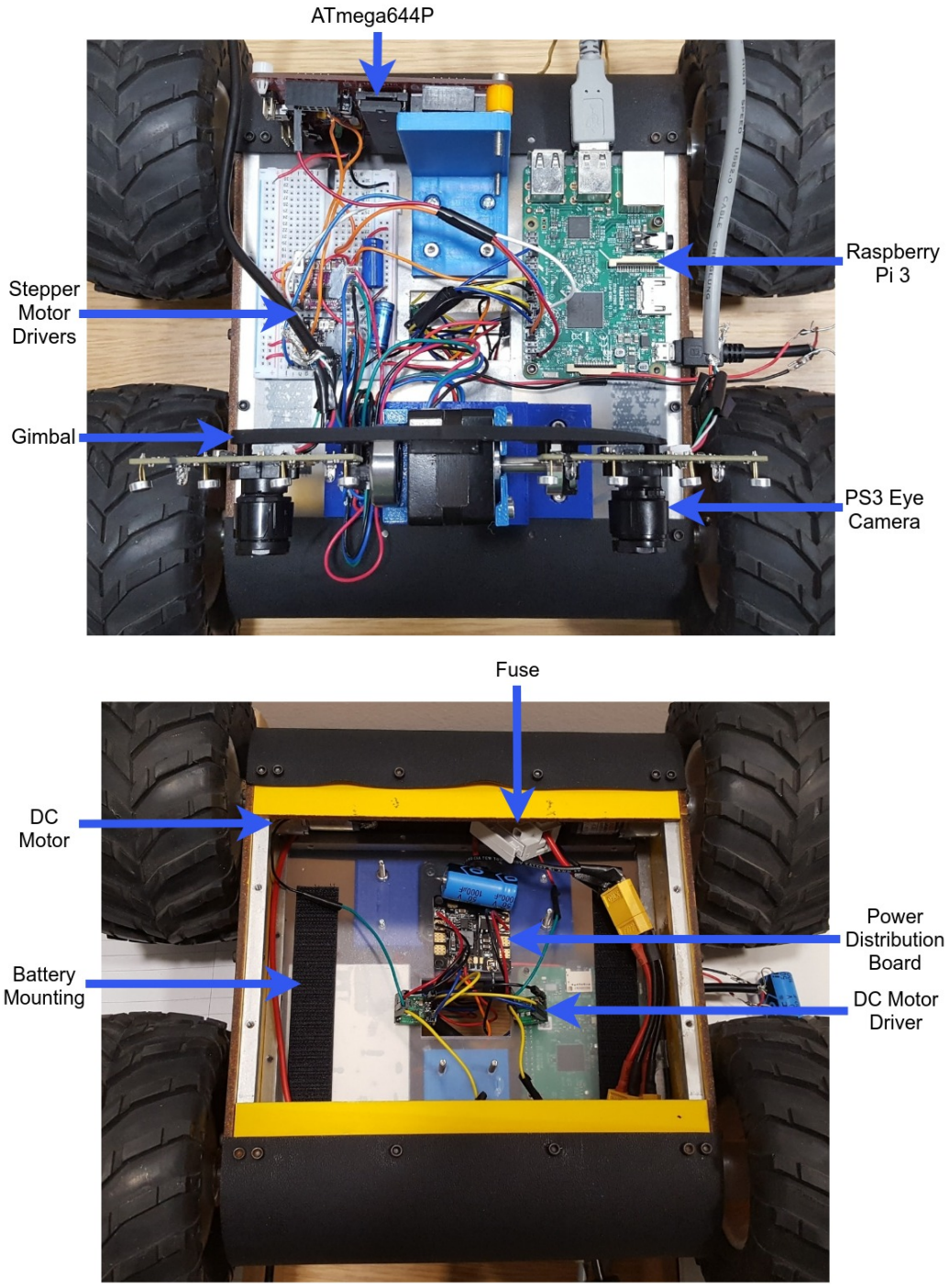


FIGURE 5.1: Rover Internals. The top picture is a top-down view of rover, in which you can see the Pi, gimbal control circuitry, and gimbal. The bottom picture is of the inside of the rover, taken through a hatch in its underside. In this picture you can see velcro strips marking where the batteries are mounted, power distribution board, safety fuse, 2 DC motor drivers, and 2 of the 4 DC motors.

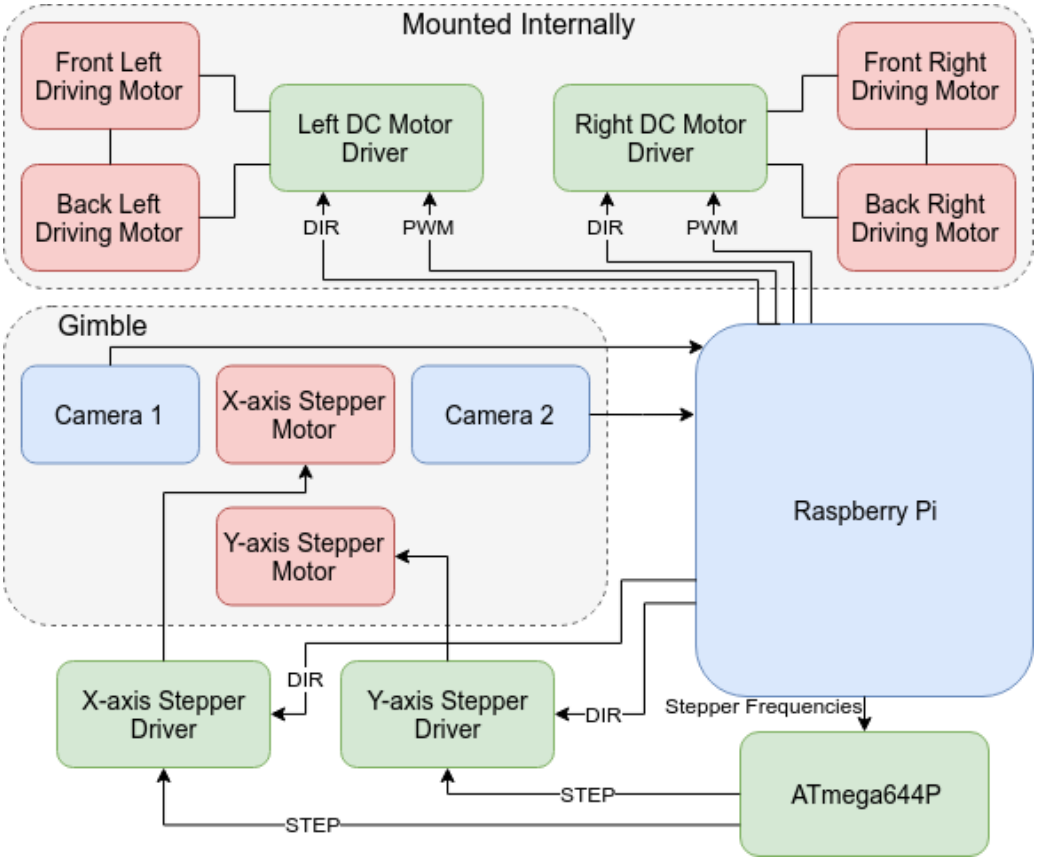


FIGURE 5.2: Hardware Block Diagram. The red blocks are motors, green blocks control system components, and blue blocks parts of the image pipeline.

5.1 Gimbal Design

The choice of cameras had to fulfil a very specific set of requirements. The two cameras must be the same, as any differences in the images due to using different lenses or sensors would reduce the quality of depth map produced on the server. This makes the most obvious camera choice, the R-Pi camera module, unusable, as the R-Pi cannot interface with two simultaneously. The two cameras must also be able to take pictures on command from the R-Pi with low latency, reducing the possible options down to primarily USB webcams. Finally, they must have a high shutter speed. Any motion blur in the images will blur all the edges they contain, making them undetectable by the edge detection algorithm, and any morphing of the image while under motion due to the time it takes the shutter to pass across the entire

sensor will once again reduce the quality of the depth map. A high shutter speed reduces motion blur and shutter related morphing, therefore making it essential for whenever the rover is in motion. This requirement reduces the possible cameras down to primarily dedicated computer vision cameras, however these are expensive and often too large to build a gimbal for without also buying expensive motors.

Drone FPV cameras were tested as an alternative, their low latency and use of a global shutter being ideal for this system, however the analogue to digital converters required to interface them with the R-Pi were too unreliable and low performance to make them a viable option. Only one camera was found that fulfilled all the requirements listed whilst also being reasonably priced and reliable—the PlayStation 3 (PS3) Eye. The PS3 Eye is a peripheral for the PS3 that facilitates games that incorporate aspects of computer vision, so it is designed with computer vision and value for money in mind. While the image quality is fairly poor, this does not significantly impact the quality of the abstractions the system produces due to most detail being discarded as part of the process. The only real flaw in using the PS3 Eye in this system is its poor dynamic range; if the camera is aimed at a window during the day, all the colours in the room will become unrecognisably dark. However, the PS3 Eye has such impressive shutter speed and latency for the price point that this deficiency is an acceptable trade-off.

The design of the gimbal (Figure 5.3) has considerable impact on the 3D model the system generates. As mentioned in Section 2.5.3, the images must be rectified to account for discrepancies in the mounting before the depth map is generated on the server, therefore the closer the cameras are to parallel with each other, the less severe the rectification applied must be. Similarly, the stability of the gimbal is important, as any vibrations will cause inconsistency in the alignment of the cameras, leading to inaccurate depth maps. This led to the chosen design where the X-axis (tilt) motor is located centrally, between the cameras, to balance the weight around the rotational axis of the y-axis (rotation) motor. The 3D-printed part the cameras are attached to is also stabilised through mountings on both sides of the X-axis motor, using a ball bearing on the side not driven by the motor.

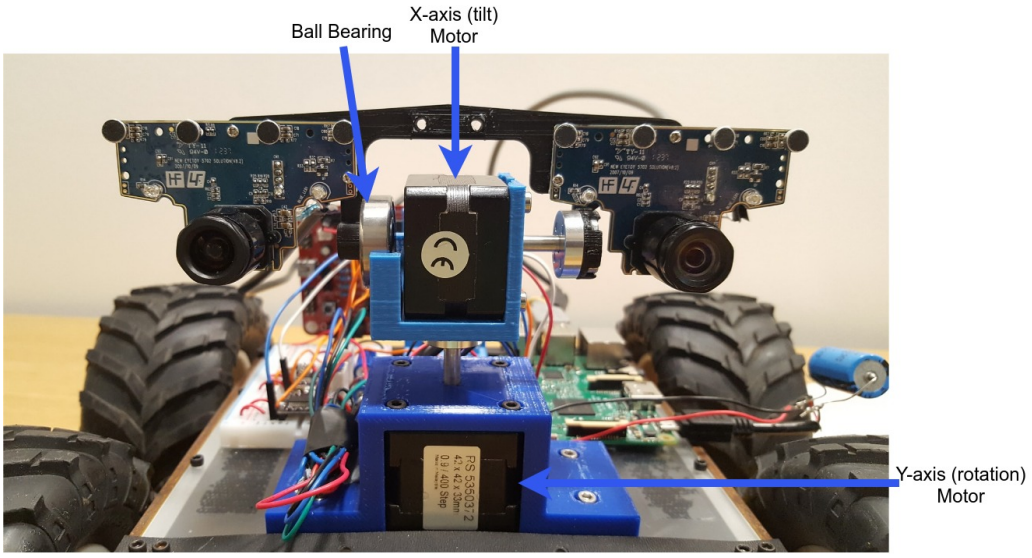


FIGURE 5.3: Gimbal Picture. The Y-axis motor is housed at the bottom, with the X-axis motor directly above it. The ball bearing that stabilises the non-driven side of the cameras' backplate can be seen to the left of the X-axis motor.

Another important aspect of the gimbal design is the distance between the camera lenses. The further apart the two cameras are, the closer distance objects will be in the depth map. Ideally we would aim to match the interpupillary distance of human eyes (63 mm on average [47]), so objects in the 3D environment appear as close as they would were the user standing in the place of the robot. However, with the X-axis motor located centrally it is not possible to mount the cameras that close together. The inter-lens distance in the final design is 120 mm, as this is the closest distance possible without reducing the stability of the gimbal. While not ideal, it simply results in objects appearing closer in the depth map than they actually are, and the distance from the cameras where an object is too close for the depth mapping algorithm to match it between the two images (the cameras are "cross-eyed" if you will) being extended.

5.2 Image Pipeline

As established in Chapter 3, the rover takes a picture with both cameras, abstracts those images, then sends that data off as a single combined packet to the server.

While Chapter 4 discussed the abstraction process as a single step that produces a full abstraction with both edges and spaces filled with colour, this does not reflect the implementation utilized in the full system. The final step of filling the spaces in the edge detected image using the selected seed points and average colours is completed on the server, not on the rover. Also, only one of the two images needs colour information at all, as the edges are the only part of the abstraction required to produce the server depth maps. The colours are to be used as part of a texture applied to the 3D model produced from the depth maps, therefore only one set is required. This results in the data packets consisting of two edge detected images and a set of seed points with corresponding colours for one of the images.

Attempts to implement this process on a single thread on the R-Pi, as tested successfully on a laptop, either crashed the R-Pi or would produce an unacceptable frame rate of around 1 fps. To rectify this, both pipelining and parallelism were utilized to make better use of the Pi's quad core processor (cf. Figure 5.4), and compromises were made in the quality of the abstractions to reduce the workload.

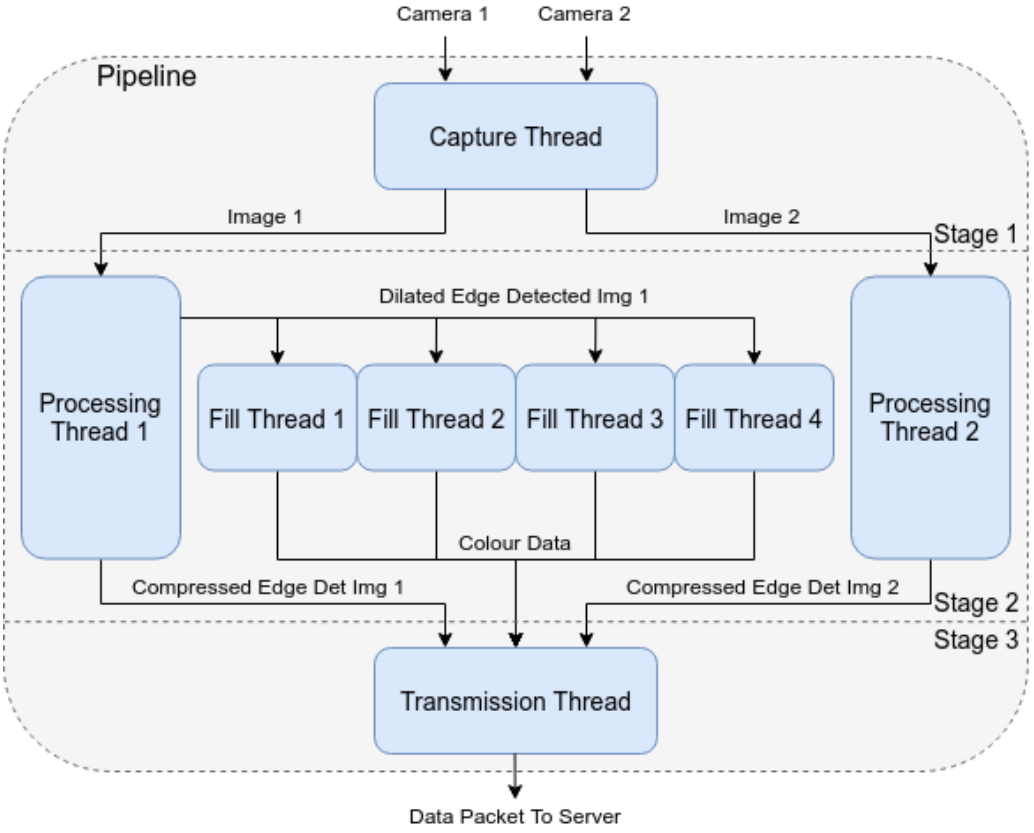


FIGURE 5.4: Raspberry Pi Image Pipeline Threading Block Diagram.

The capture thread handles signalling the cameras to capture and decoding the images. The capturing and decoding are done as separate operations in an effort to capture the images as close to simultaneously as possible in a single thread. The first compromise in quality in favour of performance is made here, where the images are captured with a resolution of 320x240 rather than the cameras' standard resolution of 640x480. As will be discussed in greater depth below, the highest workload task in the pipeline is flood filling. Reducing the pixel count of every space by a factor of four therefore provides a significant improvement in performance. The smaller images also have the benefit of lower detail in high detail areas, so sections that would become areas of dense lines when edge detected (examples of this effect can be found in Appendix A) are significantly less dense, containing less extraneous data.

The processing threads are concerned with the edge detection and compression of the images. The edge detection process is mostly as described in Chapter 4; the only difference is that only the image being sent from *Processing Thread 1* to the fill threads undergoes the final dilation step. The purpose of the dilation is to bridge gaps between the edges, creating defined shapes for the flood fill process. A side effect of dilation is a reduction in edge accuracy, which is very important when the images are to be used to produce depth maps later on. This leads to the logical conclusion that the images should be sent without dilation, and dilation only applied to find the colour data in the fill threads and then again on the server to create the complete coloured abstraction, allowing the depth maps to be constructed from non-dilated edges.

While it can be induced from comparing the original images to their edge detected versions by eye that the latter contains less information than the former, this will only be reflected in real numbers if file format and compression are considered carefully. When the common image formats are compared, PNG would appear to be effective for this use case [48], as it excels at efficiently storing large blocks of the same colour (most of each edge detected image is black space). When tested on the edge detected images (Figure 5.5), it was confirmed that PNG provides the lowest file sizes, and compression level 5 provided the best ratio between file size and compression time (using the imencode function in OpenCV). More intelligent compression was tested using libimagequant [49], however it resulted in poor performance (<2 fps) with negligible improvement to the level of compression. An alternate option to

compression as a bitmap was vectorising the images, however this was found to be inaccurate and not as effective as bitmap compression (more information can be found in Appendix E).

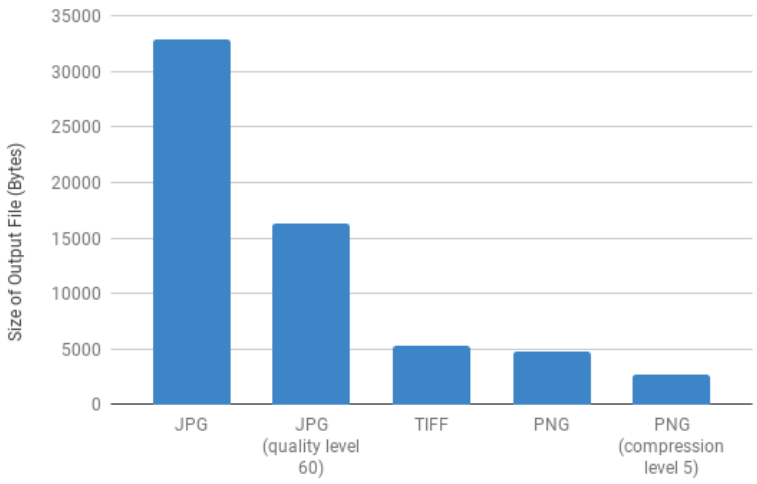


FIGURE 5.5: Comparison of Image File Formats. This data shows that PNG provides the smallest file sizes of the readily available file formats. It also shows that both standard JPG and PNG compression are effective at reducing file size, however attempting to compress the JPG versions enough to compete with PNG would result in images of unusably poor quality (quality level 5-10).

The stage in the image pipeline that has the highest performance impact is the colour averaging, due to its use of flood filling. As the number of spaces being flood filled is determined by the number of points from the Sobol sequence that hit unfilled spaces, the number of Sobol points used is an important variable in tuning the performance of the system. In the testing done on a laptop in Chapter 4, the number of Sobol points being used per image was between 400 and 600, and this filled a reasonable number of spaces while providing a reasonable frame rate. When this was attempted on the R-Pi however, the frame rate was below 1 fps. This called for a significant reduction in Sobol points and the application of parallelism. Good performance was achieved using 4 threads dedicated to the flood fill colour averaging task, each taking 50 Sobol points. A total of 200 Sobol points is a significant reduction from the laptop implementation of the algorithm, however it must also be noted that the images being used are much lower detail, so do not require a large number of Sobol points to be mostly covered (Figure 5.6).

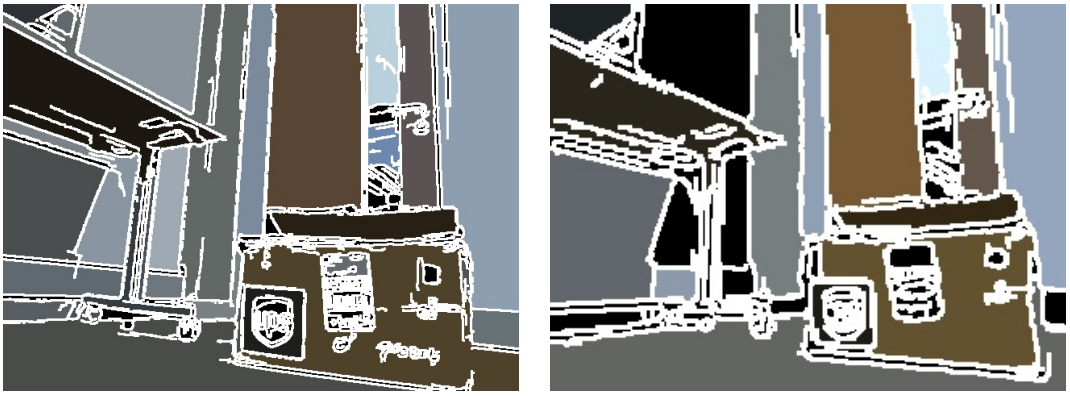


FIGURE 5.6: Comparison of Laptop and Rover Abstraction. The laptop abstraction (left) is high resolution, with only the smallest of spaces remaining unfilled such as the detail on the box stickers. The rover abstraction (right, this image being after the colour data and edge detected image had been combined on the server) is much lower resolution, with many more spaces remaining unfilled. However, it is still clearly the same scene with the geometry represented with almost the same level of accuracy, therefore making it an acceptable abstraction.

While it may seem as though the parallelization of this process could cause issues if two threads attempt to fill the same space in the image simultaneously, this simply results in two successful seed points with slightly inaccurate average colours assigned to them, and only one of them being used to create the complete abstraction on the server. As the colours of the spaces are only recorded to provide the user a better understanding of the objects in the environment, it is not concerning if occasionally one is not an exact average.

Once the three pieces of data for an image pair (two compressed edge detected images and a set of colour data) have been generated, they must be combined into a single packet and sent to the server; this is covered by the transmission thread. The communications method chosen for this was the UDP (User Datagram Protocol) Wi-Fi communications standard [42]; this is due to the reliability of Wi-Fi over a large variance of distance, and the speed and efficiency of UDP. While the images are at this stage already compressed into a set of bytes that can be transmitted as is, the colour data needs its own custom packaging. Each seed point-average colour pair is formed into its own sub-packet with the structure shown in Table 5.1. These sub-packets make up the colour data data segment within the complete data packet.

TABLE 5.1: Colour Data Sub-Packet Structure. The seed point components are given 2 bytes each to allow the transmission of colour data for images larger than 255x255.

Byte in Colour Data Sub-Packet	1	2	3	4	5	6	7
Usage	Seed Point				Average Colour		
Component	X		Y		Red	Green	Blue

As the size of each data segment is unknown and highly variable, knowing where one ends and another begins on the server is a challenge. This has been solved by storing the number of colour data sub-packets (in a single byte) at the start of the packet, allowing the length of the first image (stored in two bytes) to be located at the end of the colour data. With the length of the first image, its end and the start of the second image can then be easily located. This therefore leads to the data packet format shown in Table 5.2.

TABLE 5.2: Data Packet Format.

Data Segment	Colour Data Length	Colour Data			Img 1 Length	Img 1	Img2
Contents	Number of Sub-Packets	Sub-Packet 1	Sub-Packet 2	...	Number of Bytes	PNG Data	PNG Data

5.3 Control System

The primary aim when designing the control system was to make controlling the rover easy and intuitive; the user must have no trouble understanding the controls whether in or out of VR. The Xbox 360 controller can be integrated into almost any system, is intuitive for even non-gamers to operate, and can be used both with and without a VR headset, leading it to be selected as the control input method.

The motion of the rover is restricted to forwards or backwards with variable speed, or rotation with a fixed speed. This is so the rover is forced to only rotate in short intervals, as rotation causes a reduction in depth map accuracy that forward and backwards motion does not (further discussed in Chapter 7). The user has the ability to chose between full and half speed drive modes, to provide the flexibility of both being able to run the motors at full power in one mode and drive at low speeds with greater control in the other. Due to the gimbal not having a full range of motion

(it can rotate roughly 200 degrees before it starts to pull its own cabling out), its range must be restricted in software. This is done by keeping track of the number of steps the stepper motors have undertaken as an estimator of the gimbal's orientation and preventing it from moving past certain angles. This system is effective, however the limits drift over time due to the stepper motors not always moving through the exact same number of steps requested by the control system. This drift, while not ideal, is only an issue over long activation periods, so is not a major concern for this project. The final element the user has control over is the low threshold in the edge detection applied as part of the data abstraction (this is the single point of contact between the image pipeline and control system that is mentioned in Chapter 3). Changing the low threshold of the edge detection changes the amount of detail in the 3D model, having a major impact on its quality. Additionally, the threshold level that produces the optimal level of 3D model detail changes depending on the space the rover is observing, therefore it is essential that the user has control of this at run-time.

The user inputs are transmitted to the rover over UDP, each packet with the structure shown in Table 5.3. The packets are sent at a constant rate of 50 per second, even when the user is not inputting any commands. This is so that in the event that the wireless link is failing, the inconsistency of the constant message stream will notify the rover that there is an issue. When the rover detects that it has been an unreasonable amount of time since the last message, it stops all its motors until it starts receiving messages again, to prevent it from damaging itself while the user has lost control.

TABLE 5.3: Control Data Packet Structure. The analogue stick axes are mapped to between 0 and 255 to fit them within a byte each, as greater accuracy than that is not required. The low threshold only being allocated a byte means it is also capped at 255, however through testing it has been determined that values above 100 produce images with no edges anyway, so this limit is not a concern.

Byte in Control Packet	0	1	2	3	4	5
Usage	Driving Control (Left Analogue Stick)		Low Threshold	Gimbal Control (Right Analogue Stick)		Drive Mode
Contents	X-axis	Y-axis	Threshold Value	X-axis	Y-axis	0/1

Chapter 6

Server-Side Implementation

The purpose of the server is to receive image data from the rover, produce a 3D environment from this data, and feed control inputs from the user back to the rover. This is all done within the framework of the 3D game engine Unreal Engine 4 [50]. Unreal 4 was chosen because it supports almost any VR headset on the market, is simple to integrate with OpenCV, and is free to use. The structure of any Unreal program is complex, as the engine utilises multithreading heavily to implement complicated lighting and rendering pipelines. The implementation of this project within the Unreal environment (cf. Figure 6.1) consists of a receive thread that receives images, processes them, then updates the 3D model, and a modification to the game thread (the Unreal thread that handles game logic, such as physics and collisions) that handles and sends user input to the rover 20 times a second. The fact that the 3D model is updated on a thread separate to the rendering of the images that are sent to the headset is what allows the headset to run at an unrelated and significantly faster frame rate than the model is being updated at.

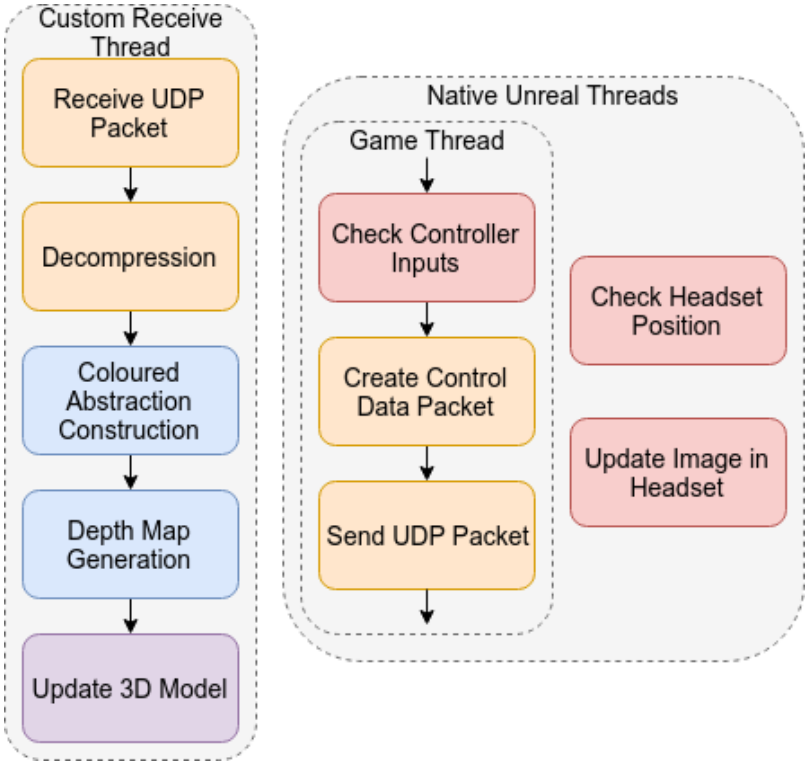


FIGURE 6.1: Server Program Structure. Red blocks are standard Unreal functionality, yellow blocks are communications related, blue blocks are OpenCV based, and the purple block is custom integration with the Unreal environment. The blocks within the game thread run 20 times a second.

6.1 Depth Map and Abstraction Construction

For a depth mapping algorithm to function accurately, the 2 images it receives must be rectified (established in Section 2.5.3). The rectification parameters for the cameras in our system have been calculated using a set of programs provided by Sourish Ghost [51]. The rectification is applied using these parameters just before the depth maps are calculated (Figure 6.2).

FIGURE 6.2: Demonstration of Stereo Image Rectification. The original left and right images are shown top left and top right, with their rectified counterparts bottom left and bottom right. It can be seen that the features of the bottom images are better matched in the Y-axis than in the original images.

When selecting a depth mapping algorithm, the major factors to consider were speed and performance with abstracted images. The use of abstractions makes demonstrations of the algorithms with normal images of little help; there is no guarantee that the algorithms would be able to map depth for edge detected lines with the same accuracy, if at all, since this is not what they were designed for. Three different depth mapping algorithms were tested: StereoBM, StereoSGBM, and Libelas. StereoBM and StereoSGBM are a block matching and semi block matching algorithm provided by OpenCV [30], and Libelas is a more complex algorithm provided by the Autonomous Vision Group [52]. Of the three algorithms presented, StereoBM is the fastest and Libelas the slowest (Figure 6.3).

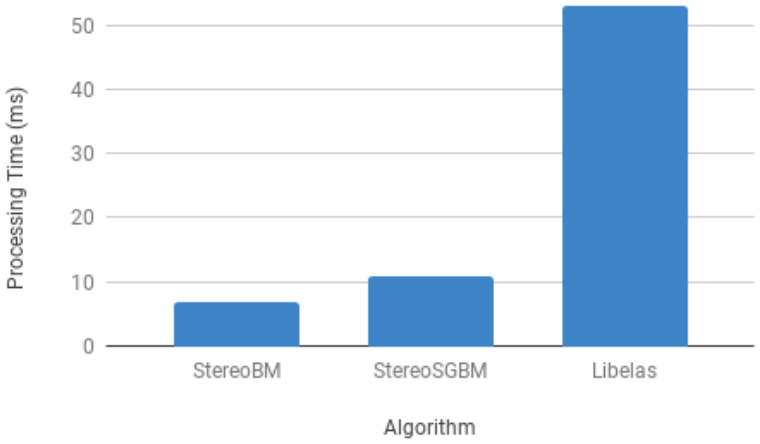


FIGURE 6.3: Comparison of Depth Mapping Algorithm Speed. The test images used to produce this data were the rectified images presented in Figure 6.2. These same images were used in the production of all further figures in this section.

When tested on images received from the rover, StereoBM and StereoSGBM produced noisy results, yet, if inspected closely, results that include reasonable depth information along the areas where edges were in the original images (Figure 6.4). Libelas, however, did not produce results that were in any way recognisable. Combining this with its significantly slower processing speed, it is clear that Libelas is not that correct choice for this use case.

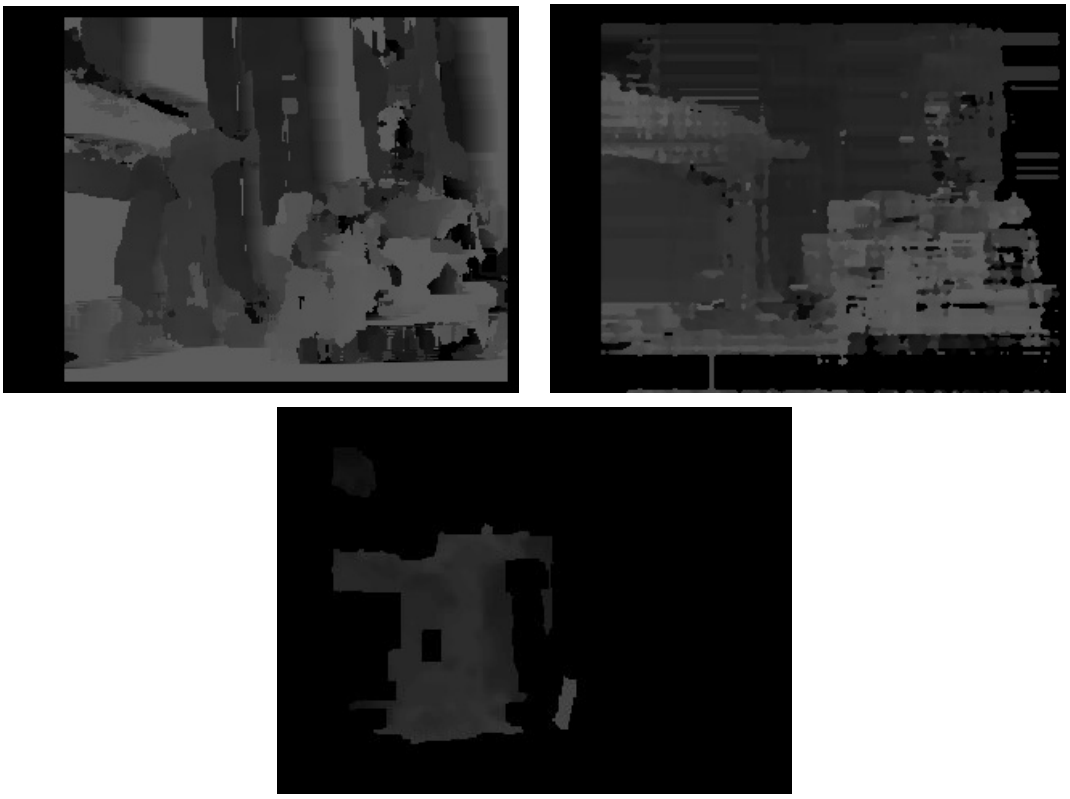


FIGURE 6.4: Comparison of Raw Depth Mapping Algorithm Output. The StereoBM output (top left) is very noisy, however correct depth can be seen on the box in the bottom right and on the table in the top left (when compared to the input images in Figure 6.2). The StereoSGBM output (top right) shows similar results, though with a significant reduction in noise. The Libelas output (bottom) shows very little—it does not seem to work with edge detected images.

To be able to compare StereoBM and StereoSGBM further, the noise their outputs contain must be filtered out. This can be achieved by applying Weighted Least Squares filtering to the depth map (Figure 6.5). This is effective in producing consistent depth across the objects in the map. Unfortunately, regardless of the angle to the camera that an object sits at, the filtering will often assign it a single depth across the whole surface. This results in surfaces like the floor or the ceiling being inaccurately represented as a vertical plane close to the camera, however most other objects are not significantly affected.

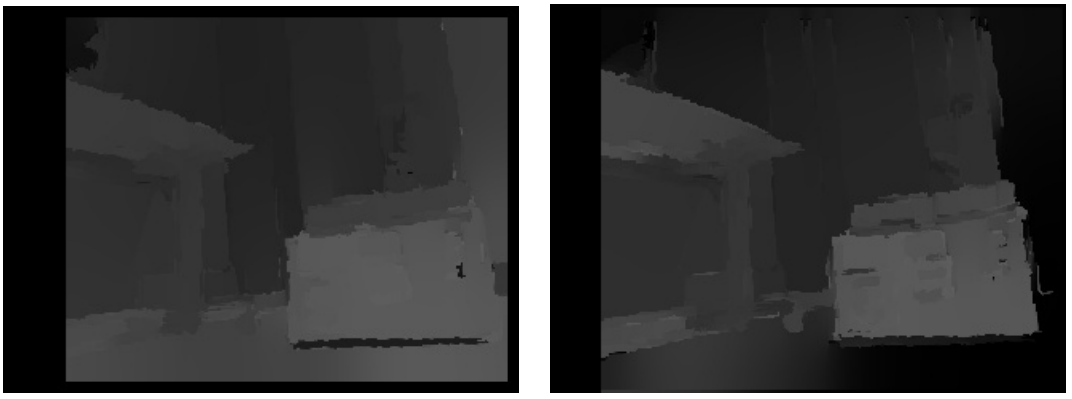


FIGURE 6.5: Filtered StereoBM and StereoSGBM Outputs. The StereoBM output (left) has been improved significantly by the filtering, with mostly consistent and correct depth across all objects other than the floor. StereoSGBM (right) has been similarly cleaned up, however has not provided depth for the floor or right wall. The single section of floor it attempted (under the table in the bottom left) is the same incorrect depth as produced by StereoBM.

When comparing the filtered maps, there are few differences. Both are reasonable representations of the original scene, and when one makes a mistake in the depth of an object, the other will probably make the exact same mistake. The only major difference is that StereoBM will consider the floor incorrectly as a vertical plane close to the camera, whereas StereoSGBM will incorrectly not record a depth for it at all. This lack of a significant gain in quality when using the noticeably slower StereoSGBM led to StereoBM being chosen as the algorithm used in the final system.

While the individual depth maps (post filtering) are reasonable approximations of the space being viewed, the depth of a single object with be inconsistent between frames, due to inconsistencies between the edge detected frames they were produced from. When viewed as a video feed, most objects will oscillate slightly, with greater oscillations in objects that are more ambiguous or less consistent in the edge detected frames. This issue is mitigated in the system by taking a running average, the weighting of the frames in the average reducing with the age of the frame. This increases consistency substantially at the expense of a minor reduction in responsiveness.

The construction of the coloured abstraction is as addressed in Chapter 4. The seed point/average colour pairs are each used to flood fill *Edge Detected Img 1* from Figure

3.1, producing the full abstraction that is to be applied as a texture to the 3D model generated from the depth maps.

6.2 3D Model Generation

To generate a 3D model dynamically based on image data, an Unreal Engine 4 class that allows for mesh generation at run-time is required. Although more intelligent options are available as add-ons, the core Unreal 4 class *Procedural Mesh* was selected due to its simplicity and compatibility with OpenCV. The class is used as a basis for the generation of a grid based mesh (Figure 6.6).

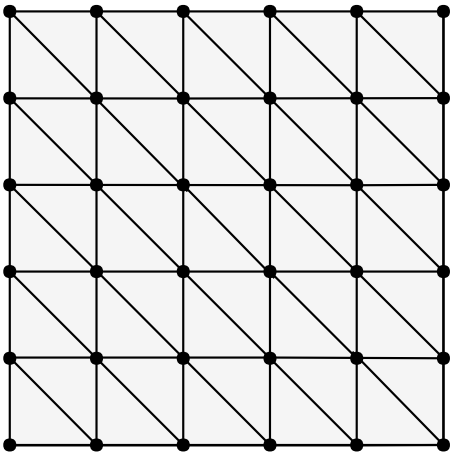


FIGURE 6.6: Mesh Grid Segment. Each corner of the grid is a vertex, with triangles formed from the vertices. The grid is built in the X-Y plane, leaving the Z component of each vertex available to be easily assigned depth.

Each pixel in the depth map is mapped to a vertex in the mesh, resulting in a grid of 320x240 vertices for 320x240 pixel images. The colour value of each depth map pixel is mapped to the Z component of its corresponding vertex, leading to the bright pixels in the depth map (close objects) generating vertices that are pushed forward from the grid. When this grid is placed an appropriately realistic distance from the the user in the headset, the close objects in the depth map will be pushed closer to the user and the further objects will remain closer to the grid, further from the user. When the full coloured abstraction is then applied to the mesh as a texture,

the pixels once again mapping 1:1 to the vertices, the space the rover is observing is finally being portrayed in VR (Figure 6.7).

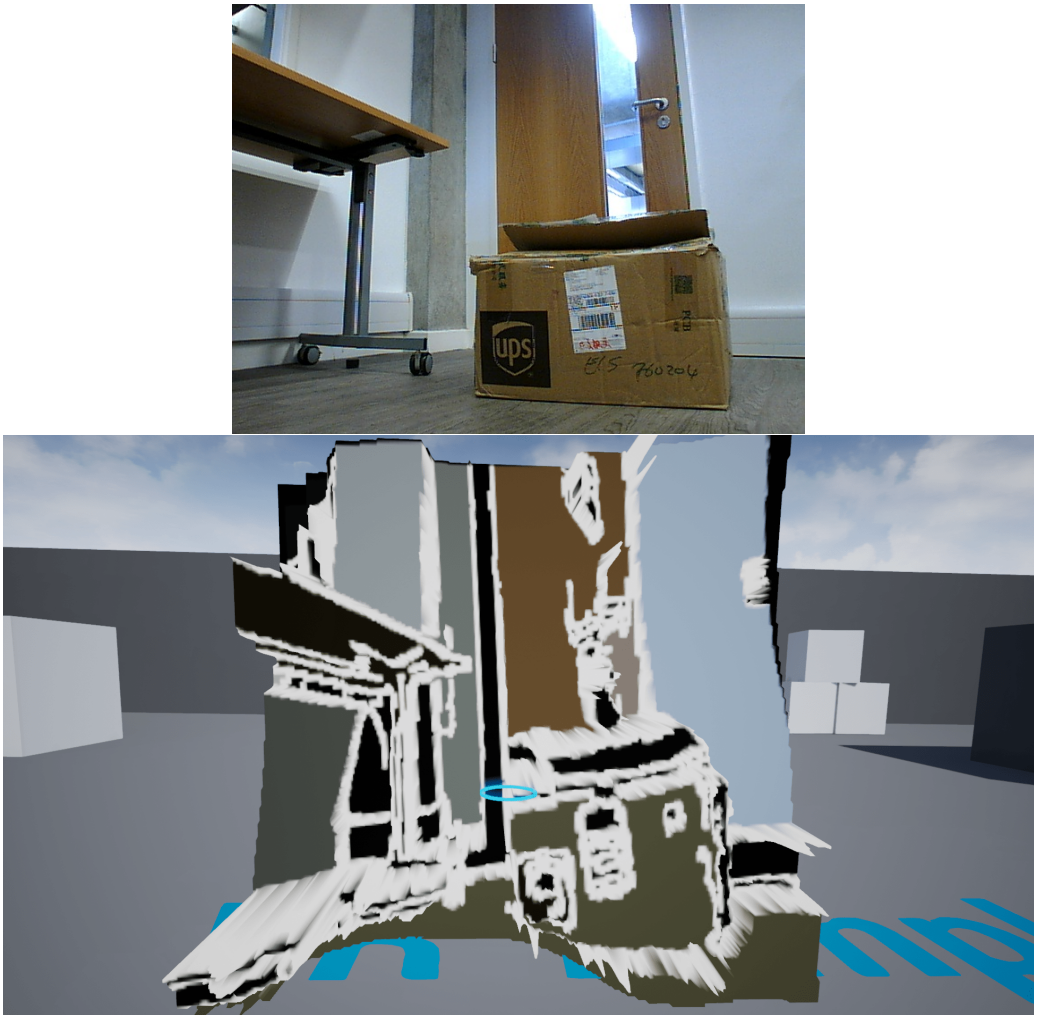


FIGURE 6.7: Final Textured 3D Model. The blue ellipse is a UI element that indicates a location that the user can jump to within the environment—not part of the 3D model.

Chapter 7

Testing and Evaluation

For the purposes of fairly evaluating this system, three different aspects of it were tested: the quality of its 3D representation as an abstraction of the rover's space, its real-time performance, and its bandwidth compared to a non-abstraction based system. It should be noted that none of these are a direct evaluation of the presence the system provides, as the presence of a system is a fairly subjective assessment that would require studies of large user groups to evaluate with any accuracy. While this prevents direct evaluation, the potential for presence can be indirectly evaluated through the aspects that were tested.

7.1 3D Model Quality

The system's ability to produce recognisable representations of the rover's environment was tested in a lab environment, with encouraging results. Most objects were recognisable from their abstracted representations and placed at the correct depth in the scene (Figure 7.1).

FIGURE 7.1: Full System Demonstrations. The left images are of test scenes, the images taken at the resolution of the rover's cameras, and the right images are of the 3D representations of the test scenes. The scenes are clearly recognisable, with most objects at their correct depths such as the boxes and bag being at the front in the bottom right image. The issues with the system can also be seen, with the errors and odd floor position in the top right image.

It was found that objects with simple shapes, such as boxes, were represented with high accuracy, however the more complex the shape the less accurate the representation became. This is because the system does not detect changes in depth across the face of an object very well—depth is only calculated from the edges. This results in every object being represented as mostly flat, however, as predicted in Chapter 4, the user only needs the outline and average colour of an object to be able to recognise it, so object recognition is not significantly affected.

The issue of the floor being represented incorrectly in the depth map (discussed in Section 6.1) remains an issue in the 3D model. For example, the dark grey section of

the floor in the top right image of Figure 7.1 should extend all the way to the back of the scene, instead it has been pushed forward as a large block at around ankle height. While it is most noticeable in the floor, the problem is actually a general inability to correctly produce depth maps of slanted surfaces from edge detected images—every surface in the scene will be represented as perpendicular to the camera. Due to this issue being based in the complexities of edge detection based depth mapping, solving it would require significant reworking of the mapping algorithm and filtering technique. However, it is also an issue that does not significantly impact the user under normal usage unless they explicitly attempt to look at the floor.

The other issue noted at the depth mapping stage, the mistakes the algorithm makes with certain objects, also remains an issue in the 3D model. If an object is very thin, or contains parallel edges in close proximity to each other, it will be incorrectly placed significantly closer to the camera (Figure 7.2).

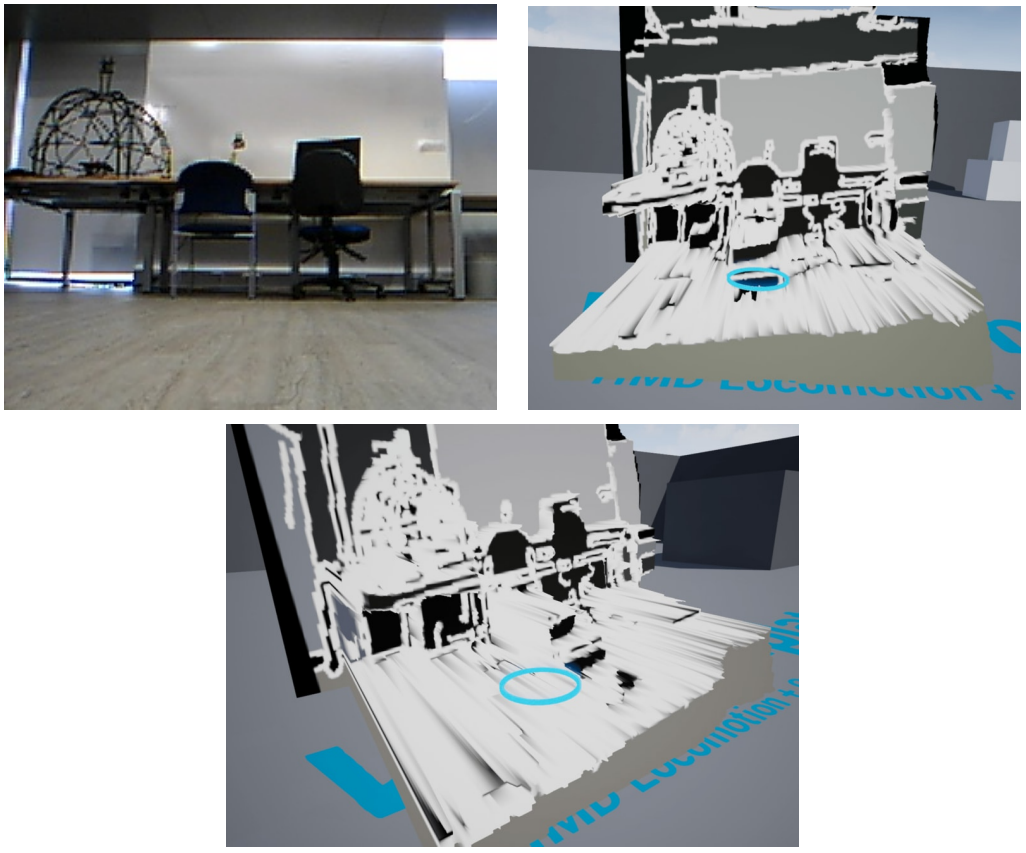


FIGURE 7.2: Demonstration of System Errors. The scene (top left) is very difficult for the system to reproduce, as it consists of thin, poorly lit objects. While the reproduction is recognisable, there are clear errors such as under the left chair and the left side of the table.

The errors are a by-product of using small images, as this results in thin objects often only being 5 or less pixels thick. With so few pixels representing them, its unsurprising these objects are not accurately placed in the depth map. The solution to this would be to use higher resolution images. This is not possible within the performance constraints of the R-Pi 3, though in a future iteration of the system where the R-Pi could be replaced with a higher performance option, it becomes entirely achievable.

7.2 Run-Time Performance

As intended, the VR headset maintains 90fps and low latency regardless of how the rest of the system is performing. It is therefore unlikely that the user would feel motion sickness while using the system (no-one that tried it did), meaning the issue that prevents high levels of presence in most VR teleoperations systems has been successfully tackled. The rate the 3D model updates at is fairly consistently between 20 and 25fps, and the latency of the system is generally between 0.25 and 0.5 seconds. The frame rate and latency are entirely dictated by the processing power of the rover, as this is the bottleneck of the system, therefore replacing the R-Pi 3 with a more powerful device would significantly improve these results.

At the present level of performance, controlling the rover through the 3D model is intuitive and comfortable. The latency does affect the user's ability to move the rover with precision, though not to a significant degree. The 3D model remains accurate regardless of the speed the rover moves forward or backwards at, however turning does have an impact on depth map accuracy. This is not a serious issue though, as the coloured abstraction texture remains accurate, even if the depth map does not, resulting in the user's awareness of the environment being largely unaffected.

7.3 Comparison to a Non-Abstraction System

The final aspect that required testing is whether the system has lower bandwidth requirements than similar systems that do not use data abstraction. A test non-abstraction system was build that sends 2 normal images over UDP, then attempts to generate a depth map from them. The lowest packet size that this system could produce while still generating depth maps comparable to the abstracted system was discerned by varying the level of JPEG compression applied to the images (Figure 7.3).

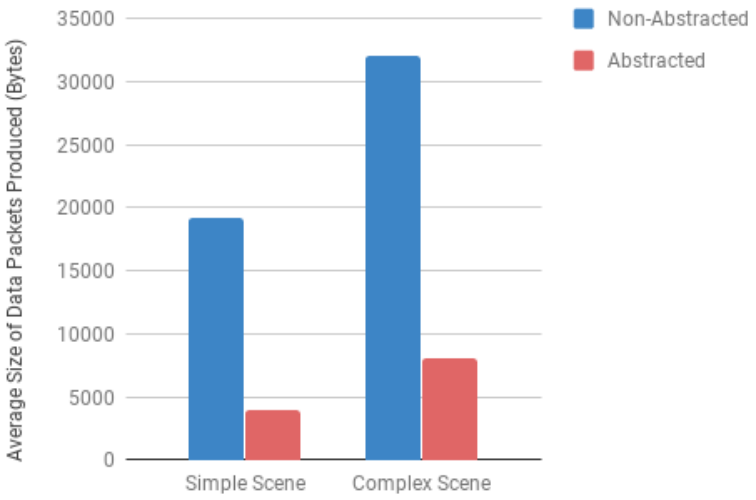


FIGURE 7.3: Comparison of Packet Size for Reliable Depth Maps. JPEG Compression was decreased until the non-abstraction system produced depth maps of comparable accuracy to the abstraction system.

Attempting to reduce the packet size of the non-abstraction system to the maximum packet size observed from the main system results in depth maps that do not contain any useful information (Figure 7.4), leading to the conclusion that the system presented by this report does provide a significant reduction in the bandwidth requirements for VR teleoperations.

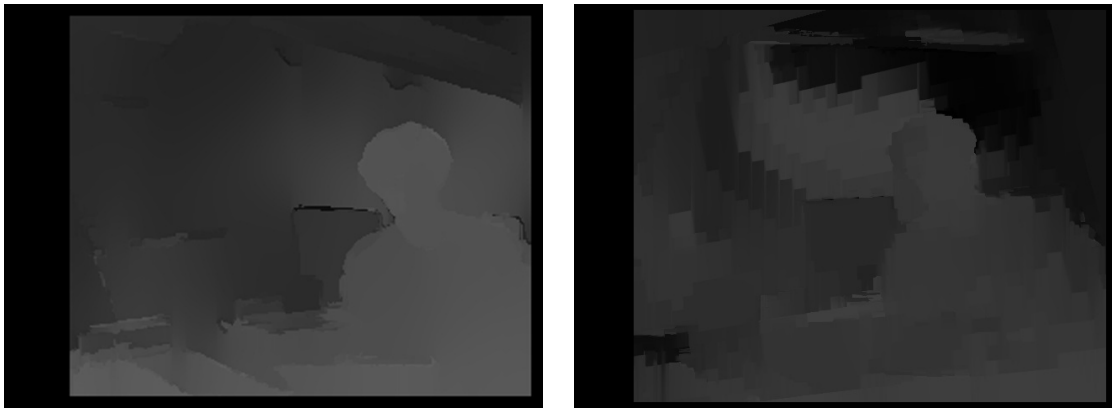


FIGURE 7.4: Non-Abstraction Depth Map Comparison at Equal Packet Size. At a packet size of roughly 5800 bytes, the abstraction system (left) produces clearly defined and accurate shapes with mostly correct depths, whereas the non-abstraction system (right) produces mostly noise, with little relation to the original scene.

Chapter 8

Conclusion

The aim of this project was to research the use of data abstraction to minimise the technical issues involved with VR telerobotics through the design and implementation of an abstraction based teleoperations system. Through its use of a 3D model, the system created does not cause motion sickness when operated, successfully mitigating the primary issue with VR telerobotics. The 3D model updates at 20-25 fps with a latency of 0.25-0.5 seconds, providing the user reasonably responsive controls, and is also accurate enough to provide the user spacial awareness. The novel data abstraction algorithm implemented allows this to be achieved with data packets of <8kB each.

The next steps for this research would be to replace the R-Pi with a more powerful alternative, with the aim of mitigating the issues with the system that are linked to the R-Pi's bottlenecking, and to improve the system's ability to represent slanted surfaces. As the depth information required to correctly map a slanted surface is already present in the edge detected images of it, it is the depth mapping and depth map filtering that must be redesigned to better extract that information.

Chapter 9

Project Management

A comparison of the original project brief (Appendix F) to the final system reveals some evolution in the scope of this project. The one significant change is the removal of inter-frame interpolation from the design. This decision was made in response to the effectiveness of the system in preventing motion sickness without interpolation, therefore making its addition of no merit. This change is also reflected in the comparison of the work plan produced for the interim report and the actual time frames work was completed in (Appendix G). Not only was the inter-frame interpolation component removed, but a complex image pipeline incorporated, shifting the planning considerably. Comparing the shifted timings, the other components were all well planned and were completed roughly within the time frames expected. The risk assessment produced for the interim report was accurate to the risks the project faced, and can be found with comparisons to the actual events that occurred in Appendix H.

Prior to working on this project, I had some experience of robotics, using Unreal 4 and working with the HTC Vive, and more considerable experience of embedded programming on the Raspberry Pi. To complete a project that is heavily based on these fields therefore required a substantially increase in my competency in them. Furthermore, I had no experience of computer vision, OpenCV, or custom wireless communications, so built my knowledge of these fields completely from scratch for the purposes of this project.

Bibliography

[1] A. L. Alexander, T. Brunyé, J. Sidman, and S. A. Weil, "From gaming to training: A review of studies on fidelity, immersion, presence, and buy-in and their effects on transfer in pc-based simulations and games," *DARWARS Training Impact Group*, vol. 5, pp. 1–14, 2005.

[2] K. M. Lee, "Presence, explicated," *Communication Theory*, vol. 14, no. 1, pp. 27–50, 2004.

[3] J. M. Loomis, "Presence in virtual reality and everyday life: Immersion within a world of representation," *PRESENCE: Teleoperators and Virtual Environments*, vol. 25, no. 2, pp. 169–174, 2016.

[4] S. A. McGlynn and W. A. Rogers, "Considerations for presence in teleoperation," in *Proceedings of the Companion of the 2017 ACM/IEEE International Conference on Human-Robot Interaction*, HRI '17, (New York, NY, USA), pp. 203–204, ACM, 2017.

[5] P.-H. Han, Y.-S. Chen, H.-L. Wang, Y.-J. Huang, J.-C. Hsiao, K.-W. Chen, and Y.-P. Hung, "The design of video see-through window for manipulating physical object with head-mounted display," in *ACM SIGGRAPH 2017 Posters*, p. 31, ACM, 2017.

[6] IrisVR, "The importance of frame rates," 2017. Available: `https://help.irisvr.com/hc/en-us/articles/215884547-The-Importance-of-Frame-Rates`. [Accessed: April 2018].

[7] M. Borg, S. S. Johansen, K. S. Krog, D. L. Thomsen, and M. Kraus, "Using a graphics turing test to evaluate the effect of frame rate and motion blur on telepresence of animated objects," in *GRAPP/IVAPP*, 2013.

[8] P. Zachares, E. Tanirgan, D. Sibanda, and J. Nappo, "DORA," 2015. Available: `http://doraplatform.com/`. [Accessed: April 2018].

[9] M. Gupta, F. Hennecker, T. Isaacs, M. Sharhan, and L. Stauskis, "Biologically inspired robots: A vision system inspired by the human eye," 2016. ELEC6212 Report, Electronics and Computer Science, University of Southampton.

[10] Microsoft, "Windows Official Site," 2018. Available: `https://www.microsoft.com/en-gb/windows/`. [Accessed: April 2018].

[11] HTC, "HTC Vive Homepage," 2017. Available: `https://www.vive.com/uk/`. [Accessed: April 2018].

[12] Virtual Reality Society, "What is virtual reality?," 2017. Available: `https://www.vrs.org.uk/virtual-reality/what-is-virtual-reality.html`. [Accessed: April 2018].

[13] Oculus, "Oculus Rift Homepage," 2017. Available: `https://www.oculus.com/rift/`. [Accessed: April 2018].

[14] T. B. Sheridan, "Telerobotics," *Automatica*, vol. 25, no. 4, pp. 487–507, 1989.

[15] T. Fong, "Interactive exploration robots: Human-robotic collaboration and interactions," 2017.

[16] V. A. Huvenne, K. Robert, L. Marsh, C. L. Iacono, T. Le Bas, and R. B. Wynn, "Rovs and auvs," in *Submarine Geomorphology*, pp. 93–108, Springer, 2018.

[17] C. Smith and A. Lightman, "Radiation hardened telerobotic dismantling system development final report crada no. tc-1340-96," tech. rep., Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2017.

[18] NASA, "Robonaut 2 homepage," 2016. Available: `https://robonaut.jsc.nasa.gov/R2/`. [Accessed: April 2018].

[19] RCExplorer, "FPV Starting Guide," 2009. Available: `https://rcexplorer.se/educational/2009/09/fpv-starting-guide/`. [Accessed: April 2018].

[20] N. Smolyanskiy and M. Gonzalez-Franco, "Stereoscopic first person view system for drone navigation," *Frontiers in Robotics and AI*, vol. 4, p. 11, 2017.

[21] J. I. Lipton, A. J. Fay, and D. Rus, "Baxter's homunculus: Virtual reality spaces for teleoperation in manufacturing," *IEEE Robotics and Automation Letters*, vol. 3, no. 1, pp. 179–186, 2018.

[22] P. Wang, J. Xiao, H. Lu, H. Zhang, R. Yan, and S. Hong, "A novel human-robot interaction system based on 3d mapping and virtual reality," in *Chinese Automation Congress (CAC), 2017*, pp. 5888–5894, IEEE, 2017.

[23] K. Sayood, *Introduction to data compression*. Elsevier, 2005.

[24] A. Barrett-Sprot, "Data abstraction for low-bandwidth communication," 2017. BEng Project Report, Electronics and Computer Science, University of Southampton, Available: `https://secure.ecs.soton.ac.uk/notes/comp3200/e_archive/COMP3200/1617/abs1g14/pdfs/Report.pdf`. [Accessed: April 2018].

[25] S. Joe and F. Y. Kuo, "Constructing sobol sequences with better two-dimensional projections," *SIAM Journal on Scientific Computing*, vol. 30, no. 5, pp. 2635–2654, 2008.

[26] L. Grünschloß, "gruenscholoss.org," 2017. Available: `http://gruenschloss.org/`. [Accessed: April 2018].

[27] Wikipedia, "Sobol sequence," 2017. Available: `https://en.wikipedia.org/wiki/Sobol_sequence`. [Accessed: April 2018].

[28] BMVA.org, "What is computer vision?," 2017. Available: `http://www.bmva.org/visionoverview`. [Accessed: April 2018].

[29] D. H. Ballard and C. M. Brown, "Computer vision," *Prenice-Hall, Englewood Cliffs, NJ*, 1982.

[30] OpenCV team, "OpenCV," 2017. Available: `https://opencv.org/`. [Accessed: April 2018].

[31] R. Jain, R. Kasturi, and B. G. Schunck, *Machine vision*, vol. 5. McGraw-Hill New York, 1995.

[32] R. Maini and H. Aggarwal, "Study and comparison of various image edge detection techniques," *International journal of image processing (IJIP)*, vol. 3, no. 1, pp. 1–11, 2009.

[33] J. Canny, "A computational approach to edge detection," *IEEE Transactions on pattern analysis and machine intelligence*, no. 6, pp. 679–698, 1986.

[34] A. Mordvintsev and K. Abid, "OpenCV Canny Edge Detection Tutorial," 2013. Available: `http://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_canny/py_canny.html`. [Accessed: April 2018].

[35] V. Jaimini, "Hackerearth.com flood fill tutorial," 2017. Available: `https://www.hackerearth.com/practice/algorithms/graphs/flood-fill-algorithm/tutorial/`. [Accessed: April 2018].

[36] E.-M. Nosal, "Flood-fill algorithms used for passive acoustic detection and tracking," in *New Trends for Environmental Monitoring Using Passive Systems, 2008*, pp. 1–5, IEEE, 2008.

[37] N. Qian, "Binocular disparity and the perception of depth," *Neuron*, vol. 18, no. 3, pp. 359–368, 1997.

[38] L. Shapiro and G. Stockman, *Computer vision*. Prentice Hall, Upper Saddle River, NJ, 2001.

[39] OpenCV Dev Team, "Camera calibration," 2014. Available: `https://docs.opencv.org/3.0-beta/doc/py_tutorials/py_calib3d/py_calibration/py_calibration.html#calibration`. [Accessed: April 2018].

[40] S. Ghost, "Stereo calibration using C++ and OpenCV," 2016. Available: `http://sourishghosh.com/2016/stereo-calibration-cpp-opencv/`. [Accessed: April 2018].

[41] L. Alvarez, R. Deriche, J. Snchez, and J. Weickert, "Dense Disparity Map Estimation Respecting Image Discontinuities," 2000. Available: `http://serdis.dis.ulpgc.es/~lalvarez/research/demos/StereoFlow/index.html`. [Accessed: April 2018].

[42] J. Postel, "User datagram protocol," tech. rep., 1980.

[43] Microsoft, "Xbox 360 Controller for Windows," 2018. Available: https://www.microsoft.com/accessories/en-gb/products/gaming/xbox-360-controller-for-windows/52a-00004. [Accessed: April 2018].

[44] OpenCV Documentation, "OpenCV: ffilldemo.cpp," 2017. Available: https://docs.opencv.org/3.3.0/d5/d26/ffilldemo_8cpp-example.html. [Accessed: April 2018].

[45] G. Bradski and A. Kaehler, *Learning OpenCV: Computer vision with the OpenCV library.* " O'Reilly Media, Inc.", 2008.

[46] Raspberry Pi Foundation, "Raspberry Pi Homepage," 2018. Available: https://www.raspberrypi.org/. [Accessed: April 2018].

[47] N. A. Dodgson, "Variation and extrema of human interpupillary distance," in *Stereoscopic Displays and Virtual Reality Systems XI*, vol. 5291, pp. 36–47, International Society for Optics and Photonics, 2004.

[48] P. Aguilera, "Comparison of different image compression formats," *ECE*, vol. 533, 2006.

[49] K. Lesiski, "libimagequant—Image Quantization Library," 2017. Available: https://pngquant.org/lib/. [Accessed: April 2018].

[50] Epic Games, "What is Unreal Engine 4?," 2018. Available: https://www.unrealengine.com/en-US/what-is-unreal-engine-4. [Accessed: April 2018].

[51] Sourish Ghost, "OpenCV C++ Stereo Camera Calibration," 2016. Available: https://github.com/sourishg/stereo-calibration. [Accessed: April 2018].

[52] A. Geiger, M. Roser, and R. Urtasun, "Efficient large-scale stereo matching," in *Asian conference on computer vision*, pp. 25–38, Springer, 2010.

[53] P. Selinger, "Potrace," 2017. Available: http://potrace.sourceforge.net/. [Accessed: April 2018].

Appendix A

Demonstrations of Full Data Abstraction

FIGURE A.1: Demonstration of Full Abstraction Process. The final parameters and methods are a blur kernel of 5x5, a low threshold of 25 (for this example), Sobol seed point generation, and flood fill averaging. It can be seen that most areas of the image are being effectively edge detected and flood filled with the correct colours, however, some areas are hard to interpret such as around the bushes in the bottom left, and some spaces have been left unfilled such as the panel below the top left corner of the building. These issues are minimal though, therefore leading to the conclusion that the process is effective at producing recognisable abstractions of the input images.

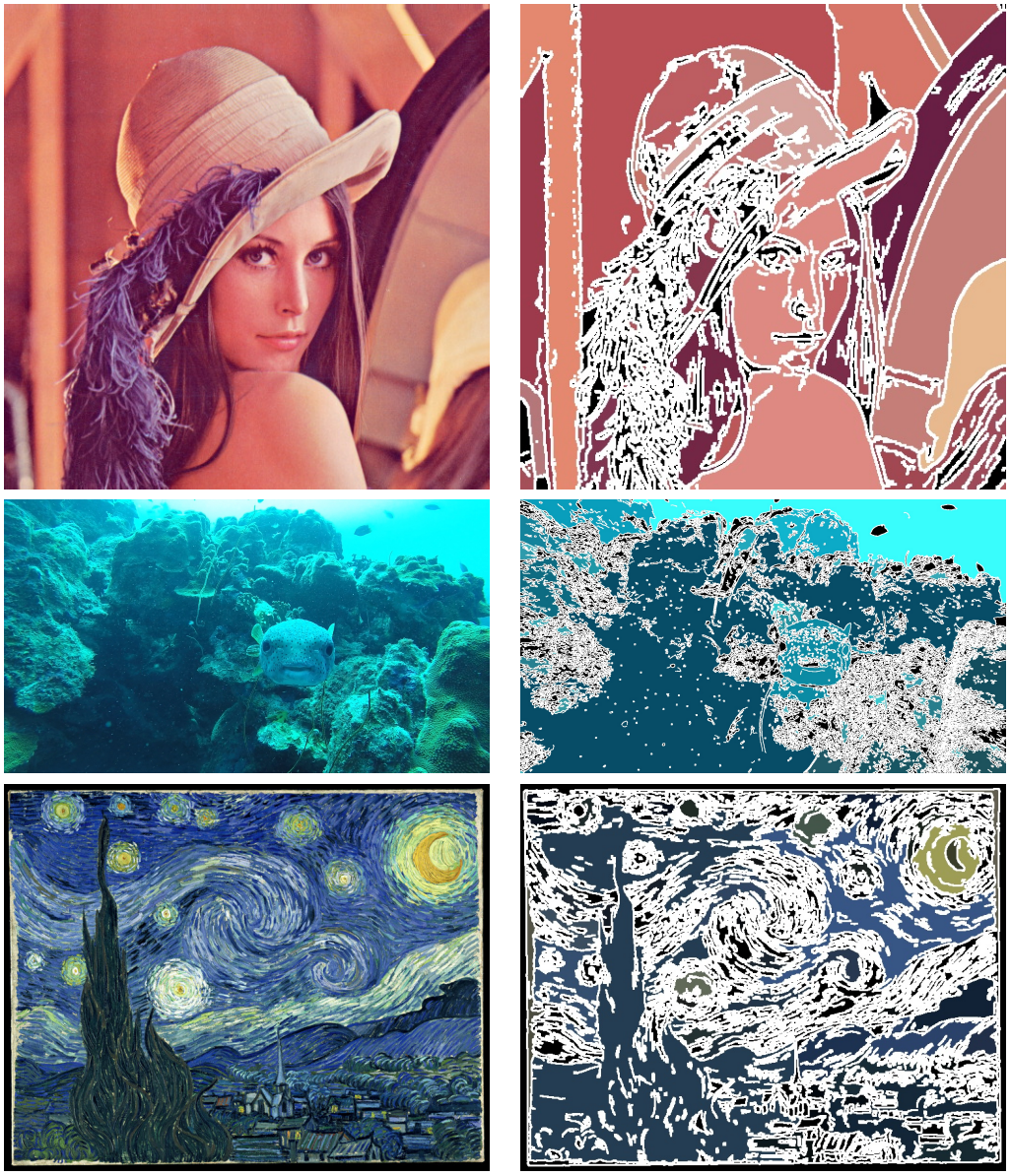


FIGURE A.2: Additional Examples of the Full Abstraction Process. The top image (provided by OpenCV [30]) is very simple, so the abstraction gives clean and easily recognisable results. The middle image (provided by fellow student Tom Darlison) was selected due to its limited colour palette and poor focus to test the limits of the process. While the image has mostly filled to a single colour, the pufferfish, rocks, coral, background fish, and open water are all identifiable. The bottom image (provided by OpenCV) was selected due to the extreme number of edges provided by its clear brush strokes. Once the Canny threshold is turned down considerably, the result is recognisable as Starry Night, though much of the image has been overwhelmed by the edge detected lines. It can be concluded from these tests that the data abstraction process is effective at producing recognisable images, however the difficulty in interpreting the abstracted images is heavily dependant on the focus and complexity of the image.

Appendix B

Contour Based Seed Point Location

The ideal place to aim to flood fill a space from would be the centre point of the space. Aiming for the centre provides robustness against errors, as the seed point is far from the edges and is unlikely to be incorrectly placed outside the boundaries of space. OpenCV provides the functionality to take an edge detected image (a matrix of colour values) and convert it into a set of contours. Contours are line objects stored in a hierarchical structure, similar to how vector images are stored, and have functions that can provide the centre point of each contour. Although the centre points of the contours do not map exactly to the centre points of the spaces in the image, they are close approximations (Figure B.1).

Unfortunately, due to a combination of the processing time required to convert lines into contours and the number of very small extraneous contours produced, this method has resource requirements too high to produce a frame rate above even 5 fps on a laptop, therefore making it completely unusable on a Raspberry Pi.

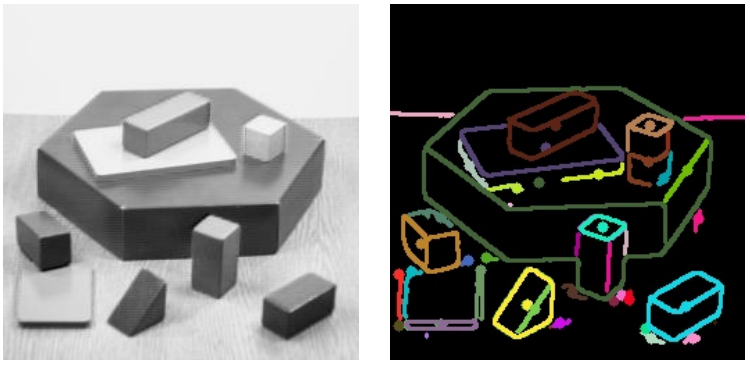


FIGURE B.1: Demonstration of Contour Centre Point Location. The original image (provided by OpenCV [30]) on the left has been edge detected, the edges converted into contours, and the centre points of these contours located. The results of this are displayed on the right, with each contour and its corresponding centre point in a different colour. It can be observed that the centre points provide adequate coverage of the spaces in the image to be used as seed points for flood filling.

Appendix C

Images of Algorithm Performance Test Scenes

FIGURE C.1: Images of Algorithm Performance Test Scenes. From top to bottom, these are images of the simple scene, medium scene, and complex scene.

Appendix D

Rover Hardware Breakdown

TABLE D.1: List of all the components that make up the rover. Items listed as UoS were supplied by the University of Southampton.

Component	Description	Quantity	Source
Raspberry Pi 3	Main computer	1	UoS
ATmega644P	Generates PWM for stepper drivers	1	UoS
Pololu DRV8825	Stepper motor drivers	2	HobbyTronics
RS 535-0372	Y-axis stepper motor	1	Previous Project [9]
SY35ST26-0284A	X-axis stepper motor	1	Previous Project [9]
PlayStation 3 Eye	Cameras	2	Amazon
MAX14870	DC motor drivers	2	HobbyTronics
Lynxmotion GHM-04	Drive motors	4	UoS
GEP-XT60-PDB	Power distribution board	1	Hobby King
Turnigy 2650mAh 4S 20C	Lipo Batteries	2	Hobby King
IC 108TTA050M	Decoupling capacitors	2	UoS

Appendix E

Vectorization

As edge detected images consist of white lines on a black background, it would be logical to consider vector graphics as a storage method for wireless transmission. Vector graphics is the storage of images as line drawing and space filling instructions, allowing scaling to any size. When testing this, Potrace [53] was used to convert the edge detected image bitmaps into encapsulated postscript documents (Figure E.1).

FIGURE E.1: Conversion to Vector Graphics. While the vectorised image (bottom right) is clearly of the edge detection (bottom left), it utilises heavy approximation.

When the documents were received by the server, they were converted back to OpenCV bitmaps using a custom interpreter built for the project. Both due to the approximation applied by Potrace and the custom interpreter not including all the necessary functions to accurately convert the vector files back to bitmaps, the final result loses much of its accuracy to the original image (Figure E.2). Due to the high edge accuracy requirements of producing a depth map from edge detected images, this method was discarded in favour of standard image compression.

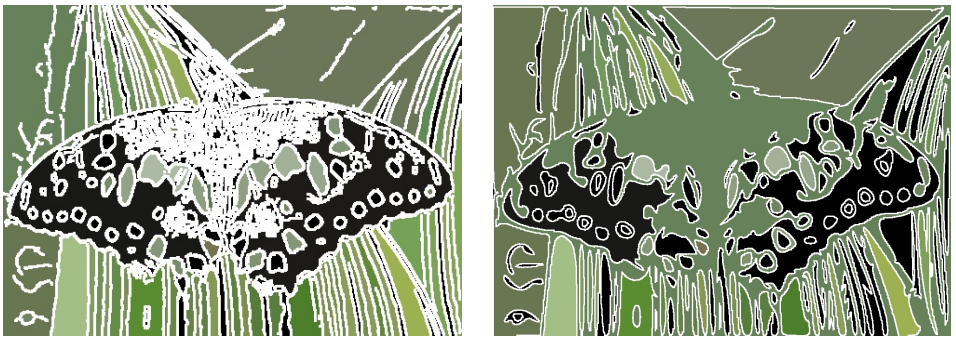


FIGURE E.2: Comparison of Vector and Bitmap Accuracy. Once the vector image has been converted back into a bitmap (right), it has lost much of its resemblance to the original image. A comparison to the image that results from transmitting compressed bitmaps (left) shows the sheer number of edges that have been either lost or combined together.

Appendix F

Project Brief

Using Data Abstraction and Inter-Frame Interpolation for Low Data Rate Communication Between a 3D Camera and VR Headset

Adam Melvin am20g15@soton.ac.uk
Klaus-Peter Zauner kpz@soton.ac.uk

There are many challenges, such as monitoring hostile environments, that call for the use of remotely controlled robots. Often in these scenarios, it would be useful to be able to view the environment with a sense of depth to better understand the scale, and dangers, of the robots surroundings. This can be provided through the use of a 3D camera on the robot and Virtual Reality (VR) goggles, however due to the minimum frame rate that can be displayed in a VR headset without causing motion sickness in the user being 60fps (optimally 90fps is preferable), a comfortable and useful experience would often require an unfeasibly high wireless data rate.

The aim of this project is to significantly reduce the data rate required to be transmitted between a robot and teleoperator through the use of data abstraction and inter-frame interpolation. A 3D camera rig mounted on a remotely controlled rover is to take around 10 pictures per second, then they are reduced down to the minimum amount of data required to identify the objects in the environment. These much smaller images are sent wirelessly to a server, where they are used to create a 3D map of the environment. These 3D maps are analysed and intermediate frames are estimated to increase the frame rate from 10fps to 60-90fps. This much higher frame

rate estimated map of the environment can then be displayed in the VR headset through the use of a video game engine. Although the estimated frames will not accurately represent the real world, the comfort they provide the user will allow them to focus on the transmitted information without feeling ill.

A simple rover and off-the-shelf VR equipment will be used as a foundation for the system. Different camera systems and interpolation algorithms will be tested on this foundation to discern the setup that produces the best ratio between data rate and usability.

Appendix G

Gantt Chart

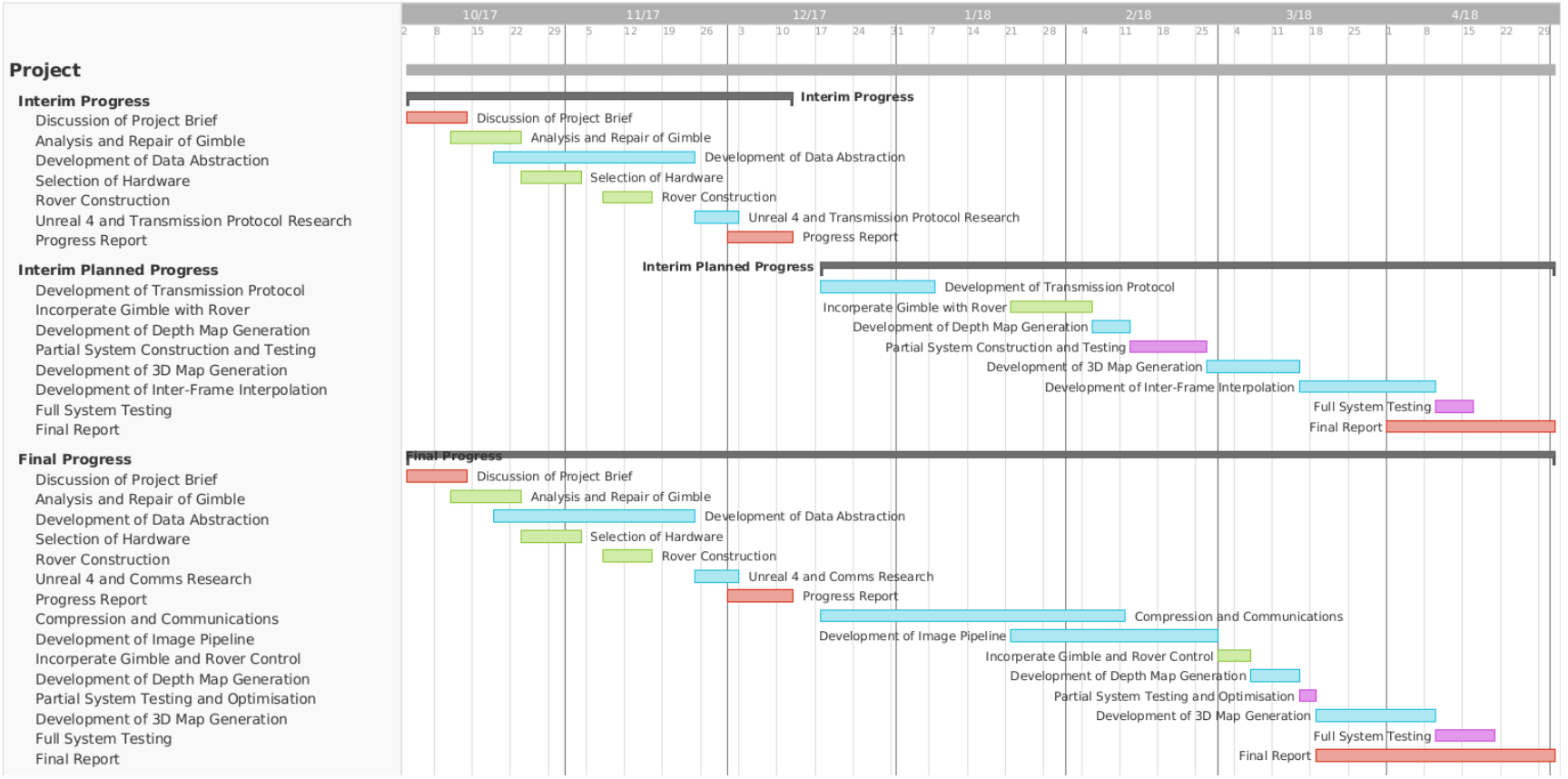


FIGURE G.1: Gantt Chart. Red segments are report writing, green are hardware construction, blue are software development, and purple are testing.

Appendix H

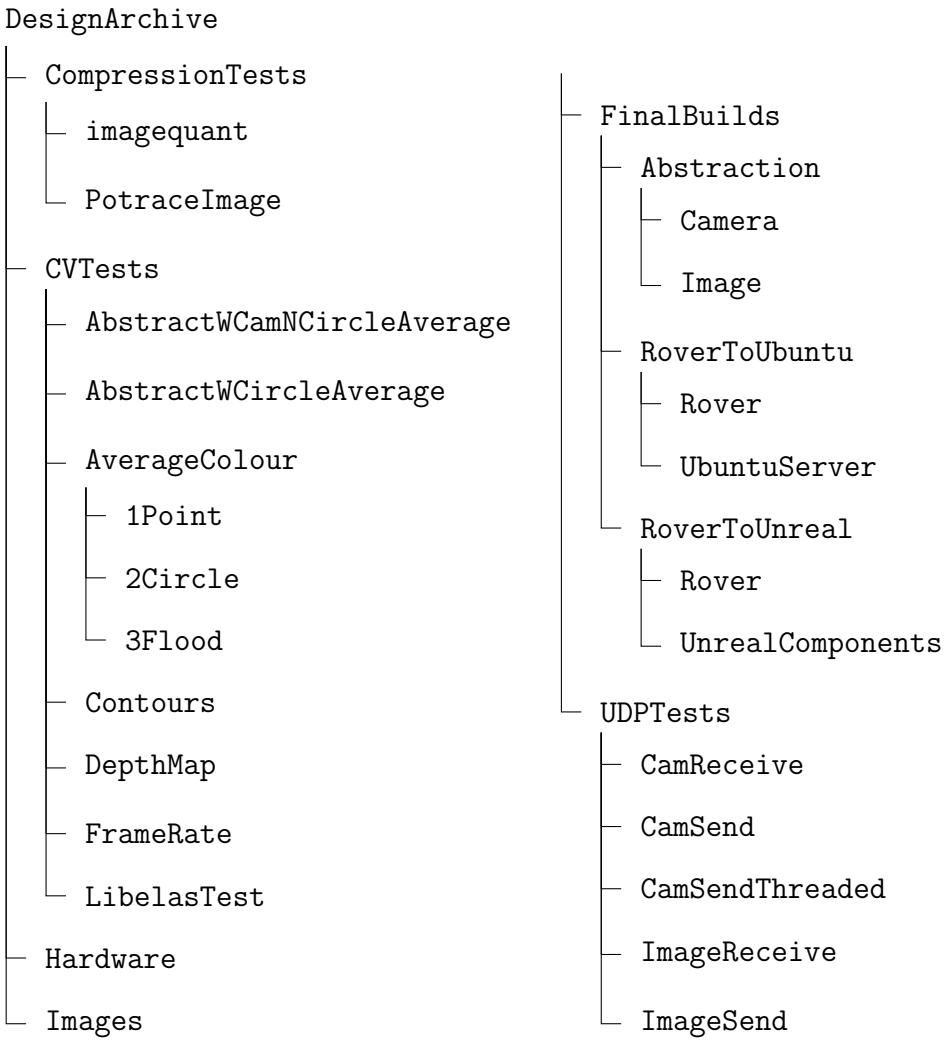
Risk Assessment

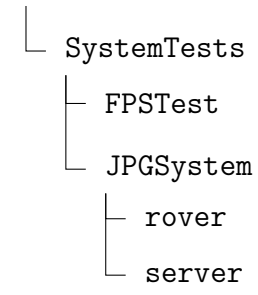
Risk Event	Likelihood (1-5)	Impact (1-5)	Risk Exposure (1-25)	Action	Did Event Occur?	Impact of Event
Unable to complete system in time	4	4	16	System is built from the input to the output, allowing for the submission of the working section instead	No	N\A
Response time of the system is unacceptable for use in VR	3	5	15	System is designed and being build with runtime performance as the highest priority	No	N\A
Inter-frame interpolation does not perform as intended	3	3	9	System can function without this component	No	Interpolation removed for other reasons, with no impact on system functionality.
Data abstraction process is not accurate enough to form a depth map from	2	4	8	Areas where refinement is possible are known so the process can be made more accurate if neccessary	No	N\A
EasyCap devices are never available or do not work with the Raspberry Pi	4	2	8	Preparations have been made to use my supervisor's budget to replace FPV cameras with webcams	Yes	FPV cameras successfully replaced with PS3 Eyes.
Data abstraction process is too resource intensive for the Raspberry Pi	2	3	6	Areas where quality reductions could be made in exchange for performance are known	Yes	Good performance achieved through quality reduction and threading.
LiPo battery combusts	1	4	4	LiPo batteries are supervised while charging, charged in a flame-retardant bag, and only connected in parallel when the same voltage	No	N\A
Hardware failure	1	2	2	Remaining budget can be used to replace failed component	Yes	Failed components were replaced.

FIGURE H.1: Risk Assessment. Left of the thicker line is the original risk assessment from the interim report, and right of the thicker line is the end of project evaluation of the risk assessment.

Appendix I

Design Archive Contents




SystemTests
FPSTest
JPGSystem
rover
server