Chapter 1

Introduction

There are many areas of science and engineering that require the observation of, and interaction with, environments not suitable for human beings. These environments are instead observed using teleoperated robots, potentially completely removing the need to put people in any danger. However, a challenge presented by telerobotics is giving the operator sufficient information about the robot's surroundings to give them a feeling of presence [1] within the space, allowing for effective manoeuvring and interaction.

Virtual Reality (VR) is a technology that has proven itself to be able to provide the user with unparalleled presence within a virtual space- comparable to presence within a real, physical space [2]. To be able to incorporate VR into teleoperation is therefore desirable. While this has been successfully attempted for the purpose of control within an already known space [3], there has been much less success in exploring an unknown space through a VR interface. This is due to the high frame rate and low latency required to prevent motion sickness while in a VR environment.

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) [4]; a minimum of 60 fps can also be acceptable, 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 very high bandwidth requirements. Also, a major factor in providing presence to the user in VR is their

ability to look around the space independently. This can be achieved by mounting the stereo camera rig on a gimble, however for the gimble to track the angle of the user's head accurately and with low latency would be very challenging; the user would be more likely to suffer sickness and dissociation from the space than if the gimble was not used at all.

The aim of this project is to design and implement a VR based teleoperation system that utilises data abstraction and inter-frame interpolation to minimise the outlined technical issues, providing increased comfort and therefore presence to the user than otherwise possible. Data abstraction must be used in the robot to reduce each image down to its most essential features, reducing its size and therefore the required data rate significantly. Data rate must be further reduced by taking images at a low frame rate, such as 10 fps, and then increasing it up to 90 fps on the computer using inter-frame interpolation methods. Although this process will be based on interpolation methods, the frames will be extrapolated from optical flow based predictions so as to not increase latency. Each image pair must be finally be combined into a single 3D map of the space that can be looked around freely through the VR headset. As previously suggested, the camera gimble will track the movement of the headset, but it does not have to be very accurate as it is only updating the 3D map and not affecting the headset directly.

The system will be realised using off-the-shelf VR equipment, a camera gimble produced by previous students [5], and a simple rover that I am also building. This progress report will be focused on the progress made in data abstraction, so will not be covering the other aspects of the project other than to put the abstraction in context.

Chapter 2

Background

2.1 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 as opposed to attempting a full drawing, and is comparable to data compression as the aim is also 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 [6]. 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 purpose. For example, if all that is required of a 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 2.1.

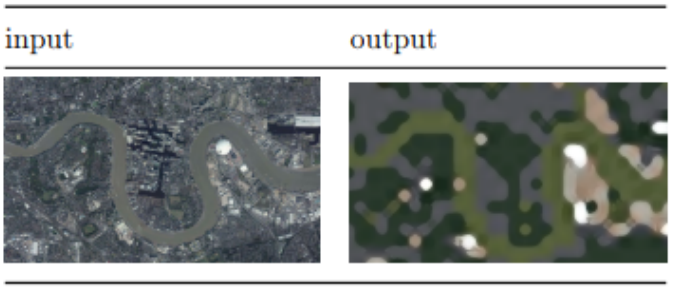

FIGURE 2.1: Data Abstraction Example. This is the abstraction of an aerial photograph of London, reproduced from [7]

2.2 Sobol Sequences

Sobol sequences are quasi-random sequences that were introduced to aid in approximating integrals over an S-dimensional unit cube [8]. The aim is to form a sequence x_i in S-dimensional unit cube I^S that approximates

$$\int_{I^S} f \quad by \quad \lim_{n\to\infty} \frac{1}{n} \sum_{i=1}^{n} f(x_i) \tag{2.1}$$

For this to hold, the points of x_i should be selected so they are evenly spread across I^S. This provides a much more even spread of points across the chosen space than can be produced from a pseudo-random number source (Figure 2.2). The code used in this project to produce these sequences was created by Leonhard Grünschloß [9] and is presented in Appendix A.

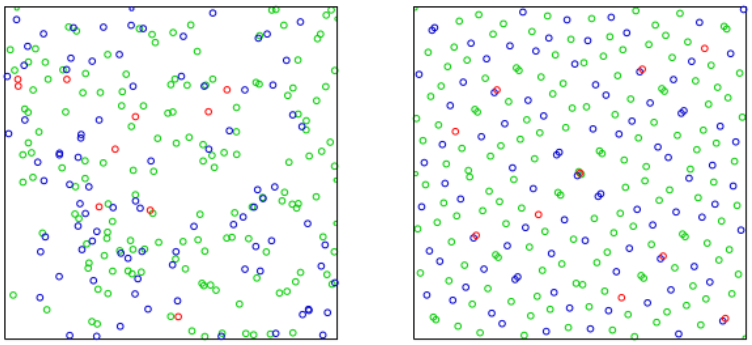

FIGURE 2.2: Comparison of pseudo-random and quasi-random sequences. 256 points from a pseudo-random generator (left) and 256 points from a Sobol sequence (right), reproduced from [10].

2.3 Computer Vision

Computer vision is the automatic analysis of images and extraction of the useful information they contain [11]. 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 many different techniques such as statistical pattern classification and geometric modelling [12]. 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 [13].

2.3.1 Edge Detection

When attempting to recognise the features of an image, knowing the locations of the edges of objects within the scene is often very useful. An edge is defined as a significant local change in intensity, usually due to a discontinuity in either the intensity or its first derivative [14]. 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, the most effective in almost all scenarios is Canny edge detection [15],

therefore this is the algorithm utilised in this project. Canny edge detection [16] (Figure 2.3), and can be divided into 4 stages:

1. Reduce noise using a Gaussian filter
2. Find the intensity gradient of the image by taking the first derivative in the horizontal and vertical directions.
3. Remove from the gradient map any pixel that is not a local maximum, reducing the edges down to their minimum thickness.
4. Remove any edge that either has an intensity gradient lower than a lower threshold value or not connected to pixels with a value larger than an upper threshold value (hysteresis thresholding).

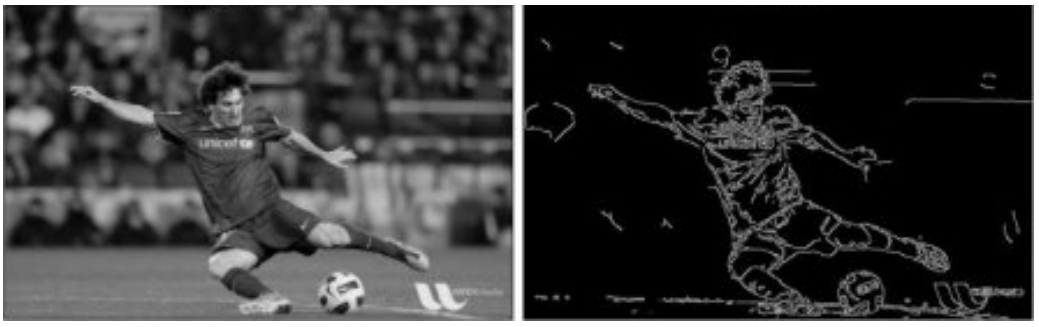

FIGURE 2.3: Canny Edge Detection Example. Simple edge detection program applied to a fairly detailed photo of Messi, to demonstrate its effectiveness even with more complex images. Figure taken from an OpenCV Canny tutorial [17].

2.3.2 Flood Fill

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 [18]. 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) [19]. A demonstration of flood fill has been presented in Figure 2.4.

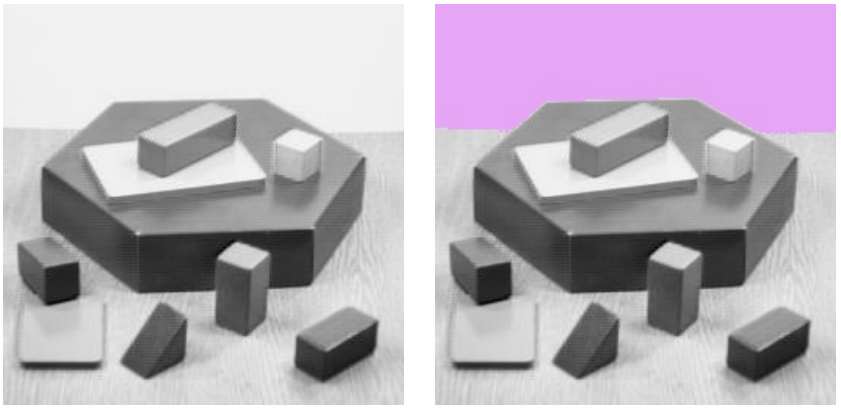

FIGURE 2.4: Example of Flood Fill. The original image (left) was provided by OpenCV. The right image is the result of flood filling from the top left corner, filling the lighter space at the back of the scene.

Chapter 3

Progress in Data Abstraction

In this implementation of data abstraction, the aim is to reduce images down to only the boundaries of the objects in the scene and then fill the spaces between theses boundaries with block colours based on the original image. Therefore the general process of the design is:

1. Use edge detection on the image, presenting the boundaries of the scene as white lines and the rest as black.
2. Divide up the original image into sections that can reasonably be averaged into a single colour, defined by the boundaries produced by the edge detection or otherwise.
3. Find the average colours (or reasonable alternatives) for these sections.
4. Flood fill the spaces in the edge detection output with the average colours of the original image.

While this section will be presented in the context of the entire process occurring on a single computer (as this was the testing platform), when incorporated into the final system points 1-3 will be done using the raspberry pi on the rover and point 4 will be done on the computer. This allows for much smaller data packets to be sent over a wireless connection (wifi), due to being able to send the colour information for entire sections of the image as a single colour value.

3.1 Edge Detection

Canny edge detection was implemented using the Canny function provided by OpenCV. The sequence of processes implemented to support the algorithm in producing high quality edges are shown in Figure 3.1.

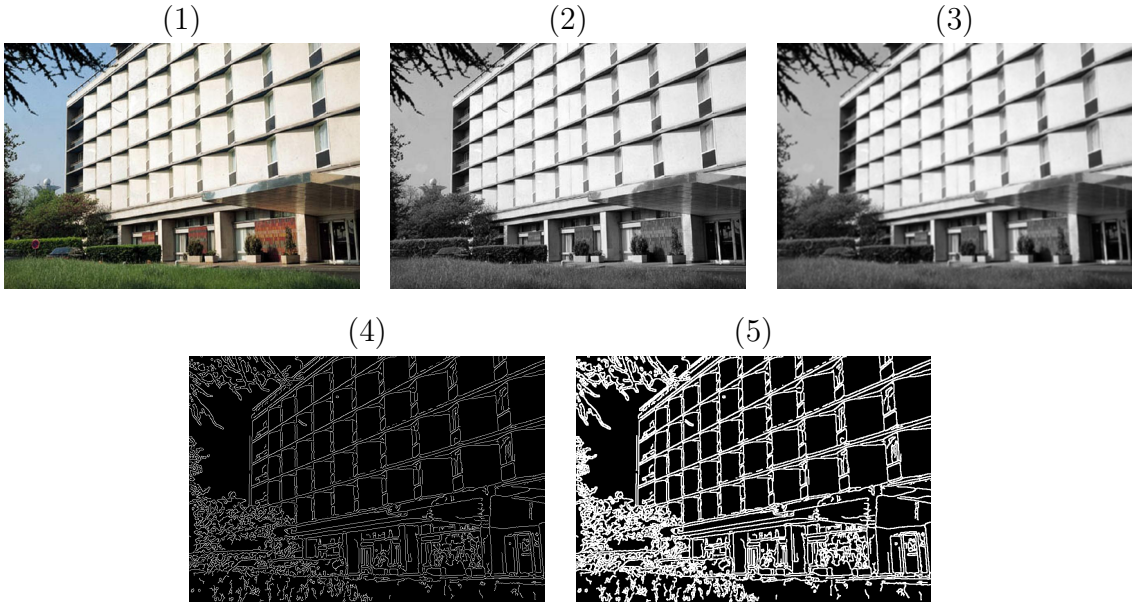

FIGURE 3.1: (1) the original image, provided by OpenCV, (2) post grayscaling, (3) post blurring, (4) post edge detection, and (5) post dilation.

The image is first made grayscale, as Canny detects large changes in light intensity and not colour. The image is then blurred to remove any unnecessary edges and noise that Canny may pick up. The edge detection is used, producing white lines representing the edges on a black background (the details of parameter experimentation with Canny and the previous blurring are presented in Appendix B). 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 make the image more cartoon-like and more generally aesthetically pleasing, reduce the number of lines produced by areas that are dense with detail such as hair and foliage (an example of this detail density can be seen in the bush in Figure B.1), and to bridge the gaps between lines that are close together, increasing the likelihood of defined shapes being created that can be easily flood filled later.

3.2 Flood filling

Flood filling was chosen as the method for applying colour to the Canny output image, as it is the most efficient way to fill spaces of unknown size and shape that are defined by high contrast boundaries; using it makes dividing up the image into sections unnecessary. To use the OpenCV flood fill function you must provide a seed point to start flooding from, a colour to fill with, and parameters for the filling itself (unchanged from the defaults provided by the OpenCV documentation [20]).

3.2.1 Seed Point

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 anywhere. Three different methods were attempted to solve the problem. The first was an attempt to use OpenCV's contour functionality to provide seed points, however this was unusable due to high resource requirements. The method is detailed in Appendix C.

Although it would by ideal to aim to flood fill from the centre point of each space, it is only necessary if a few specific pixels are being selected to fill from, as each time a line pixel is selected that potential seed point can only be discarded. It is possible to instead iterate through the whole image and flood fill from every pixel found that is not part of a line or an already filled space.

This method is more effective at filling every space than the previous, and is also less resource intensive. However, if presented with a complicated environment with many spaces to flood fill, it must fill every single one, therefore leading to unacceptable drops in frame rate.

The most simple solution to the performance issues caused by allowing the number of times flood fill is used per image to vary would be to set a maximum. However, if this is done 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 solution is to select a set number of points quasi-randomly across the image using Sobol sequencing. Although a certain number of points will land on lines and therefore not

be used, if there are enough points then all the important spaces are filled without serious impact on performance. Also, with this method performance is not affected by the complexity of the image, however more complicated images will be processed with many of the more dense areas unfilled (Figure 3.2). For these reasons, the seed points are selected using this method in the current build.

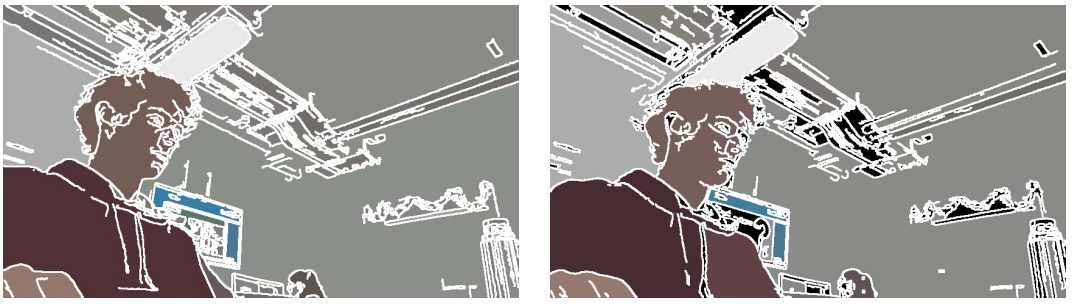

FIGURE 3.2: Comparison of brute force and Sobol seed point generation. It can be clearly seen that the brute force method (left) accurately fills every space in the image, whereas using Sobol results in many unfilled spaces. However, Sobol is higher performance with a frame rate of 17.14fps, compared to 8.57fps using brute force.

3.2.2 Fill Colour

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

While filling the abstracted image with the average colours present within the input is preferable, it is not essential to the project; provided that the objects in the scene are still recognisable, they don't have to be exactly the right colour. 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 of the seed point. This is a very fast method, however produces incredibly 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 sample from a different point each time 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 to be filled. This is due to the quasi-random points often being closer to the edge of the space that they are being flooded from then the radius of the circle. Therefore, the size of the circle must be carefully selected to balance the benefits of increasing size (more consistency when the seed point is further from the edge) 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 [21]. 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 the 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 Canny output itself is filled. The consistency between images for this is the maximum possible based on colour alone (Figure 3.3), 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. This method has a noticeable impact on performance, though within acceptable bounds, leading this to be the chosen method for the current build. Examples of the results produced by the final build can be seen in Appendix D.

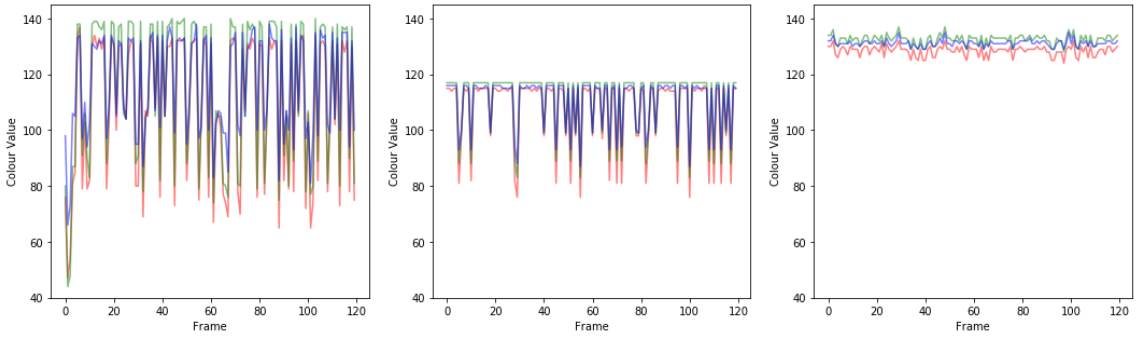

FIGURE 3.3: Comparison of Colour Averaging Methods. These are RGB values over time produced by flood filling an example area using the seed point colour (left), circle average (middle), and flood fill average (right). The increase in consistency from left to right is very apparent.

Chapter 4

Plan for Remaining Work

When building a system of this scale, great care must be taken to thoroughly test each stage both individually and in the context of the full system. For this reason, the general approach when organising my time was to work from the start point of the process (the hardware of the rover and the data abstraction it uses), making my way through each stage and combining it with the last to create larger subsections of the system. This also allows for better risk management; if a stage does not function as intended to the extent that the project cannot be completed, the work already done would still function as its own smaller system. A gantt chart providing the details of this approach, both going forward and its results, can be found in Appendix E.

A break down of how the budget has been used so far has been presented in Appendix F. The total budget spent is £124.97, leaving £25.03 spare to be used in the event of hardware failure. Also in Appendix F is a full risk assessment for the project going forward, necessary due to the high risk presented by projects of this scale.