Situational Awareness of a Ground Robot from an Unmanned Aerial Vehicle

Daniel Hager

Thesis submitted to the faculty of the Virginia Polytechnic Institute and State University in partial fulfillment of the requirements for the degree of

Master of Science
In
Computer Engineering

A. Lynn Abbott
Kevin B. Kochersberger
Peter M. Athanas

May 4, 2009
Blacksburg, VA

Keywords: Object Tracking, UAV, Obstacle Detection, Supervisory Control, Image Processing

Situational Awareness of a Ground Robot from an Unmanned Aerial Vehicle

Daniel Hager

ABSTRACT

In the operation of unmanned vehicles, safety is a primary concern. This thesis focuses on the use of computer vision in the development of a situational awareness system that allows for safe deployment and operation of a ground robot from an unmanned aerial vehicle (UAV). A method for detecting utility cables in 3D range images is presented. This technique finds areas of an image that represent edges in 3D space, and uses the Hough transform to find those edges that take the shape of lines, indicating potential utility cables. A mission plan for stereo image capture is laid out as well for overcoming some weaknesses of the stereo vision system; this helps ensure that all utility cables in a scene are detected. In addition, the system partitions the point cloud into best-fit planes and uses these planes to locate areas of the scene that are traversable by a ground robot. Each plane's slope is tested against an acceptable value for negotiation by the robot, and the drop-off between the plane and its neighbors is examined as well. With the results of this analysis, the system locates the largest traversable region of the terrain using concepts from graph theory. The system displays this region to the human operator with the drop-offs between planes clearly indicated. The position of the robot is also simulated in this system, and real-time feedback regarding dangerous moves is issued to the operator.

After a ground robot is deployed to the chosen site, the system must be capable of tracking it in real time as well. To this end, a software routine that uses ARToolkit's marker tracking capabilities is developed. This application computes the distance to the robot, as well as the horizontal distance from camera to the robot; this allows the flight controller to issue the proper commands to keep the robot centered underneath the UAV.

Acknowledgements

This thesis would not have been possible without the help of several individuals. First, I would like to thank all of my committee members for what they have contributed to my work. Dr. Athanas was the first to get me interested in research, and extended me the opportunity to do undergraduate research in his lab. His classes were also what led me to realize that I wanted to do research and even pursue a career in the design of embedded systems. Dr. Abbott's knowledge in the field of computer vision has been crucial for my work on this thesis. In addition, his insights on technical writing have helped me hone this thesis to a point beyond what I could do on my own. I must especially thank Dr. Kochersberger for bringing me into the Unmanned Systems Lab and giving me the opportunity to work on this project. His constant support of my work and continual suggestions on ways to improve on it have been invaluable. In addition, this project would not have been possible without the support of the Defense Threat Reduction Agency and the partnership of Savannah River National Laboratory.

I would also like to thank my friends and colleagues, the other research assistants at the Unmanned Systems Lab. Jimmy May, Prather Lanier, Eric Brewer, Mike Rose, and Nathan Short have all helped me in one way or another during my year at the lab. Between giving suggestions on my work and laughing at some of the troubles we've encountered dealing with the helicopter, they've helped me form this thesis and stay sane while doing it. In addition, I would like to thank Mohamed Saleh for his assistance in obtaining stereo images for the utility cable detection experiment.

My family has been instrumental in helping me get to where I am. My parents Mike and Ann have given me constant love and support for my entire life. They taught me the value of education at an early age, and gave me the passion for learning that I have now.

Finally, I would like to thank my wife Heidi for all the love and encouragement she has given to me during my work on this project. I have spent a large amount of our first year of marriage working on this thesis, and so I appreciate her patience through this entire process.

Contents

List of Figures

List of Tables

Chapter 1
Introduction

Computer vision is a topic of considerable interest to those in the unmanned systems field. Humans are quite familiar with gathering and interpreting information about their surroundings through the sense of sight. In a similar way, computers are capable of interpreting visual information to some degree through the analysis of digital imagery. In autonomous navigation systems, this can be especially useful for detecting obstacles and identifying various terrain features. In recent years, considerable research has gone into the development of computer vision for these types of applications. The developments discussed in this thesis are built on much of this recent research.

1.1 Background

Through the use of digital camera interfaces, computer systems have access to a large amount of visual information. By taking advantage of proper algorithms, these computers can process the images to contribute to a multitude of applications. They can be used for tracking specific objects, interpreting information about an environment, or even estimating the position of objects relative to the camera. With regard to the field of unmanned vehicles, this information becomes particularly critical, as it allows a robotic system to gather and interpret data about its surroundings without the need for human interaction. Alternatively, this data can be used to enhance remote control of unmanned vehicles. By using these computer systems to process imagery, the visual information can be filtered for the important pieces. Thus human in-the-loop operators are given much more useful, focused feedback. Providing a human user with this type of narrowed-down, informed data is intended to make his task much easier and more efficient.

There are several tools available for the development of computer vision applications. OpenCV (short for Open Source Computer Vision)[1] is a popular cross-

platform library that primarily centers on real-time image processing. OpenGL (Open Source Graphics Library) [2] is used extensively as well, though it serves a much wider range of applications, and is not specifically geared towards image analysis. Many computer vision-related toolkits build APIs on top of one of these two libraries.

An important related field is augmented reality, which combines real-time vision-based tracking with the overlay of virtual objects. A key player in this research area is ARToolkit[3]. This toolkit can be used to track the location of specific markers in an image sequence by searching through the data for black squares, which represent the frame for each marker. The software then resolves the orientation of the object relative to the camera. Once the position of the camera is known, a 3D model is then drawn from that same position, and overlaid on the real image. These steps are illustrated in Figure 1.1. Note that the term HMD in the diagram refers to a head-mounted display commonly used in augmented reality applications to display scenes to a user. This process creates the effect of introducing a virtual object on top of the real-world marker.

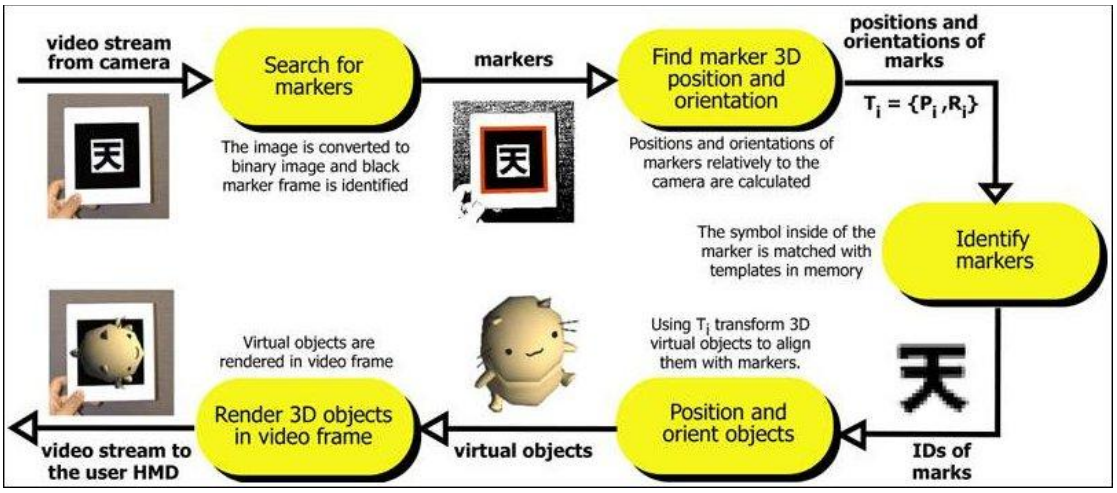


Figure 1.1 Diagram of ARToolkit steps **[3]**

An important component of any computer vision system is the camera that serves to gather data for processing. Many cameras are available, and the selection of the camera depends heavily on the specific constraints of the intended task. Typically, a single camera provides one image at a time. This type of sensing is used for the ground robot

tracking algorithms discussed in Chapter 4; the method for choosing a specific camera to use for this task is detailed there as well.

For analysis of the terrain where the ground robot is to deployed, stereo vision has been chosen. A stereo vision system uses two or more cameras to gather images simultaneously from slightly different viewpoints. There are complete stereo camera systems available commercially, such as the Bumblebee (shown in Figure 1.2), which is produced by Point Grey Research. By using appropriate stereo-matching software, these systems produce sets of 3D data points, known as a point cloud. An example of a point cloud gathered from the scene shown in Figure 1.3 is given in Figure 1.4. Taken in conjunction with other data about the environment, one can determine a considerable amount about the terrain of an area using these point clouds.

Figure 1.2 Bumblebee stereo vision camera system

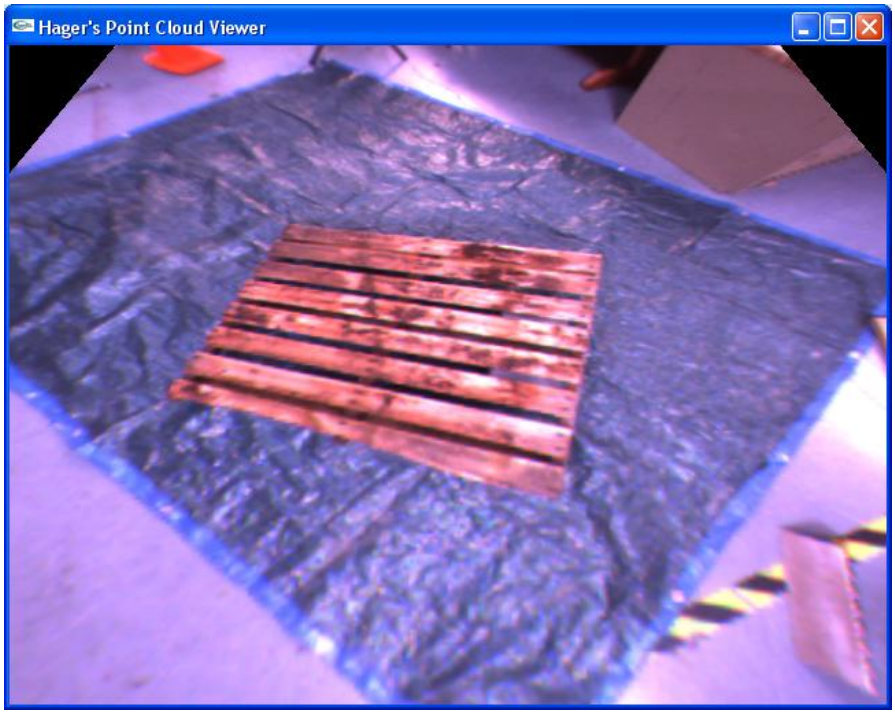


Figure 1.3 Traditional 2D image of example scene with wooden palate lying on a tarp

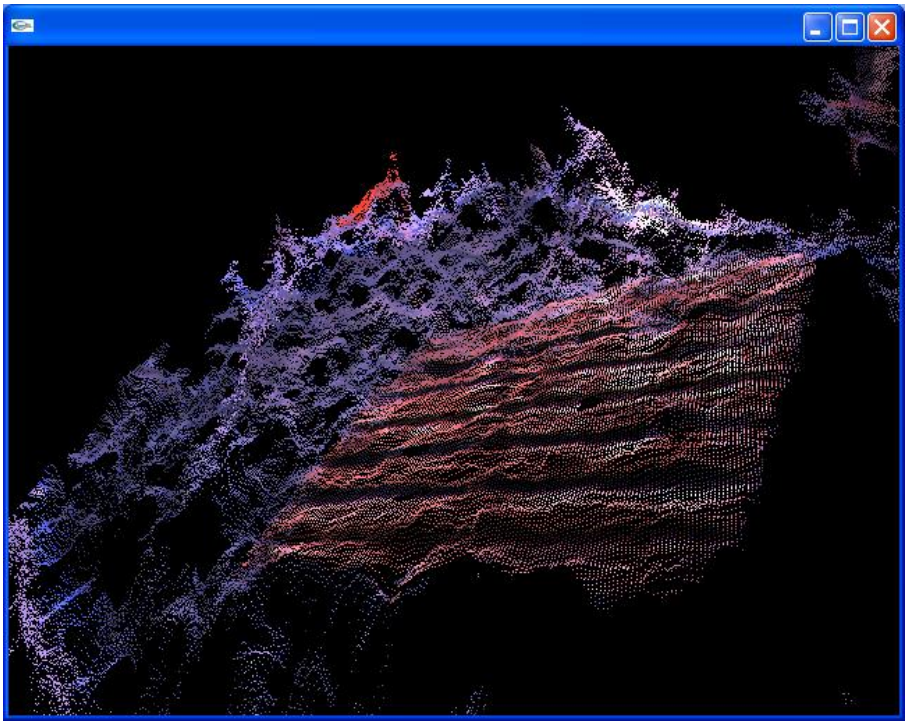


Figure 1.4 A point cloud from images taken of a wooden palate lying on a tarp

A brief description of the method used by the Point Grey software to generate these point clouds is given here. The first step of the process is the rectification of the two images[4]. This procedure slightly warps ("stretches") the images so that the corresponding points, such as *x* and *x'* shown below, lie on the same horizontal line in each image. A diagram of this geometry is shown in Figure 1.5, where *P* is an arbitrary point in 3D space, *C* and *C'* are the focal points for each camera in the stereo system, and *x* and *x'* are points in each image that lie on the same *epipolar* line. When the images are rectified, the corresponding epipolar lines form a set of parallel lines within the image planes [5]. Figure 1.6 shows the left and right rectified images of an example stereo pair.

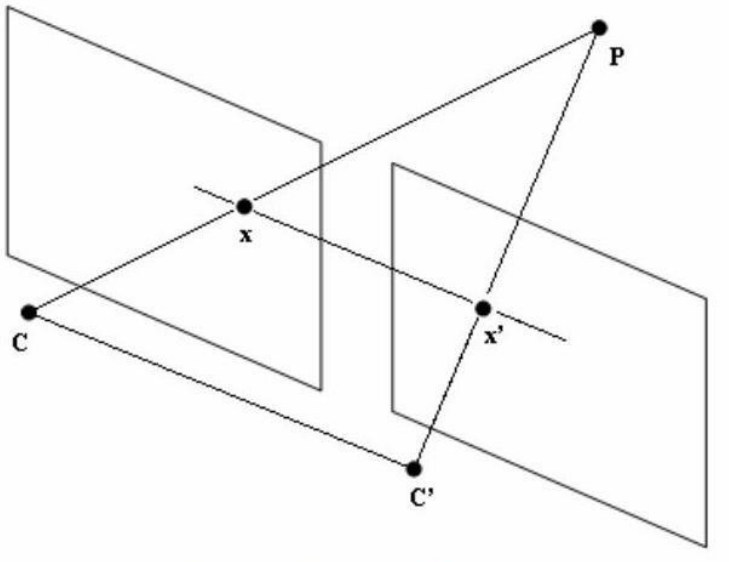


Figure 1.5 The two stereo images rectified so that epipolar lines are horizontal **[5]**

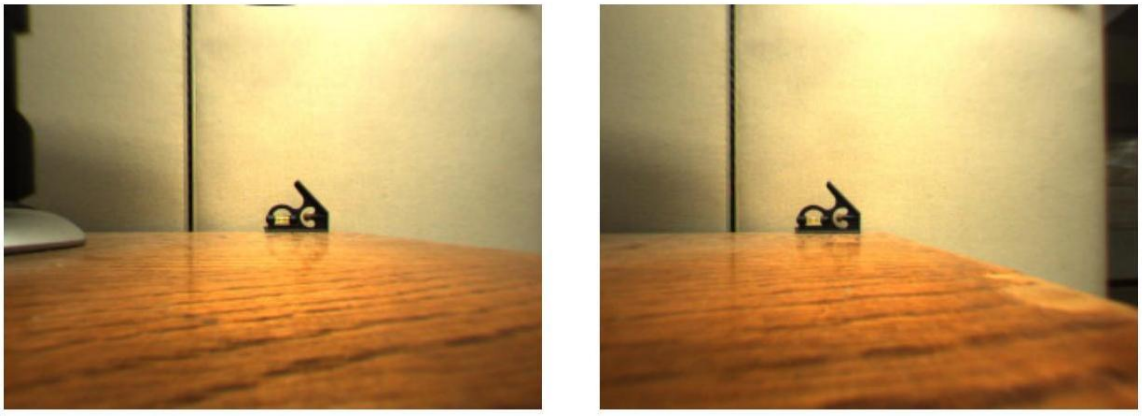


Figure 1.6 The left and right rectified images from a stereo camera pair **[6]**

Once the images have been rectified, the system attempts to identify corresponding points in the two images. The displacement between a matching pair *x* and *x'*, or disparity, is used to find the depth of points in the real world. This correlation is established in the Point Grey software by using the sum of absolute differences (SAD). This algorithm functions by creating a mask about each individual pixel and then comparing this to corresponding masks on pixels lying along the same epipolar line in the other image. The equation for SAD from [7] is shown here:

$$\min_{d=d_{min}}^{d_{max}} \sum_{i=-\frac{m}{2}}^{\frac{m}{2}} \sum_{j=-\frac{m}{2}}^{\frac{m}{2}} \left| I_{right}[x+i][y+j] - I_{left}[x+i+d][y+j] \right| \qquad (1)$$

where d_{max} and d_{min} are the maximum and minimum disparities, m is the mask size, and I_{left} and I_{right} are the left and right image arrays. The pixel in the second image with the lowest SAD output value is chosen as the best match for the original pixel of interest. The disparity between these pixels is inversely proportional to the depth of the real-world object.

Using the data calculated above in conjunction with the known camera properties, a 3D point cloud is generated. This point cloud gives the basic 3D position layout of the terrain. By analyzing this information, important aspects of the terrain, such as the slope and presence of obstacles, can be determined. The use of stereo data for assisting in the deployment of a ground robot from a UAV is discussed in Chapter 3.

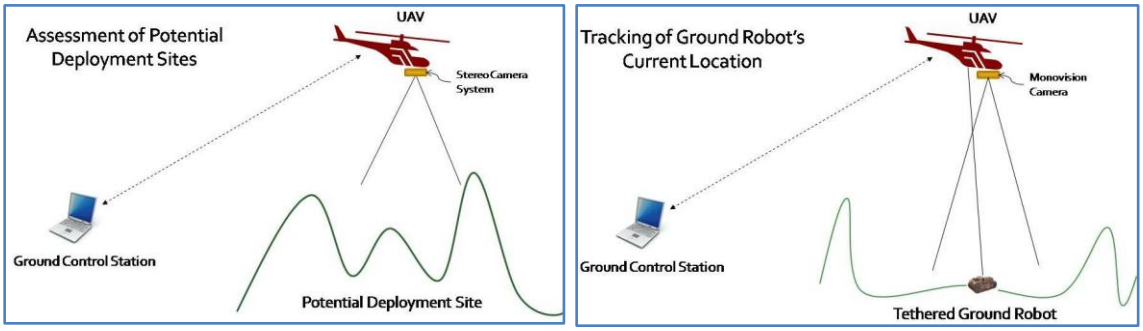
1.2 Description of Problem

The motivation for the work discussed in this thesis lies in the development of a system for deploying a ground robot from an unmanned aerial vehicle. The author of this thesis worked as a graduate student at the Unmanned Systems Laboratory at Virginia Tech. The goal of this project is a disaster response system that can take aerial images of a hazardous area and deploy a robot that can gather dirt samples. This response system utilizes an unmanned aerial vehicle (UAV); a typical platform for this is the RMAX helicopter (shown in Figure 1.7). In addition, a ground sampling robot will be deployed by the UAV.

Figure 1.7 RMAX helicopter to be used on missions

This UAV will be deployed to travel to a remote location that may be multiple miles from the human operator. The operator will be working at a ground control station communicating with the equipment on-board the UAV. To effectively supervise this helicopter and the ground robot, the operator must be provided with sufficient visual and contextual information. Relevant information for the initial deployment-site decision

includes the slope of the ground, the presence of any obstacles, and the size of the available roving area. Once a suitable deployment site has been chosen, the ground sample robot will be lowered by a tether to this site. Once the robot has been lowered to its target area, the operator should be given consistent overhead visual feedback of the robot for control purposes. To maintain this constant watch, the UAV must be kept in a constant hover over the robot. By tracking the robot in this manner, the aerial vehicle provides a birds-eye view of the robot and its surroundings. Finally, the information gathered from the helicopter in both phases of this mission must be delivered to the ground station in a useful manner. This should include an efficient user interface that provides ample information to the human ground controller of the suitability of the terrain as well as the location of the ground robot. Figure 1.8 gives a high-level view of the two primary phases of the mission, and Figure 1.9 provides a basic flowchart for this mission.



(a) (b)

Figure 1.8 High-level illustration of mission (a) phase 1 and (b) phase 2

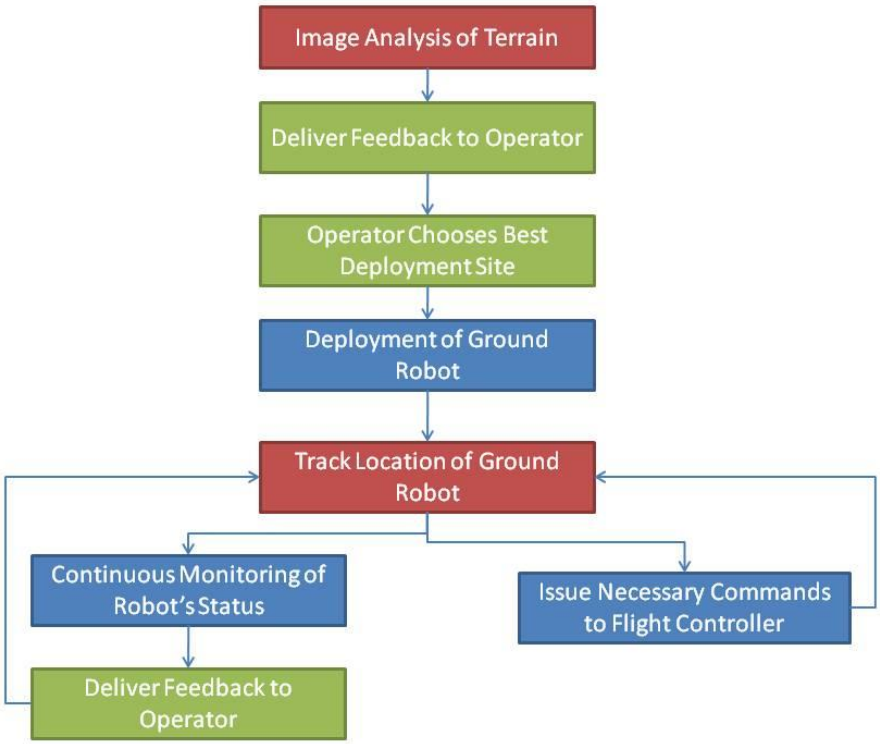


Figure 1.9 Flowchart for typical response system mission

1.3 Contribution

This thesis details the development of a situational awareness system that uses visual information gathered from an unmanned aerial vehicle to deploy and track a tethered ground robot. In the development of this system, several original contributions have been made. In the determination of suitable deployment areas, a new method for detecting utility cables using stereo imagery is presented. In addition, this method is tested across multiple possible scenarios, and the results are used to discuss mission parameters for the UAV-based response system. A new technique for determining regions in an image that are traversable by the ground robot, which takes local area drop-offs into account, is presented as well. By using concepts from graph theory, the size and adjacency of these regions is defined, and the area of an image best suited to traversal by the ground robot is located. Although the ARToolkit software used for tracking the

ground robot is not new, some original contributions with regard to the use and characterization of it are presented in this thesis. The output from the software's built-in marker detection functions are used to determine the real-world distance of a ground robot from a camera, as well as the movements necessary to center the object under the helicopter if the images are taken from a nadir perspective from the aircraft. In addition, the accuracy of these derivations is assessed, and the camera required to fulfill certain mission parameters is characterized.

1.4 Organization of Thesis

Chapter II of this thesis provides a critical review of literature related to the research discussed here. The system for determining a suitable deployment site for the robot is discussed in Chapter III. Chapter IV details the development of a means for tracking the location of the robot and keeping it centered beneath the helicopter using computer vision. In Chapter V, the user interface is discussed. This includes the transmission of the relevant information from the helicopter to a remote ground station, as well as the provided means of interaction with the human operator.

Chapter 2
Literature Review

There has been considerable research done in the area of computer vision, including some that deals with such vision systems on-board unmanned vehicles. This chapter looks at some notable papers that address the topics discussed in this thesis.

2.1 Utility Cable Detection

One key component of the situational awareness package discussed in this thesis is the need to detect narrow obstacles such as utility cables in the environment. Given the ubiquitous nature of utility cables, it is likely that they will be encountered in missions involving the autonomous helicopter/ground robot system. While the altitude of the helicopter may keep it safe from these dangerous obstacles, there exists a strong possibility of the tether attaching the helicopter to the ground robot becoming entangled in utility cables if they are not properly located. To this end, the development of a vision system that can reliably detect utility cables becomes critical.

In [8], the authors discuss the development of a vision system capable of locating utility cables in 2D images. They perform these operations for the purpose of establishing "vision-based lateral control." To locate the utility cables, they use the Hough transform[9] to search images from two different cameras for straight lines, and then calculate the position of the camera from these line positions. Figure 2.1 shows the results of the Hough line detection in these 2D images. The findings of this paper are relevant to this thesis in that they illustrate an efficient means for tracking utility cables in images. Similarly, the authors of [10] and [11] use Hough line detection to locate power lines in 2D images taken from a UAV. In [10], these results are further filtered based on prior knowledge of some properties of power lines, such as the tendency for several parallel

lines to be grouped together. By filtering the results, they limit the occurrences of false positives. Section 3.2 describes the Hough transform in more detail.

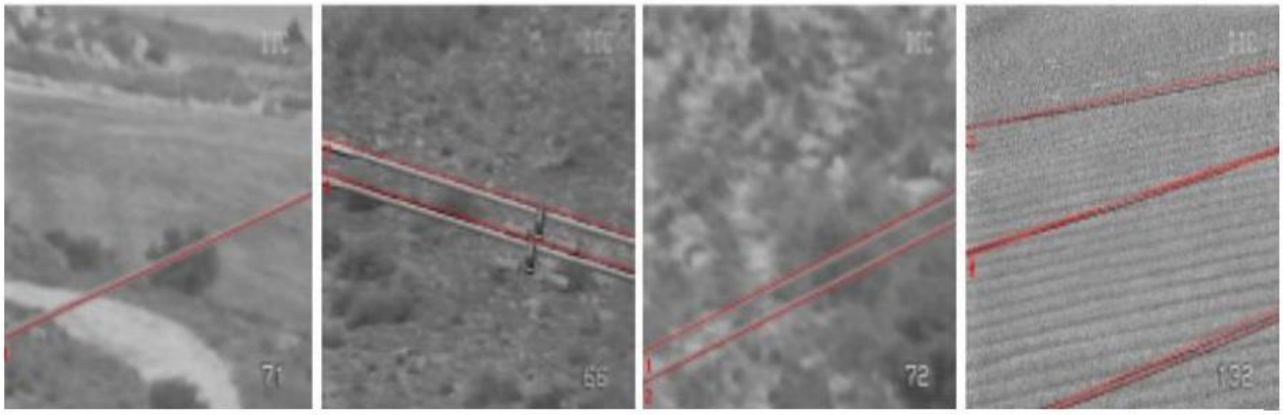
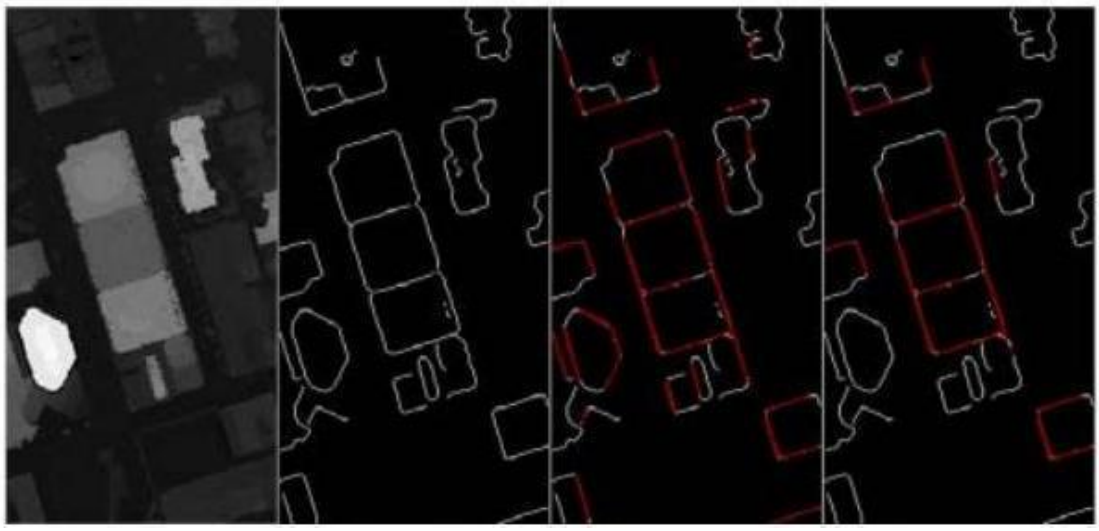


Figure 2.1 Results of 2D power cable detection **[8]**

While the Hough technique often proves effective for high resolution 2D images, it was important to consider the use of Hough transforms on 3D data. This topic is explored in [12], where the authors propose a system for reconstructing three-dimensional buildings using range images, which are 2D arrays of distance values. To accomplish this, they use a 3D Hough transform of the (*x*,*y*,*z*) data. Rather than searching for straight lines as is typically done with 2D images, the authors use the parameterized 3D data to find planes in the image. Using this data, they are able to reconstruct planar objects in the image, most notably buildings. Similarly, the 3D point data can be used to find lines in 3D space. That is the focus of [13], which concurrently uses this information to locate vertices at which these lines intersect. These calculations are performed for the purpose of locating boxes using 3D range data.

The authors of [14] are also interested in extracting feature points from 3D point cloud data. In this case, however, they do not examine the (*x*,*y*,*z*) data from the perspective of three-dimensional point coordinates. Rather, they use the *x* and *y* points as position coordinates, and treat the *z* value of each point as its "grayscale" value. They then perform edge detection on this data as one would to a 2D grayscale image. The Hough transform can then be applied to these images to extract geometric features, such as lines and rectangles. The results of this procedure are shown in Figure 2.2. As will be

shown in a later section, the power-line detection method used in this thesis was inspired by the technique from this paper. A similar technology is also employed in [15], which uses Lidar data gathered from an oblique perspective to resolve power lines. The Hough technique was applied to the Lidar data to locate curved lines.



(a) (b) (c) (d)

Figure 2.2 Range image from Lidar point cloud (a) along with the results of an edge detection (b) and the Hough transform output showing line segments (c) and rectangles (d) **[14]**

2.2 Vision-based Tracking

A common application of computer vision systems is the tracking of objects within images. This can involve either tracking the motion of natural features between frames, or locating pre-defined markers within images. When implemented efficiently, these object tracking algorithms can be run in real time to follow the movement of the features of interest. Because there are many practical uses for this capability, a large amount of research has been performed in this area, some of which is detailed here.

Natural Feature Tracking

One approach to object tracking involves the tracking of feature points in a video sequence from one frame to the next. In [16], the authors survey the research that has been done using this method. One such technique, known as template matching, identifies an object template in one frame through edge maps and pixel intensity, and then searches for this template in the next frame. Each successive frame is searched for the object template identified in the previous frame. However, this brute force search technique carries a high computation cost. A similar approach discussed by the authors is point tracking, which finds corresponding points in successive frames containing an object of interest. Of special note is the common motion constraint which detects point motion within a single object, even with the presence of occlusion or misdetection errors. The result of point correspondence using this constraint on a spinning plate is shown in Figure 2.3.

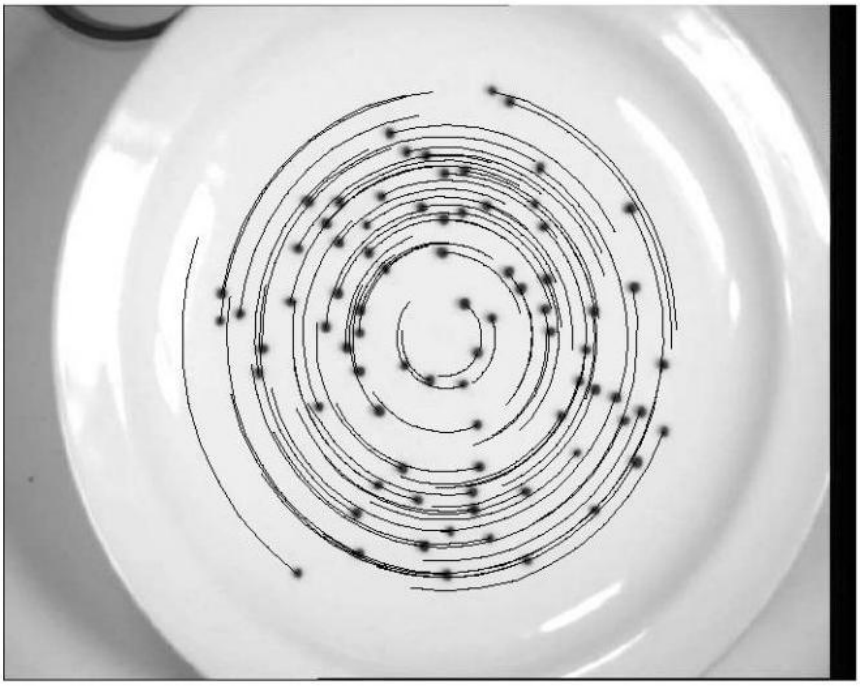


Figure 2.3 The results of the common motion constraint for point correspondence on a spinning plate **[16]**

In [17], the authors take a different approach to tracking features in an image. Rather than corresponding the location of points between frames, their approach

examines each frame for certain geometric features that correspond to likely objects of interest in the frame. Through derivation of the Jacobian matrices for the frames, along with using robust M-estimation statistical techniques, their algorithm manages to find features such as lines, ellipses, and cylinders in the visual data. A virtual control is implemented to servo the visual information and obtain pose estimation. As Figure 2.4 shows, this method works well even with interference from foreground objects.

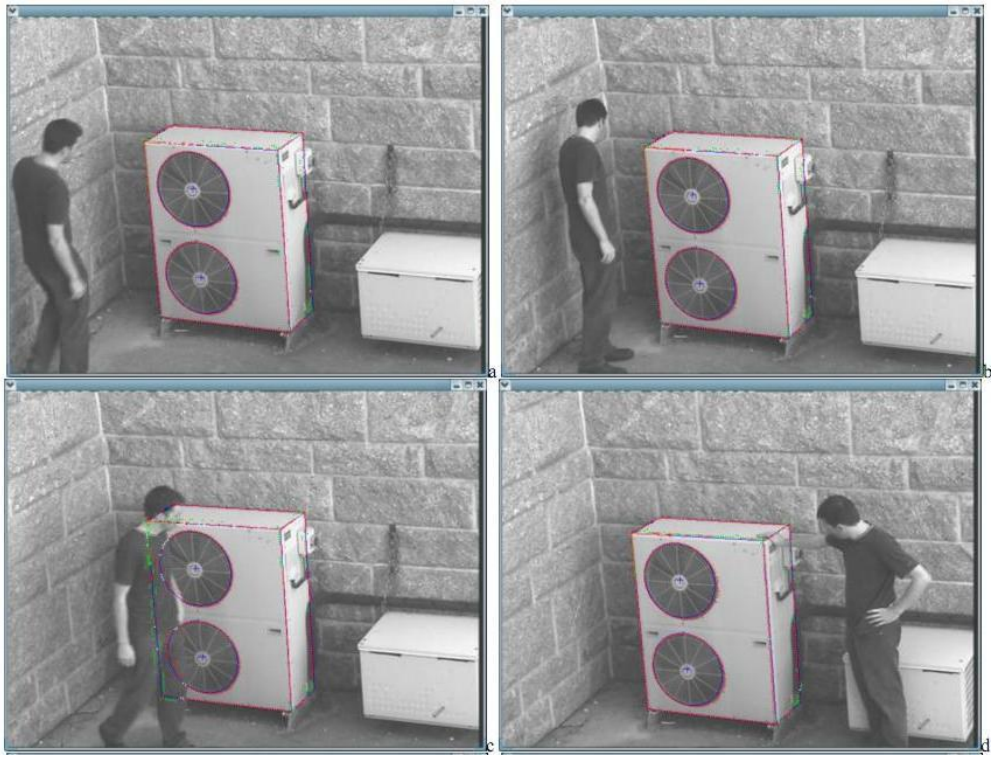


Figure 2.4 The results of markerless tracking of geometric objects in images from **[17]**

Another approach for locating and characterizing features in a scene is through the use of structure from motion. This technique, presented in [18], takes as input a sequence of images that were taken of a given area from different viewpoints. A factorization method is then used to resolve the shape and motion of the features in the sequence. The results of this method are useful for constructing a 3D rendering of the scene.

Marker Tracking

While these techniques work well for locating features in an image, visual tracking becomes easier when it is possible to place known markers on objects of interest. For the problem being addressed in this thesis, the use of fiducial markers becomes attractive. Because the system is attempting to track an object that is already known, it is possible to place a specific marker on top of that object. There exists a significant amount of research on this specific area of marker tracking with computer vision.

ARTag, a marker-tracking system presented in [19], uses digital coding theory to identify various different markers within an image. An example of these markers is shown in Figure 2.5. While the ARTag system produces a low number of false positives, the markers become difficult to identify if they occupy a relatively small amount of image pixels. Given the distance at which the tracking system must be capable of locating the ground robot, such a high pixel requirement is undesirable.

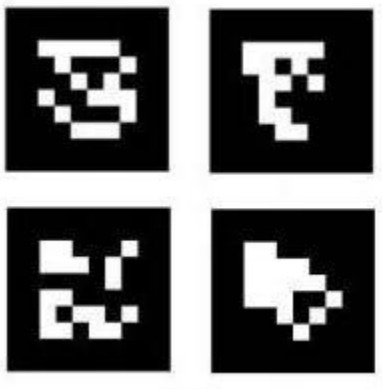


Figure 2.5 Some tags used with ARTag **[19]**

Another software system designed for marker tracking is ARToolkit, which uses simpler markers. As discussed in Chapter 1, the original intent for this toolkit was for use in augmented reality systems, but it functions remarkably well in its marker tracking capabilities. The authors of [20] present an optical tracking system that uses ARToolkit to locate the positions of various markers (similar to those in Figure 1.1) on a human body for the purpose of motion capture. Their system tracks the location of each marker across a four-frame window, and uses the translation of the marker across these frames to

predict and confirm the position of markers in successive frames. By performing this check, they are able to track multiple markers simultaneously in real-time. Another use of ARToolkit was in the creation of a video-based conferencing system [21]. This system uses a simple white board with markers placed on it, and resolves the position of the camera relative to these markers. Using this system, several users in remote locations can "share" a single board by having the markup from others displayed to them through a head-mounted display (HMD); an example of the scene as viewed through one of these HMDs is shown in Figure 2.6. The markers in this image have been highlighted by the author with red squares. This system demonstrates the capabilities of ARToolkit for resolving camera position and pose relative to markers in a scene.

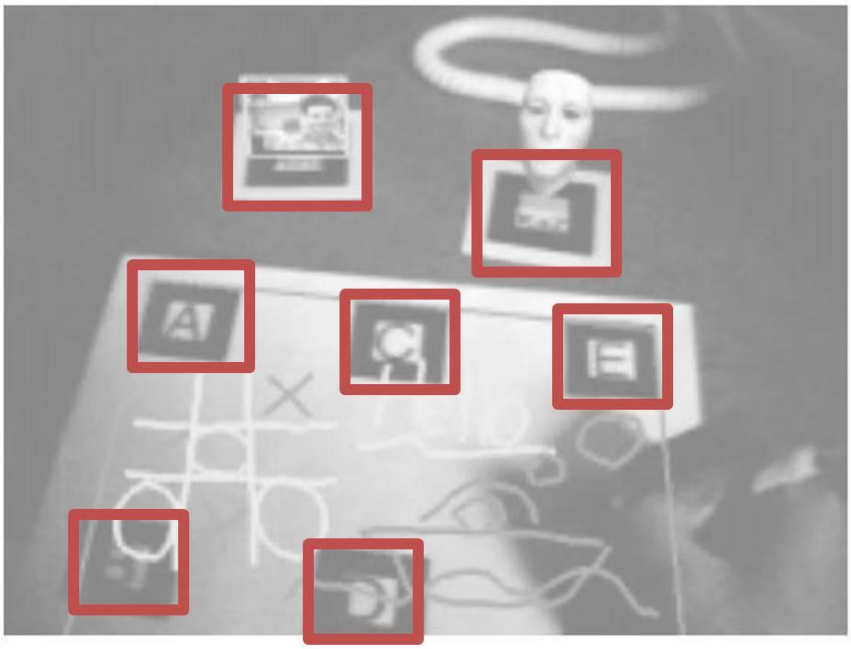


Figure 2.6 Virtual shared whiteboard using ARToolkit marker detection **[21]**

While ARToolkit's marker tracking capabilities appear to be well-suited for the task of tracking the ground robot, the operating range could become an issue. Most applications that use ARToolkit are intended for short-range (less than 10 meters) detection of objects, as opposed to tracking features tens of meters away. To this end, the authors of [22] propose a method for extending the range of marker tracking systems by combining them with natural feature tracking. Because markers further from the camera

are more likely to be lost in some frames, it is important to have a good means for reacquiring them. The authors propose doing this by matching natural characteristic points between frames, using the template technique discussed above. The movement between frames can then be estimated, allowing for the estimation of the marker position even when it is not found in a particular frame. Although this particular technique was not employed in the research done for the system discussed in this thesis, it provides some suggestions for possible future improvements.

Chapter 3
Finding Suitable Deployment Sites

This chapter focuses on the central component of the situational awareness system: the ability to determine from visual data those areas of the environment that are best suited for deploying a tethered ground robot. In this case, the visual data is provided by a stereo vision camera system from Point Grey Research. The point cloud generated by this system provides values for each point found by both cameras. These values consist of three position coordinates *x*, *y*, and *z*, which represent the 3D location of each point, along with the RGB color values of each point. The *x* and *y* positions are the horizontal and vertical distances, respectively, of each point from the center of the image in its original orientation, and the *z* value represents the distance from the camera to the point. By using this point cloud data, the situational awareness system is capable of determining the suitability of regions within the image for deploying the ground robot. Each section within this chapter will focus on an individual component of this system.

3.1 Prior Work

The developments discussed in this chapter build directly off the research performed by Dylan Klomparens for his thesis [6]. His research focuses on the use of point cloud data for determining suitable landing sites for a helicopter. However, some of his work proves useful toward the determination of good deployment sites for ground robots. His process begins with the partitioning of the point cloud data into thirty rows and columns relative to the reference camera image. The multiple linear regression of each region is then performed, generating a best-fit plane for that region. Given a set of m points (x_i, y_i, z_i), where i represents each point from 1 to m, the equation of the best-fit plane is given by $z = Ax + By + C$ for all points in the plane, where coefficients A, B, and C are found by solving the following equation:

$$\begin{bmatrix} \sum_{i=1}^{m} x_i^2 & \sum_{i=1}^{m} x_i y_i & \sum_{i=1}^{m} x_i \\ \sum_{i=1}^{m} x_i y_i & \sum_{i=1}^{m} y_i^2 & \sum_{i=1}^{m} y_i \\ \sum_{i=1}^{m} x_i & \sum_{i=1}^{m} y_i & \sum_{i=1}^{m} 1 \end{bmatrix} \begin{bmatrix} A \\ B \\ C \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^{m} x_i z_i \\ \sum_{i=1}^{m} y_i z_i \\ \sum_{i=1}^{m} z_i \end{bmatrix} \quad (2)$$

Once the best fit planes have been defined for each region, the ones suitable for helicopter landing sites are determined through analysis of the slope of each plane along with the slope between neighboring planes. Those with acceptably level planes are marked as suitable, while those with gradients too extreme are rejected. Figure 3.1 shows a side-by-side comparison of a set of best fit planes (with the suitable planes colored green and the rejected planes colored red) and the 2D image of the terrain being analyzed (with the acceptable planes overlaid in blue). It should be noted that the cameras used for capturing these images distort the edges of the image, so the center of the image is the most reliable for analysis.

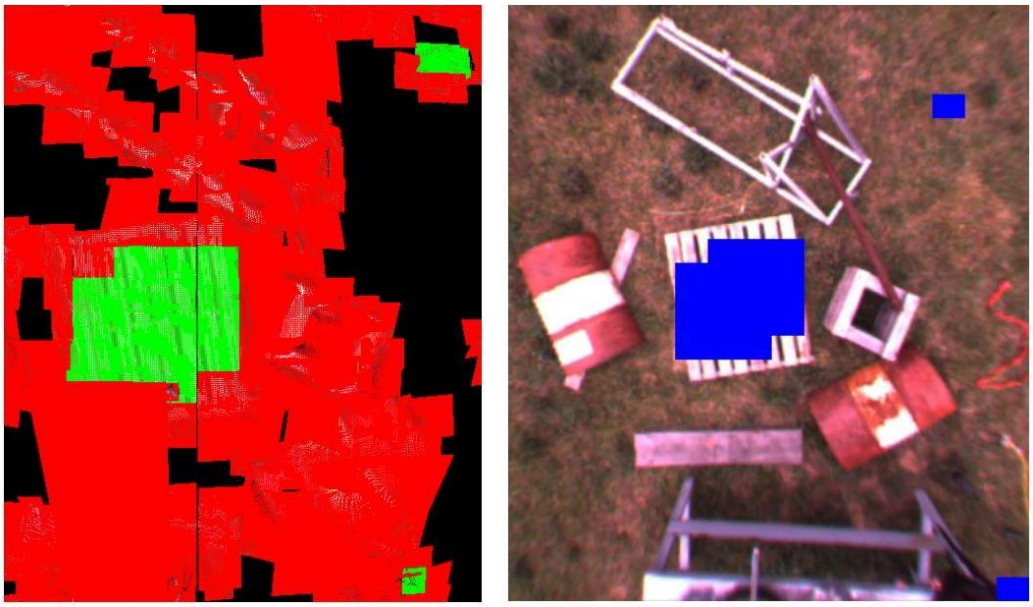


Figure 3.1 Side-by-side images showing results of best-fit planes from **[6]**

Clearly the work done by Klomparens serves as a useful starting point for finding regions in a stereo image pair suitable for deploying a ground robot. The characteristics of the terrain necessary for the ground robot share some similarities with those needed for landing a helicopter. Specifically, the grade of the terrain must not exceed a specified value able to be handled by the ground robot, a property that may be discerned using the same methods already described. However, there is other information that must be known about the area as well.

3.2 Detecting Utility Cables

A significant feature of the situational awareness system presented here is its ability to detect utility cables in the stereo image data. As mentioned in Chapter 2, the widespread presence of utility cables contributes to a significant risk of entanglement with the ground robot's tether. This danger can be mitigated by developing a reliable means for detecting utility cables from the helicopter's vision system.

Edge Detection

Given the relatively linear nature of utility cables, it was hypothesized that an image processing algorithm capable of detecting lines would find the utility cables. The first step in this process was to decide what parameters of each point would be analyzed. Feature detection in 2D images typically uses pixel intensity, or grayscale value, to find edges. By tracking areas in the image where a sharp intensity gradient exists, one can reasonably infer the location of edges.

While this same procedure could be used on the point cloud data, it would neglect the extra information gathered from stereo imagery; namely, the distance of each point from the camera. Instead, the *z*-position of each point is used in place of pixel intensity, similar to the technique employed on the range image data from [13]. In this manner, rather than finding areas with a high gradient of color contrast, an edge detection algorithm will locate areas in the image with a high gradient of distance from the camera.

This means that building edges, elevated wires, and other sharp drop-offs should be picked out.

To detect points in the image lying near these high elevation gradients, a Sobel edge detection algorithm is used. This edge detection scheme uses two 3×3 convolution masks, shown in Figure 3.2. The G_x mask is designed to respond maximally to edges running vertically through the image, whereas the G_y mask is designed to respond maximally to horizontal edges[23]. These masks are applied to each point in the point cloud, giving a vertical gradient value from G_x and a horizontal gradient value from G_y.

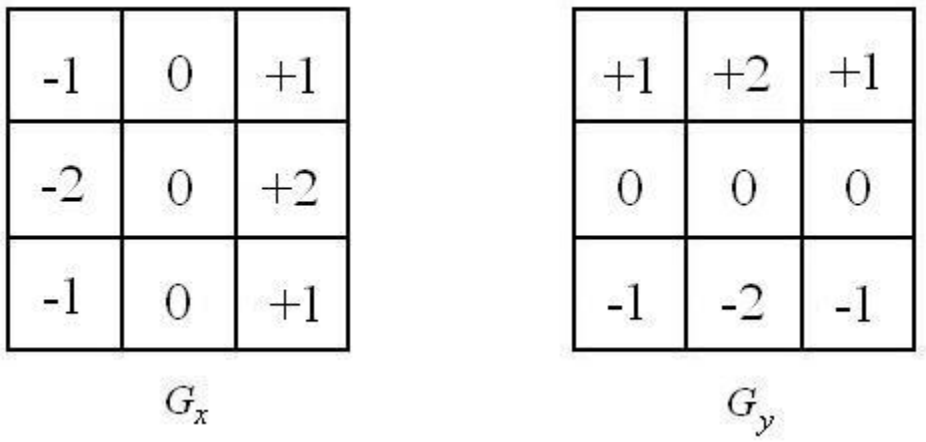


Figure 3.2 Sobel edge detection 3x3 kernels

Using the following equation produces a total gradient magnitude that represents the level of elevation drop-off at that point:

$$|G| = \sqrt{G_x^2 + G_y^2} \tag{3}$$

In this particular application, however, the equation below will be used, as this choice reduces computation time while giving a fair approximation of the gradient magnitude for each point:

$$|G| \approx |G_x| + |G_y| \tag{4}$$

The set of points with gradient magnitudes above a specified threshold are then passed on to the Hough transform.

Hough Transform

The Hough transform[9] serves as a means for parameterizing point data in an image such that it becomes computationally simpler to locate geometric features in the image. Typically, the points input to a Hough transform are not the raw image data but rather a set of image points that have been detected in some way. This typically comes in the form of an edge detection algorithm, as is the case here. Given this set of data, the Hough technique attempts to find points that form geometric shapes, in this case a straight line. The method used by Hough transforms to locate lines is briefly described here.

Traditionally, one thinks of line equations in terms of Cartesian space. Given a point (x, y) on the Cartesian coordinates, that point belongs to an infinite number of lines following the equation:

$$y = mx + b \tag{5}$$

where m represents the slope of the line and b is the point at which the line crosses the y-axis. However, in considering the points in an image, the slope m of a line covers an infinite range, making it difficult to use. Alternatively, one can represent lines through parameterization from the graph shown in Figure 3.3. Given a shortest distance from the origin r and the orientation θ of the line r with respect to the x-axis about the origin, the set of lines containing (x, y) can be represented by the curve resulting from

$$r = x\cos\theta + y\sin\theta \tag{6}$$

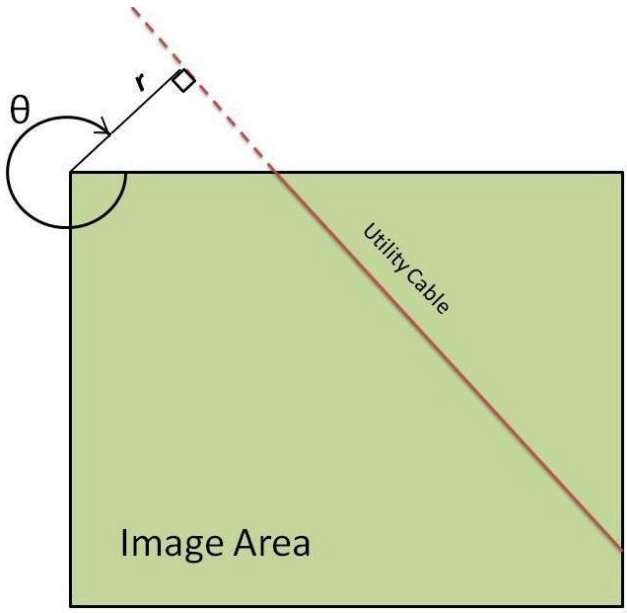


Figure 3.3 Parameterization of a line in r-θ space **[9]**

In Figure 3.4, an example of Hough transform use is shown. The graph on the left shows three points in Cartesian space, and the graph on the right shows the curves in Hough space that illustrate the corresponding lines containing each point. To identify the parameters that define the lines containing multiple points, one simply finds the intersection of the curves in Hough space. In this example, one can see that all three curves in the graph on the right share an intersection. By finding the (r, θ) values of this intersection point, the parameters of the line defined by those points can be determined.

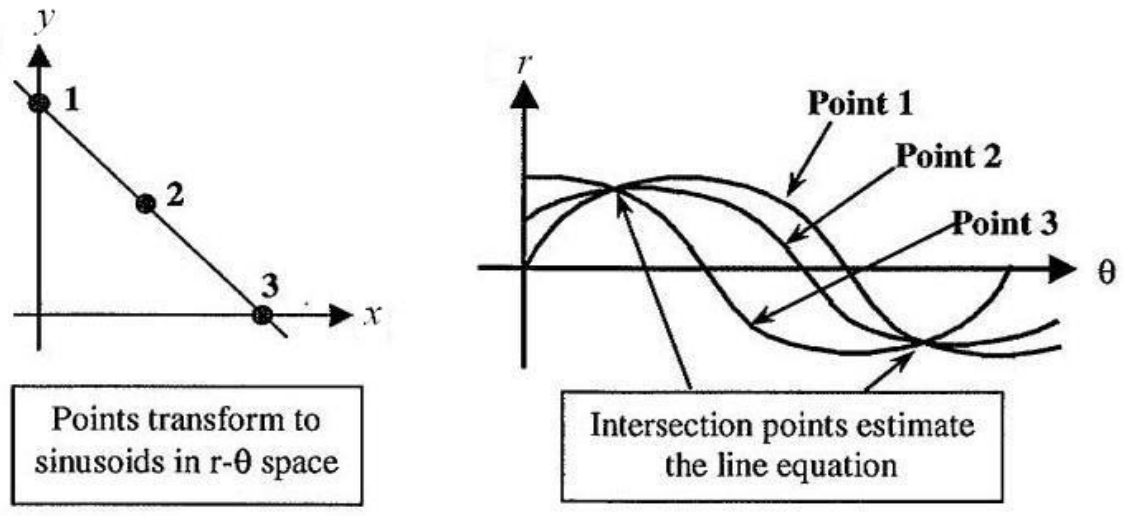


Figure 3.4 Graphs showing points in Cartesian space and their corresponding Hough curves **[9]**

This method for detecting lines in an image proves useful for detecting utility cables. By taking the size of the image into account, one can determine the possible range of values for r where r is always positive, and the range of values for θ is defined to be from 0 to 2π. A 2-dimensional accumulator array is set up such that each entry in the array corresponds to a specific r and θ value. These values are distributed evenly across the range of all values for each parameter. After the Sobel edge detection has been run, the points determined to be edges in the image are passed to the Hough transform. The algorithm steps through each of these points, incrementing the values in the accumulator array at each (r, θ) index that lies on the corresponding curve. This increment is commonly referred to as a "vote" for the line defined by those parameters. When the Hough transform is complete, each (r, θ) position in the accumulator array contains an integer value corresponding to the number of votes for that line.

Consider the scene shown in Figure 3.5. This represents terrain that may be suitable for deployment of a tethered ground robot. A utility cable was then draped across it, and a stereo system was used to gather a point cloud of the scene, which is shown in Figure 3.6. For this example, it is critical that the vision assessment algorithms detect the existence of this potentially dangerous obstacle. The point cloud is converted to a range image by defining z-values for each (i, j) position in the image. The process for defining the values at the gaps in the point cloud is discussed below. Figure 3.7 shows the output

of the Sobel edge detection algorithm run on this range image. For this example, the threshold value used is 400. This edge detection output is run through the Hough line detection algorithm, with all lines receiving at least 110 votes colored red in Figure 3.8.

Figure 3.5 Photograph of example scene used for stereo imaging

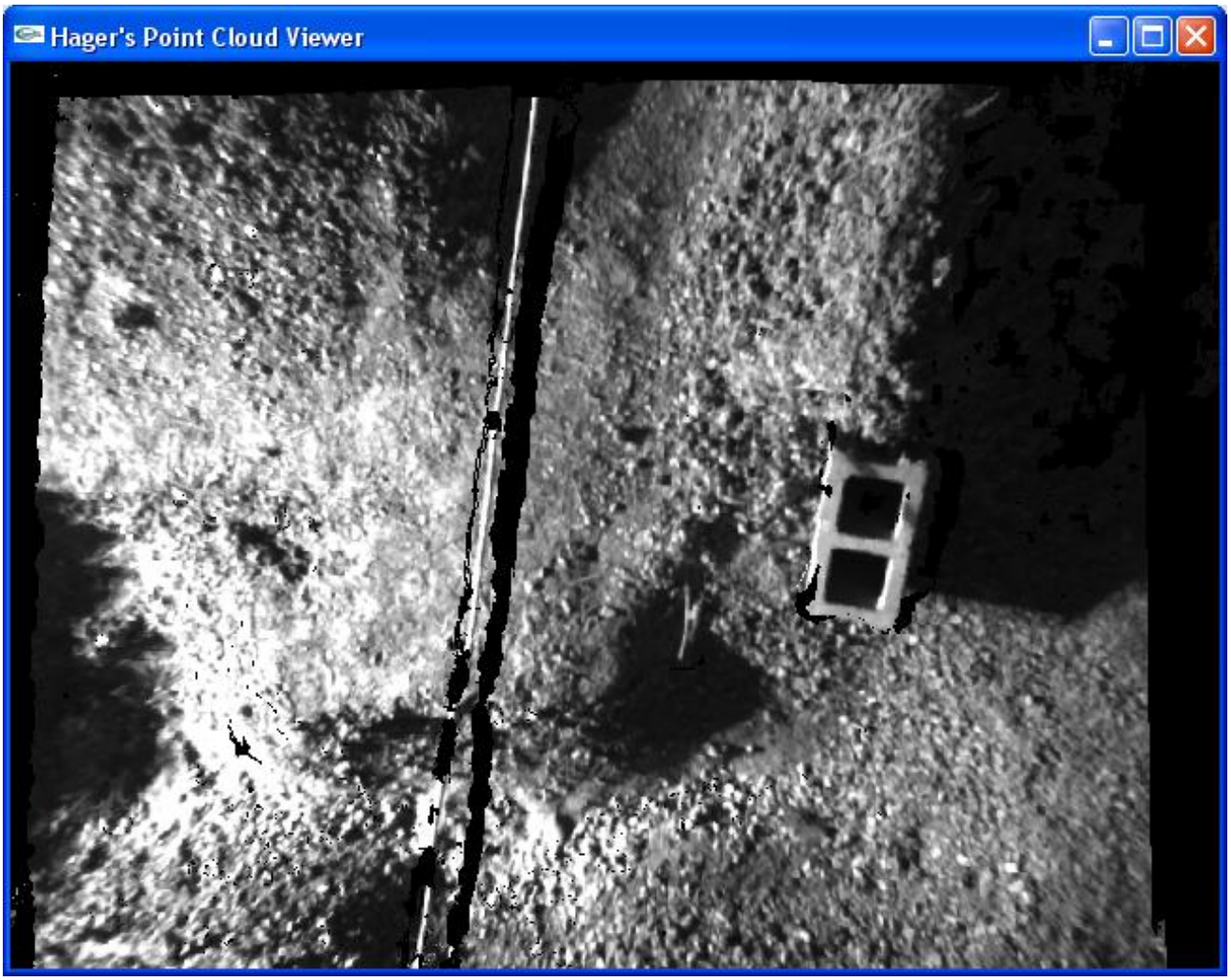


Figure 3.6 Example point cloud with a cable draped across the middle of the area

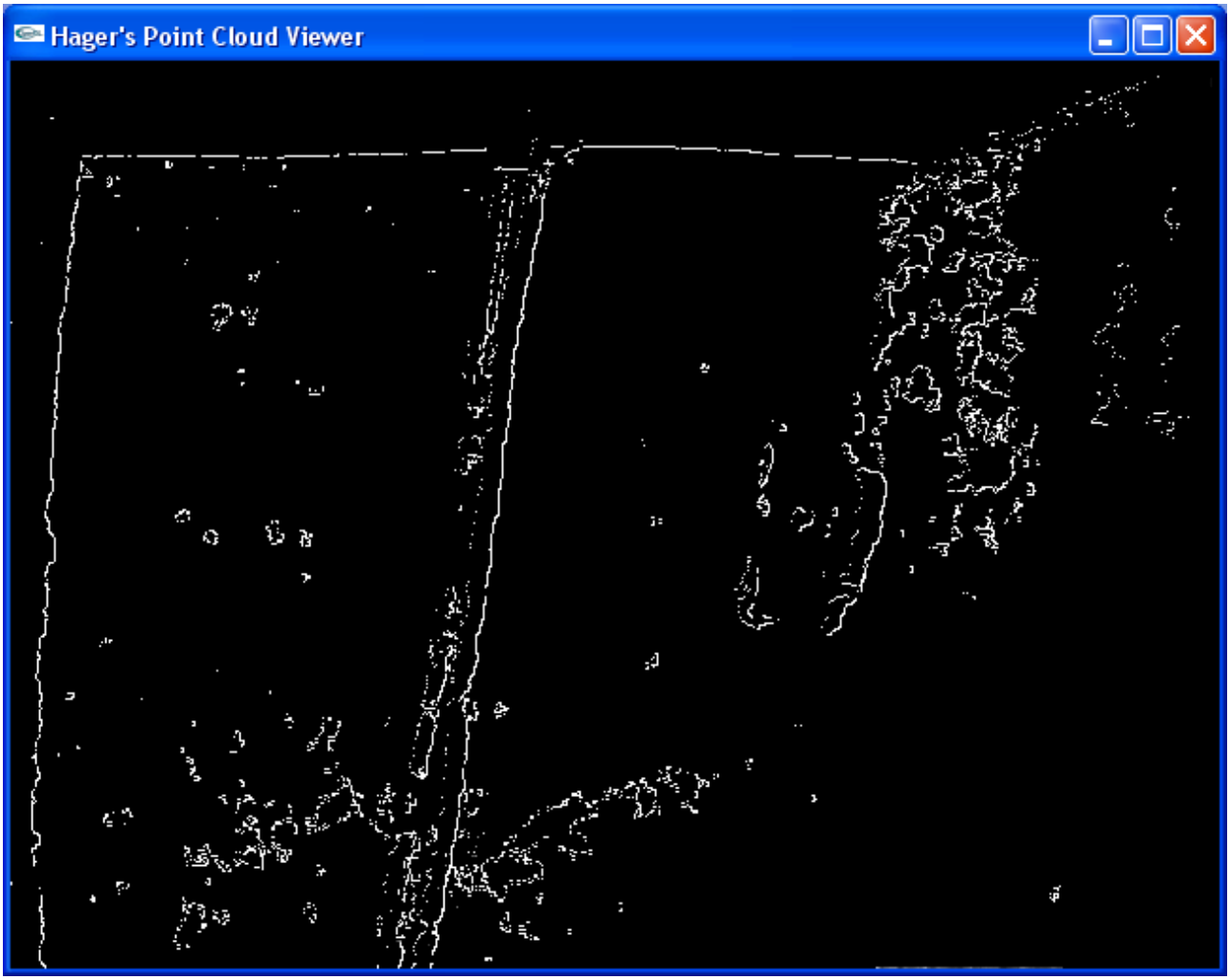


Figure 3.7 Output of Sobel edge detection run on example range image

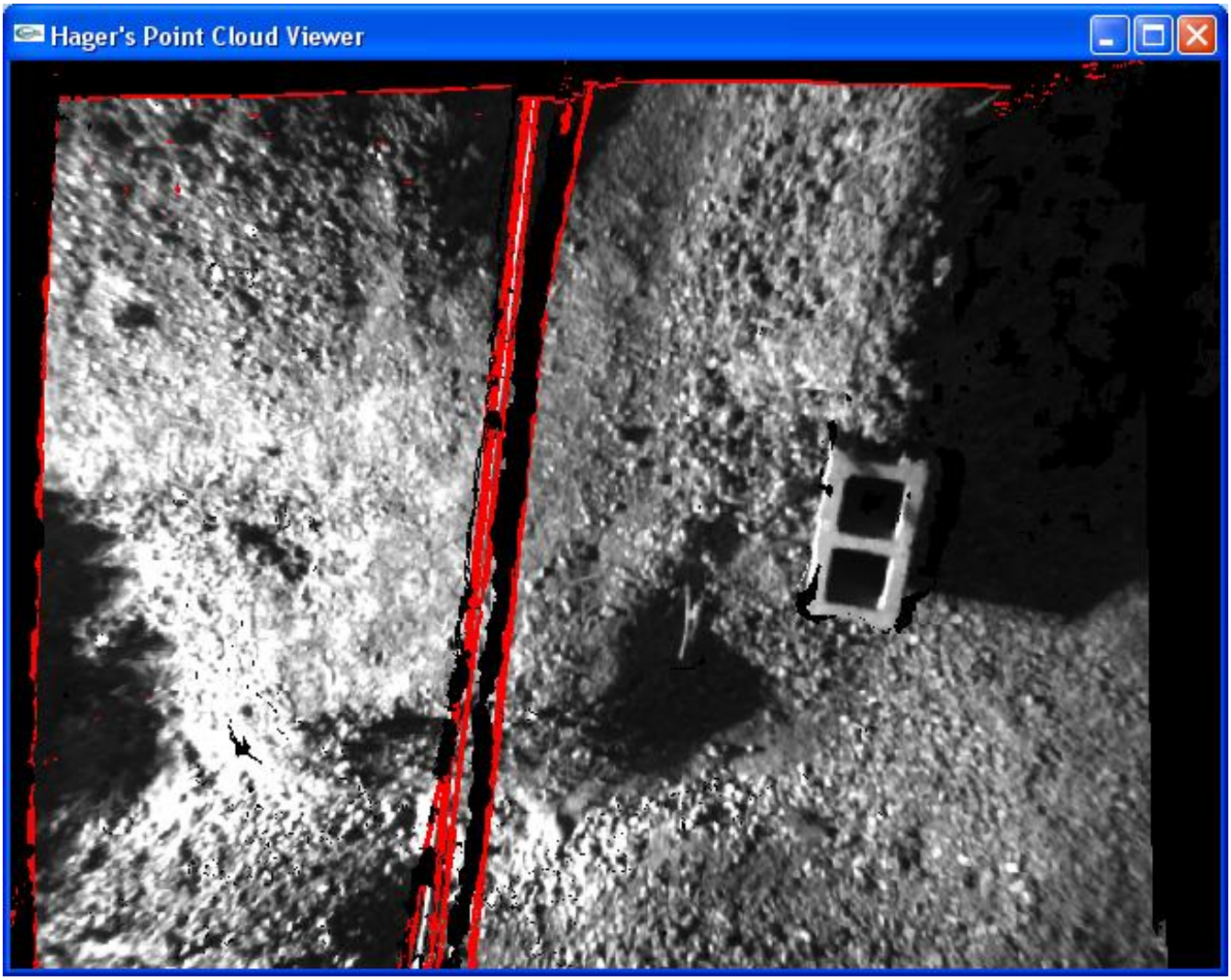


Figure 3.8 Example terrain scene with possible utility cables colored in red

Utility Cable Experiment

Due to the nature of the mission for which this technology was developed, it is important to be capable of detecting utility cables regardless of their orientation. When looking at a specific set of terrain, there is no guarantee as to the orientation of the utility cables or their relative distances to the camera. For this reason, it was clear that an experiment needed to be conducted to test the system on a multitude of cases, which examine various angles and heights of a utility cable. The set-up for this experiment required the ability to rigidly mount a camera looking straight down at a scene, as well as

the capability for draping a simulated utility cable across the terrain at different heights and angles relative to the camera.

Behind the Unmanned Systems Laboratory, a simulated terrain area was created using simple ditches, dirt mounds, and cinder blocks, which was shown in Figure 3.5 above. The area is bordered by tall support beams that provide an excellent means for suspending the stereo camera system directly over the middle of the scene using a support beam running from one end to the other. Using the camera mount pictured in Figure 3.9, the cameras were placed over the terrain from a nadir perspective. The system pictured is the three-camera version of Point Grey's Bumblebee stereo camera system, as opposed to the two-camera version which was shown in Figure 1.2. The images were acquired using the widest baseline, or distance between cameras. Using the widest baseline is accomplished by using the two cameras on either end of the camera, which are 24 cm apart. This known distance is used by the Point Grey software to calculate the disparities of points in the image.

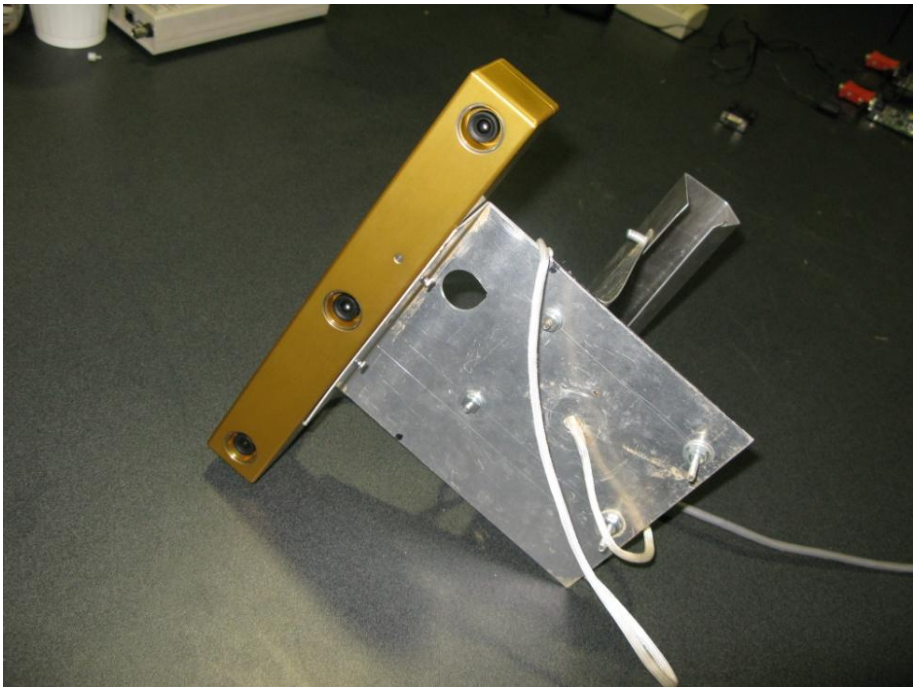


Figure 3.9 Camera mount used to suspend Bumblebee over scene

To simulate a utility cable for these images, a length of thick (approximately 0.016 meters in diameter) cable was used. This cable was tethered at either end to a pair

of high-gain antennas, as shown in Figure 3.10. These antennas are then positioned across the scene from one another such that the cable will pass through the scene being analyzed by the stereo system. Between each image acquisition, the antennas are moved such that the angle θ is increased by about 30 degrees. This angle θ is defined as the rotation about the z-axis (defined as going straight down into the ground). This relationship is illustrated in Figure 3.11. After the cable was rotated through 180 degrees, the height of the cable was adjusted and the rotation performed again.

Figure 3.10 Utility cable simulation setup; camera setup can be seen near top of photo

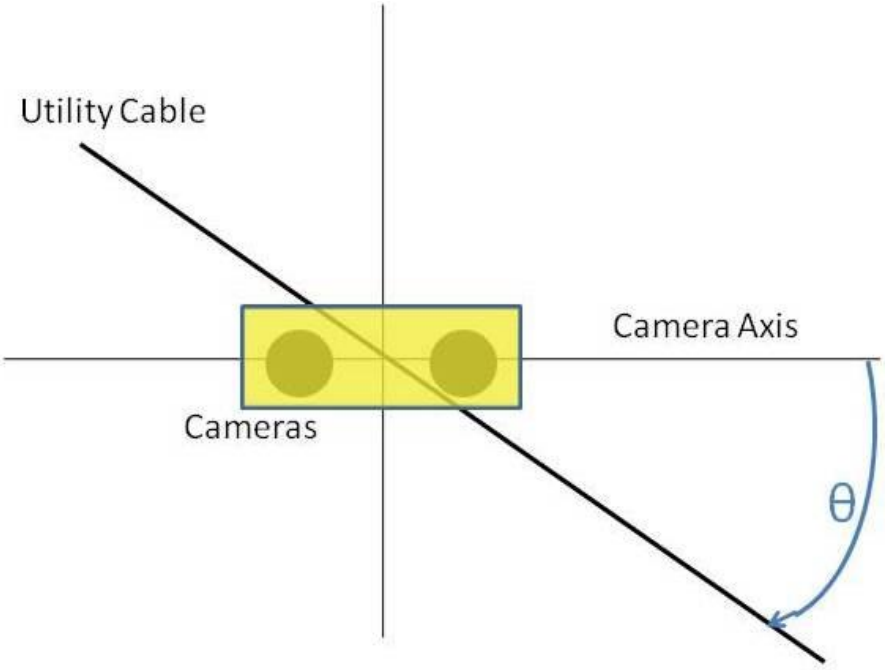


Figure 3.11 Illustration of utility cable rotation

During the experiment an attempt was made to take image pairs at even intervals, although some error in angle was inevitable. After all the images were acquired, a protractor was then used to measure the value of θ contained in each picture. Table 3.1 contains a list of images taken, along with the associated θ and z values (representative of the vertical distance from the cameras to the utility cable).

Table 3.1. List of stereo images taken with a single utility cable.

Image ID	z (meters)	θ (°)
0_H1	1.09	0
30_H1	1.09	25
45_H1	1.09	52
60_H1	1.09	66
90_H1	1.09	88
120_H1	1.09	109
135_H1	1.09	132
150_H1	1.09	154
0_H2	0.74	0
30_H2	0.74	20
45_H2	0.74	47
60_H2	0.74	58
90_H2	0.74	89
120_H2	0.74	118
135_H2	0.74	135
150_H2	0.74	154

In addition to those described in Table 3.1, there were also a few stereo images taken of a scene containing two cables crossing one another. As there could conceivably be intersecting utility cables in the area to be explored with the system, it is important to be able to detect multiple lines simultaneously, or through some defined process. For this portion of the experiment, two different setups were used: one with two utility cables perpendicular to one another, and one with two utility cables forming 45 degree interior angles. Each of these set-ups was arranged such that one utility cable would lie at $\theta = 0$ degrees, and a picture was taken from this position and then with the utility cables rotated 45 degrees. Table 3.2 describes the z and θ values for both utility cables in each of these images.

Table 3.2. Description of stereo images with two utility cables

Image ID	θ_1(°)	z_1 (meters)	θ_2(°)	z_2 (meters)
Cross_45_1	179	1.09	50	0.74
Cross_45_2	86	1.09	46	0.74
Cross_90_1	45	1.09	136	0.74
Cross_90_2	180	1.09	85	0.74

Appendix B shows the complete results of utility cable detection performed on each of the images taken from this experiment, but a summary of the findings is discussed here. As explained in Chapter 1, stereo vision works by finding disparity in feature point locations from images taken by side-by-side cameras. However, when visual texture is not present for an area of interest, it becomes difficult for the software to detect correspondences for that object. Objects that have a uniform appearance in the horizontal direction present challenges for stereo correspondence. With this in mind, it was hypothesized that as the value of θ approaches 0 degrees, the camera is less likely to detect the utility cable in the image. The scene and corresponding results from Figures 3.5 – 3.7 show a utility cable with a θ value near 90 degrees. As one can see, the system easily finds the disparity values for the utility cable in the image, allowing the utility cable detection algorithm to accurately locate the relatively horizontal object.

By contrast, the scene in Figure 3.12 contains a utility cable where θ is near 0 degrees. The point cloud in this image has been rotated to show that the camera system did not locate an elevated cable in the scene, although there was a cable draped across the center of the scene. However, some effects from the utility cable are still visible. For instance, there are gaps in the point cloud that lie along the axis of where the utility cable is located. These occur when the two cameras fail to find matching feature points in a region of the image. This would be expected to occur when an object is obscuring the cameras from viewing the ground, and when the cameras cannot identify features on the offending object as well. Interestingly, there are also pixels that lie on this axis that have color values approaching black. Because the *z*-value associated with these points correspond closely to those on the ground in that area, these would seem to be features found within the shadow of the utility cable. While this data could potentially be used to

find lines in the image, it would result in far more false positives than does the algorithm based on *z*-values.

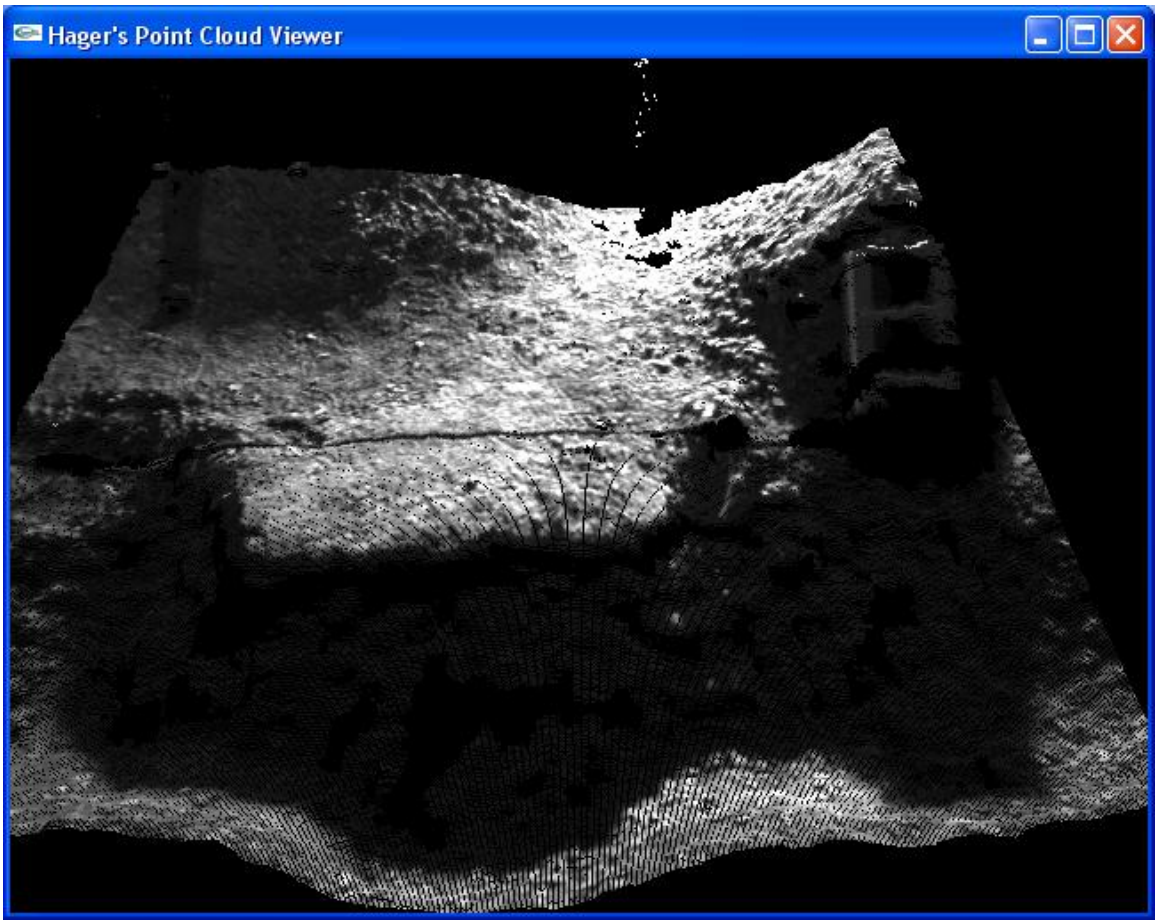


Figure 3.12 Point cloud from scene where $\theta \approx 0°$; although a cable exists near the center of the screen, it was not properly detected by the stereo system

Determining Parameter Values

The completion of the image-gathering process provided a sufficiently large set of data on which to test the utility cable detection algorithm. It then became necessary to determine what specific parameters provided the most consistently accurate results across the range of image inputs. Ideally, these parameter values should always locate any utility cables in the image, without generating any false positives. In reality, it is difficult to avoid occasional false positives, especially those generated by some of the effects caused

by the presence of an elevated object. However, such effects are often useful to highlight as they may serve as a good indicator of the presence of utility cables.

One of the first issues that must be addressed is the handling of "holes", or missing points in the point cloud. These holes occur when the system is unable to determine stereo correspondences. As mentioned previously, this can occur when an object in the foreground occludes some part of the scene from one of the cameras. When converting point clouds to range images for the purpose of examining the gradient of areas within the images, it is necessary to define the *z*-values of these gaps. In this experiment, a resolution of 640×480 pixels was used for each camera. Ideally, each pixel would be correlated to both images, which would give a total of 307,200 points in each point cloud. However, on average about 80% of this number of points are correlated; this leaves 20% of the image where holes must be "filled".

One early approach to this issue was to compute the mean *z*-value of all points in the point cloud, and define the position of gaps in the image to be this average value. However, this method proved unsuccessful in that it fails in the attempt to approximate *z*-values for a given region of the point cloud. The mean *z*-value of the point cloud can be biased by the presence of outliers, and there is also no guarantee that the holes lie in parts of the terrain that have positions relatively close to the mean for all points. The next approach was to set the *z*-values of these gaps to be the same as those of the neighboring points. However, this technique proves to be imperfect as well. The assumption behind it is that the gaps should be masked, and thus not produce any false positives by generating artificial areas of high gradient. Upon examining the example terrain scenes, however, it becomes clear that this assumption is flawed. As it turns out, in many cases these gaps serve as the primary evidence of the existence of utility cables in the scene. For this reason, it is preferable to actually "highlight" these areas as potential utility cable locations rather than attempt to mask them. To this end, these gaps are assigned a *z*-value that simulates an object suspended over the terrain; for the experiment discussed here, a *z*-value of 0.5 is used.

The next step in the parameter selection process is the choice of a good threshold value. To maintain parallelism with the pixel intensity approach to edge detection, the highest elevation points are assigned the largest *h*-values; *h* represents the height of a

point relative to the ground, whereas *z* gives the distance from the point to the camera. To find the heights, each point's *z*-value is subtracted from the mean *z*-value for the point cloud, and this quantity is multiplied by 30 to produce the *h*-value, or elevation value, for the point. These new *h*-values are then run through the Sobel edge detector, and those points with an output value higher than a pre-defined threshold are passed on to the Hough transform. The next step was then to determine what edge threshold value provided the most consistent results across all image inputs. Figure 3.13 shows a side-by-side comparison of the Sobel edge detection results as applied to Image 60_H1. Note that the grid visible in the point cloud image is a result of the partitioning by the point cloud viewer software and is not actually present in the scene. The top left and top right images use too high and too low of threshold values respectively, which results in too many false positives for the former and too few valid edges found in the latter. The bottom image, however, does a good job of identifying the regions of highest *h*-value gradient. The threshold value used to produce that image, 60, was found to achieve consistently accurate results across all single utility cable images in this experiment. Those points with a Sobel output value over this amount are then run through the Hough transform.

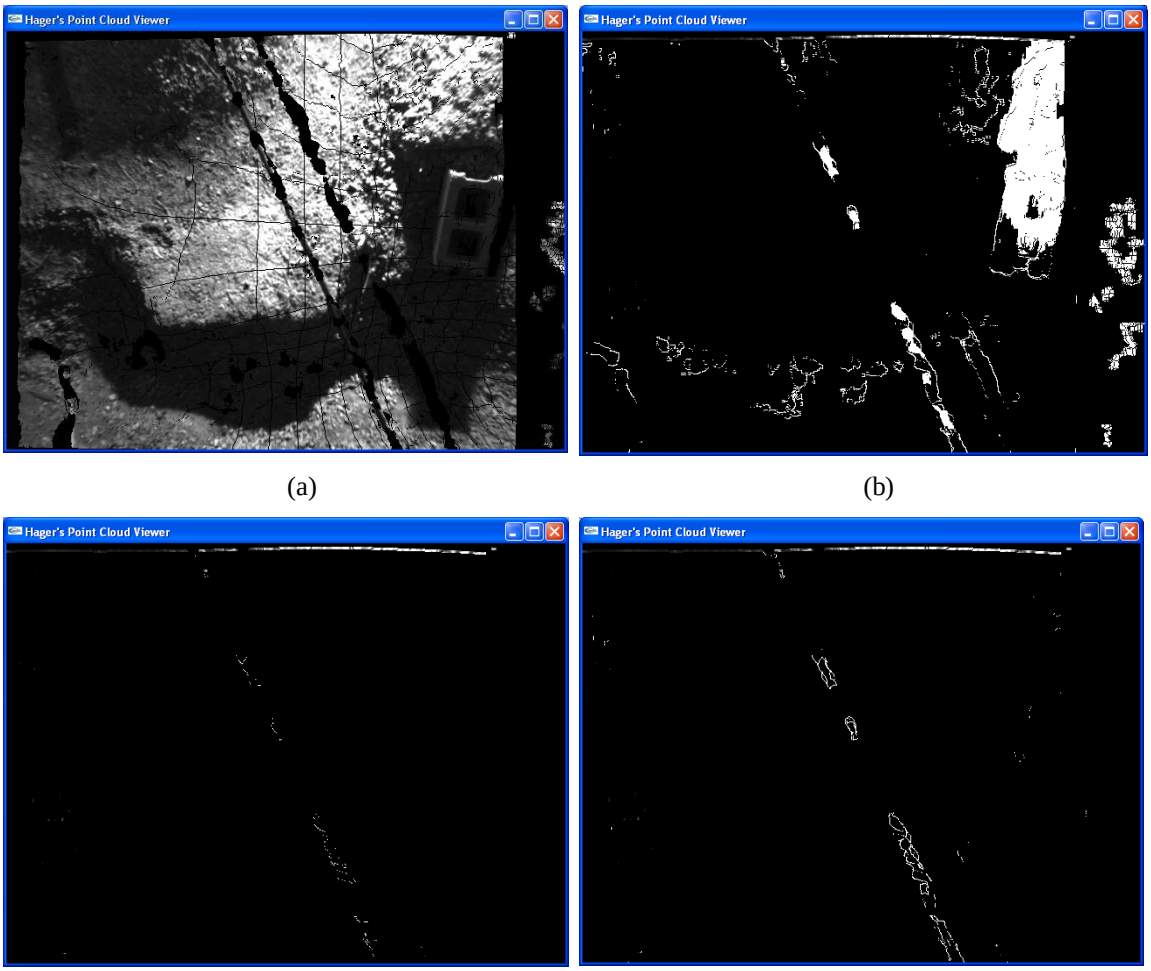


Figure 3.13 Output of Sobel routine run on (a) range image using threshold values (b) 15, (c) 145, and (d) 60

Once this process is completed, it is left to determine what quantity of votes for a particular set of line parameters is sufficient to indicate the likely presence of a utility cable. One early attempt to define this necessary vote count was through trial and error of using various minimum count values. Any lines that contained more than this pre-defined vote count threshold were marked as possible utility cables. However, this solution proved inconsistent across multiple inputs. Figure 3.14 shows a side-by-side comparison of the results from this technique as applied to two different point clouds. The top row shows the output from images 30_H1 and 60_H2 using a count threshold of 140, and the bottom row shows the outputs from these same images using a count threshold of 700. Those portions of the image found by the detection algorithm to be potential utility cables

are colored red. Clearly, each threshold value works well on one image but is useless on the other.

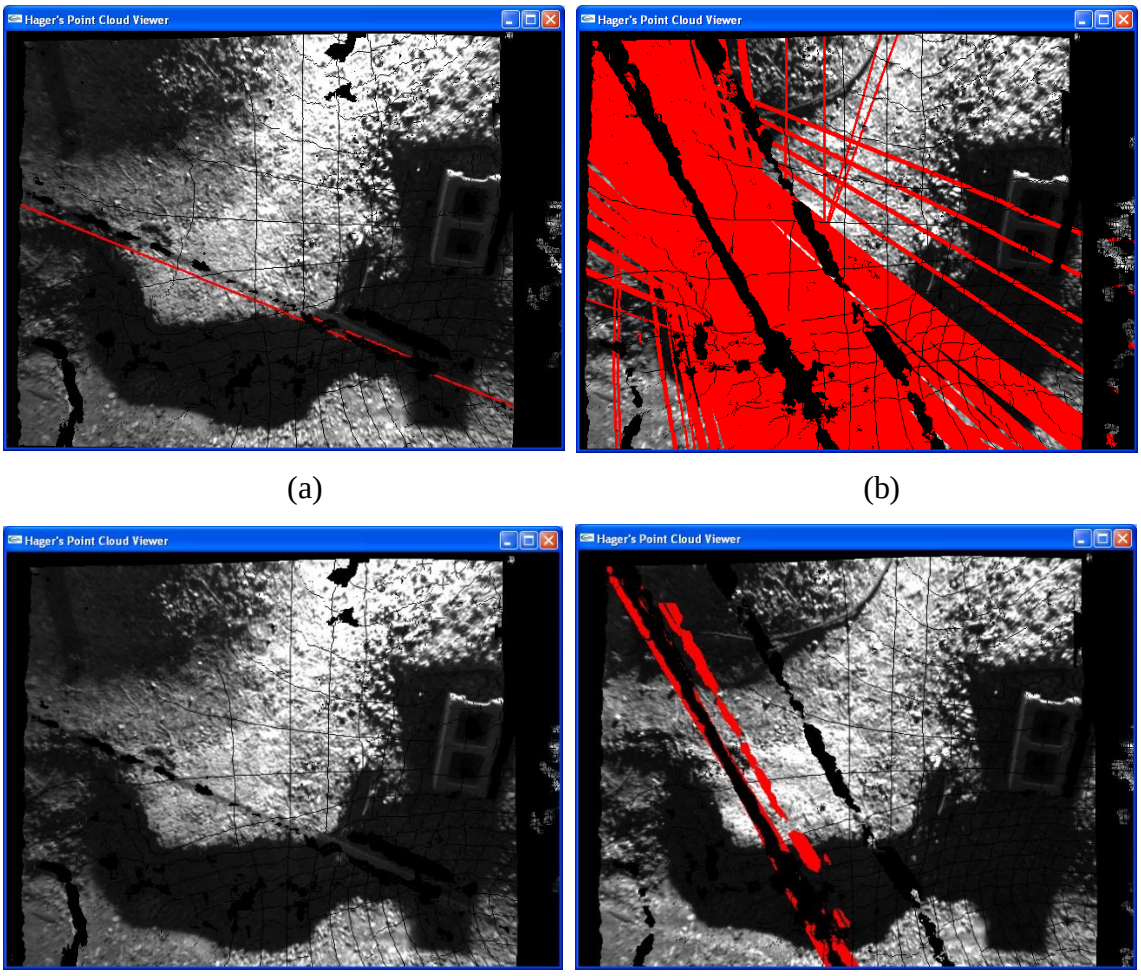


Figure 3.14 Utility cable detection output on 30_H1 (left column) and 60_H2 (right column) using vote count thresholds of 140 (top row) and 700 (bottom row)

Rather than using a pre-defined vote count threshold for locating utility cables, an adaptive method proves more effective. First, the accumulator array is searched for the line with the most votes. This most probable line is then used as a baseline for identifying likely utility cables in the image. Those lines found to have similar vote counts to the most probable line are highlighted as possible utility cables. Through the testing of several values, it was determined that 5/6 serves as a consistent multiplier *m* for locating probable lines, where *m* multiplied by the highest number of votes for any line gives the

vote count threshold for considering a line to be a potential utility cable. Therefore, all lines whose corresponding accumulator array position received at least 5/6 the number of votes as the highest vote-getter are marked as potential utility cables. In Figure 3.15, the output from this technique as applied to the same images from Figure 3.14 is shown; the detected utility cables are colored red. These results are clearly far more consistent than those shown above.

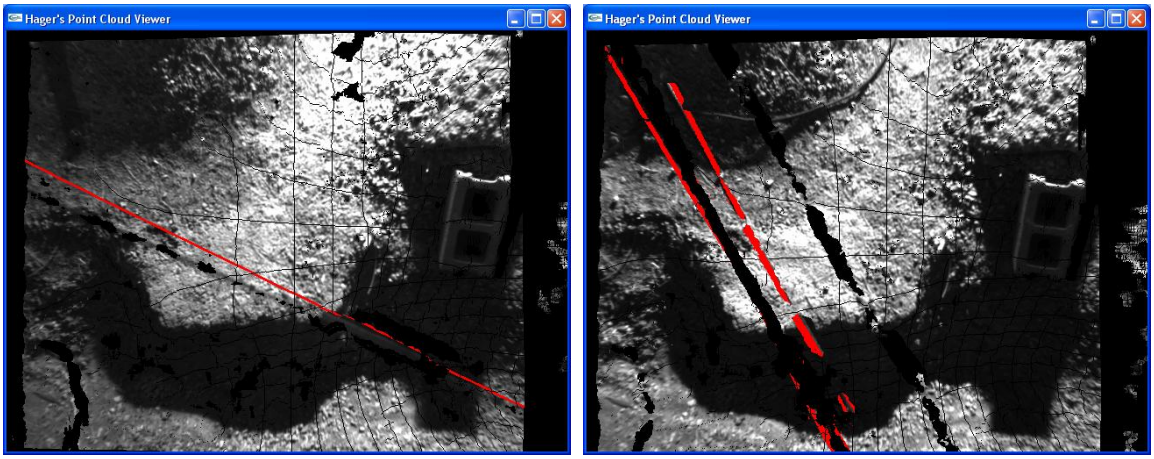


Figure 3.15 Utility cable detection on 30_H1 (left) and 60_H2 (right) with adaptive threshold method

One drawback of this adaptive approach becomes obvious when it is applied to a scene with no utility cable obstacle, or when the utility cable has a relatively low vote count. When this occurs, the points lying on the nearest-to-a-line object are selected, even if the votes for these features are very low. Figure 3.16 shows an example of this; although the image does not have any highly probable lines, the adaptive threshold method outputs those points closest to a line anyway. To avoid this, a minimum vote count is included in the requirements for a feature to be identified as a possible utility cable. The value for this minimum quantity is dependent on the resolution and quality of the imagery; the results shown in Appendix B were produced using a minimum vote count of 130.

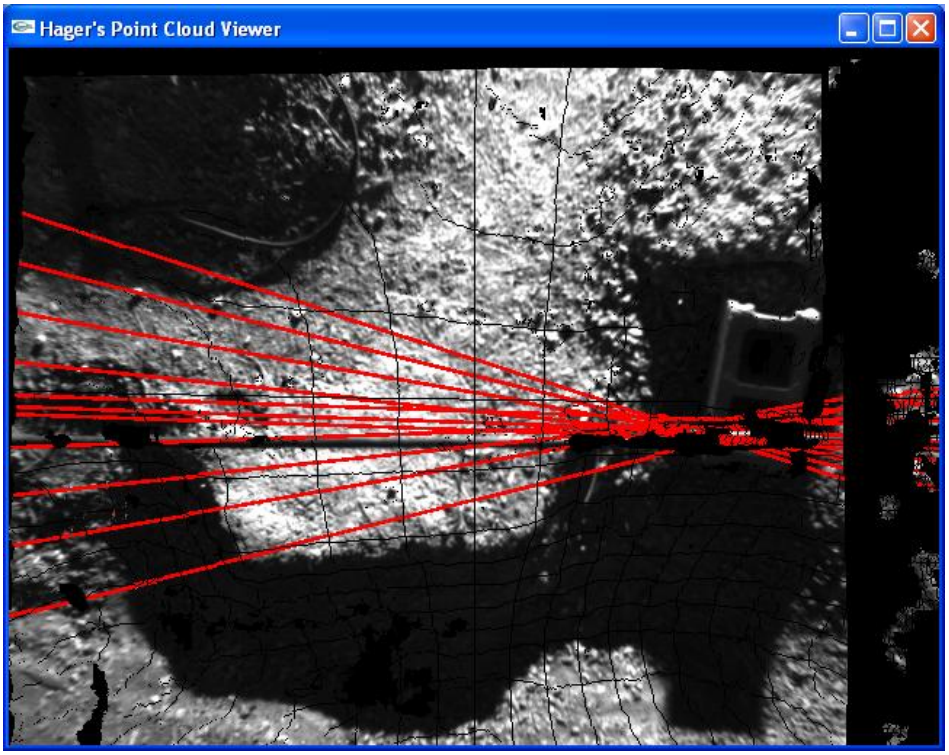


Figure 3.16 Utility cable detection output on 0_H2 when no utility cable was resolved by the stereo system and no minimum vote count is used

Although the parameters and constraints discussed above consistently locate single utility cables in images, they fail to produce reliable results when more than one utility cable exists in an image. In a given scene, one utility cable may be detected with far more points than those associated with another utility cable in the image. For this reason, the adaptive approach will locate only the line with the most votes. While other lines could be detected by lowering the value of *m*, this will also result in more artifacts around the first line and increase the likelihood of false positives. However, by maintaining a relatively high multiple factor of 4/7, the risk of false positives is reduced. In addition, only those lines with *r* values significantly different from the most probable line are returned, to reduce the amount of unnecessary artifacts around that line. Through experimenting with various values, it was found that selecting those lines with a difference of at least 50 in the *r* parameter value provided the best results. In summary, a line in the image is identified as potential utility cables if:

(1) The line received at least 130 votes in the Hough array

AND

(2) Either:

a. The accumulated vote count is at least 5/6 that of the highest vote-getting line

OR

b. The difference of the *r* parameter of the line and that of the highest vote-getting line is at least 50 AND the accumulated vote count is at least 4/7 that of the line with the most votes.

Results and Implications in Mission Planning

Using the parameters and techniques discussed above, a reasonably consistent method for detecting utility cables in stereo imagery has been produced. For each image in the test of 20 cases, elevated cables were properly located and highlighted in red. Linear objects lying on the ground were not picked up by this method, eliminating false positives that occur with traditional 2D image line detection that is based on pixel intensity. In images with two elevated cables, both were found, and one is highlighted in red while the other is colored purple. The only exception to this reliability in detection occurs when a utility cable is approximately horizontal relative to the image. As expected, utility cables that have θ values near 0 degrees cannot be reliably detected with this method, due to the difficulty of finding pixel correspondences on horizontal objects with uniform texture. However, this weakness can be overcome by correct mission planning. One advantage that helicopters provide over other aircraft is the capability for precision hovering. By hovering over a target scene and taking two separate images of it, having rotated the aircraft to change the value of θ by 45 degrees for all features in the scene, one can greatly reduce the risk of missing utility cable features due to their appearing horizontal in the images. The interface for assessing the results of the double image capture is discussed in Chapter 5.

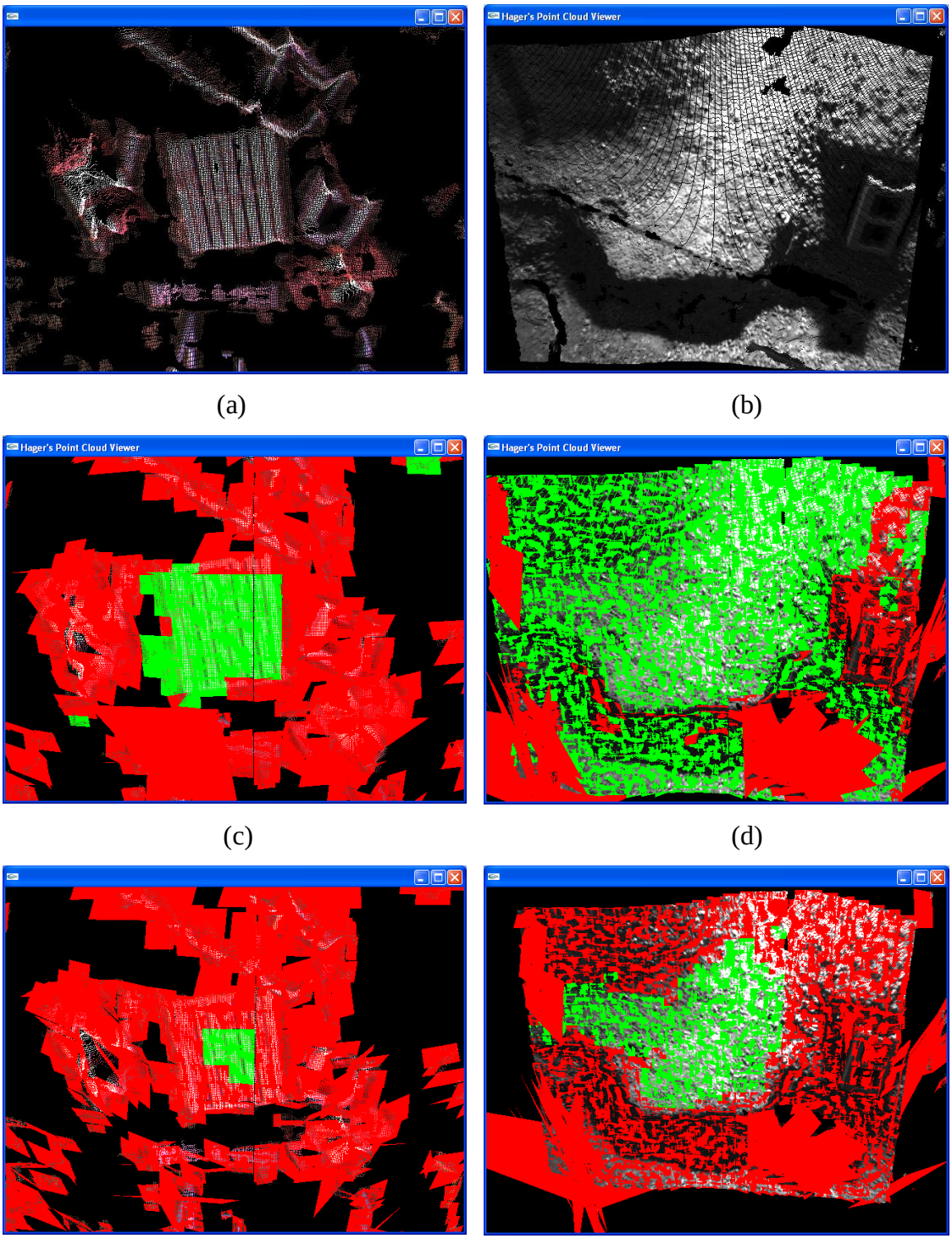
3.3 Finding Largest Traversable Region

To find a suitable site for deploying the ground robot, it is desirable to search terrain data for the largest region that is safe for continuous traversal by the robot. The contributions made in [6] are directly applicable to addressing this problem. The same method used in that approach is utilized here, with the point cloud segmented into 900 (30 rows x 30 columns) evenly distributed regions. The best-fit planes for these regions are applied to the point cloud using the multiple linear regression technique. These planes serve as the units for indicating traversable areas of the terrain. If a plane has an appropriately low angle from the z-axis (showing that is approximately horizontal), and the average slope of adjacent planes is below a specified threshold, this corresponds to a relatively low gradient slope. Such an area can thus be sufficiently defined as traversable by the ground robot.

Defining Acceptable Gradients

In [6], a constant value of 25 degrees was used as an acceptable slope threshold, where the slope is the angle from the plane's normal to the z-axis. This value works well for finding slopes of excessive gradient in that project. However, the point cloud data used by that author are relatively low in density (about 33% of points are correlated), whereas those gathered in the experiment described in the previous section are of relatively high density (about 80% of points are correlated). As a result, the constant slope threshold gives very different results between images from the utility cable experiment and those from [6]. Figure 3.17 illustrates the behavior of the acceptable planes when using threshold values of 25 and 6 on images from both these sets, where green-colored planes have acceptable slopes and red planes are unacceptable. While the threshold of 25 works well for the less dense point cloud, the threshold of 6 gives better results for the denser one. In the less dense point cloud, the presence of more gaps results in larger variations in slope, leading to a need for a higher acceptability threshold. In this case, a threshold value of 6 defines some traversable areas as unacceptable. For the denser point cloud, a more precise assessment of the terrain is provided, allowing for a lower acceptability threshold. For the example shown below, the threshold value of 25

causes most of the scene to be defined as acceptable, whereas a stricter test (one using a lower threshold value) gives a better indication of the dangerous areas in the scene. Therefore, an adaptive acceptability threshold is introduced that is computed based on the density of the point cloud, where density is defined as the number of points per best-fit plane. A set baseline threshold value is divided by the density of the point cloud under analysis to produce the acceptability threshold for best-fit planes within that image. Through experimenting with different baselines, a value of 2100 was found to give consistent results across the images tested. Figure 3.18 shows the results of using this adaptive acceptability threshold on the point clouds from 3.17.

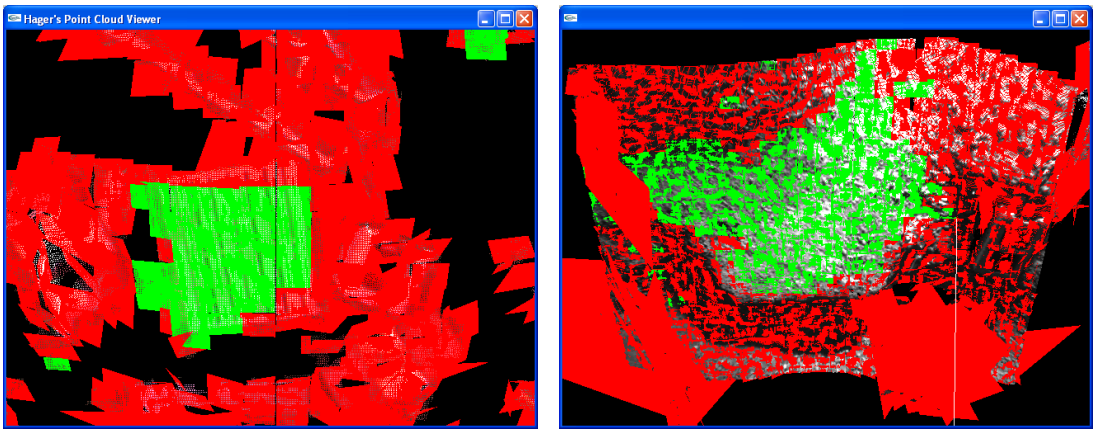


(a) (b)

(c) (d)

(e) (f)

Figure 3.17 Acceptability of planes for (a) point cloud from **[6]** and (b) point cloud from utility cable experiment using slope thresholds of (c & d) 25° and (e & f) 6°



(a) (b)

Figure 3.18 Acceptability of best-fit planes for point clouds from (a) **[6]** and (b) utility cable experiment using the adaptive acceptability threshold

Graph Theory Approach

Simply finding the set of independently traversable planes does not hold much practical value. Instead, the situational awareness system should identify the largest traversable region of the image. This involves finding the portion of the terrain where a ground robot can cover the most area without running into obstacles or encountering unsuitable slope gradients. The ability to cover the most ground is significant within the mission's context as the robot may only be capable of being lowered from the UAV once per trip. It is therefore important that the robot have the best chance of gathering the necessary samples from the ground. An early attempt at this solution was to simply find the largest number of connected acceptable planes; this involves an approach taken from graph theory. The set of acceptable best-fit planes applied to the point cloud is considered as the set of vertices for a graph G. In the first attempt at a solution, an edge between these vertices was said to exist if the two acceptable planes bordered one another in the image. An array of integers named "SubgraphSize" was created and initialized to the size of the number of planes in the image. Each entry i in the array corresponds to a subgraph H_i of G. Each subgraph is connected; this means for any two vertices u and v within the subgraph, there exists a path from u to v. Each of these subgraphs is also independent

from one another, meaning no edges exist between any vertices in two different subgraphs. Therefore, all paths from u to v must also consist exclusively of vertices within the subgraph to which u and v belong. Once the algorithm implemented here has completed, each entry i in the SubgraphSize array contains an integer value equal to the number of vertices in H_i.

The algorithm takes an iterative approach to populating the values of the SubgraphSize array. Each best-fit plane is stepped through, beginning with the top row and left-most column, and proceeding through each row until the end of the row is reached, at which time the process continues at the left-most plane on the second row. This continues until all planes have been stepped through. For each iteration of this process, the algorithm first checks if the current plane P is acceptable as defined above. If the plane P is not acceptable, the algorithm moves on to the next iteration. Otherwise, P is checked to see if it has already been identified as belonging to a particular subgraph H_i. This is accomplished by examining its subgraph ID, which has a value equal to i, where H_i is the subgraph to which P belongs. If the subgraph ID has a value of zero, this indicates that it has not yet been assigned to a particular subgraph. If this is the case, then it is defined as belonging to subgraph H_i, where i is the smallest value for which a subgraph H_i does not currently exist. At this point, the value of SubgraphSize[i] is incremented by 1.

The algorithm then checks whether an edge exists between the current plane P and the planes to the right of and below plane P. If the plane R to the right of the current plane P is acceptable, the subgraph ID of plane R is considered. If R does not currently belong to any subgraphs, its subgraph ID is changed to that of P. However, if R has a subgraph ID j where $j > 0$ and $j \neq i$, this indicates that subgraphs H_i and H_j are in fact connected to one another, meaning they belong to one subgraph. When this is the case, the value of SubgraphSize[j] is incremented by the value of SubgraphSize[i], and all planes with a subgraph ID of i have their subgraph ID values changed to j. This in effect merges all vertices from H_i into H_j. Once this has been completed, the plane D below P is considered. If D is acceptable, its subgraph ID is set to that of P. Because of the order in which the planes are stepped through, it is impossible for D to have already been assigned to a subgraph, making the merge check performed above unnecessary for D. Once this

process has been completed for all planes, the SubgraphSize entry with the highest value is defined to be the largest traversable region. Figure 3.19 shows an example of the results from this approach; the planes that are teal-colored are members of the largest traversable region found in the image from Figure 3.1.

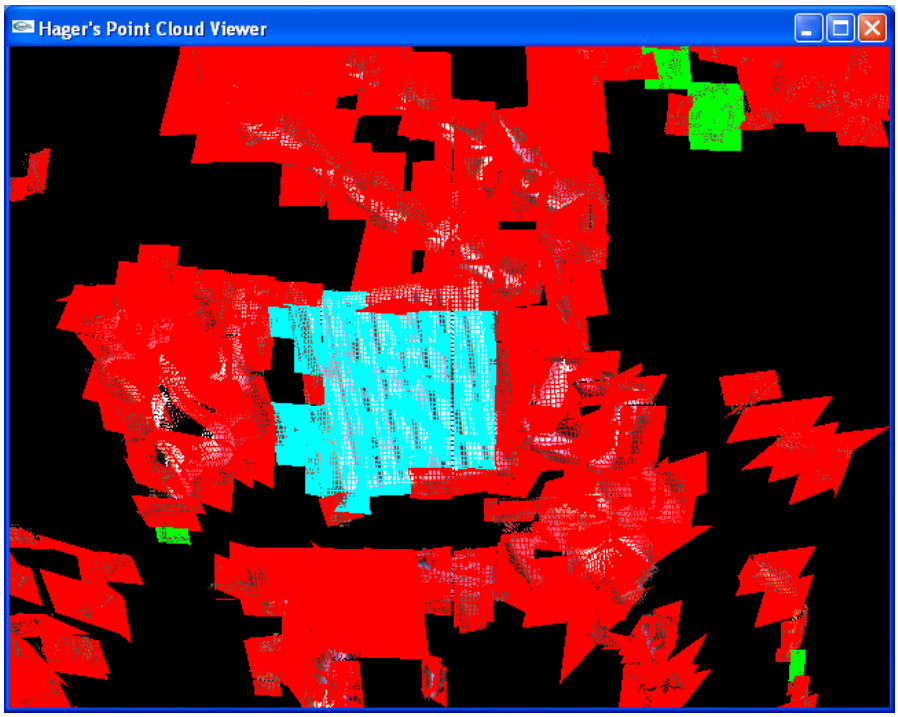


Figure 3.19 Largest traversable region output based solely on acceptability of planes

Z-Value Drop-off Between Planes

While this technique reliably locates the largest set of connected acceptable planes, this has some drawbacks when applied to ground robot operation. For example, consider the diagram in Figure 3.20. The angle of both planes' normals to the *z*-axis is similar, and both planes would have a sufficiently low slope gradient to be considered acceptable for robot operation. However, there exists a significant drop in *z*-value from the right-most border of the left plane to the left-most border of the right plane. Such a drop represents a potential hazard for the ground robot, and should be recognized as such by any algorithm attempting to identify traversable terrain.

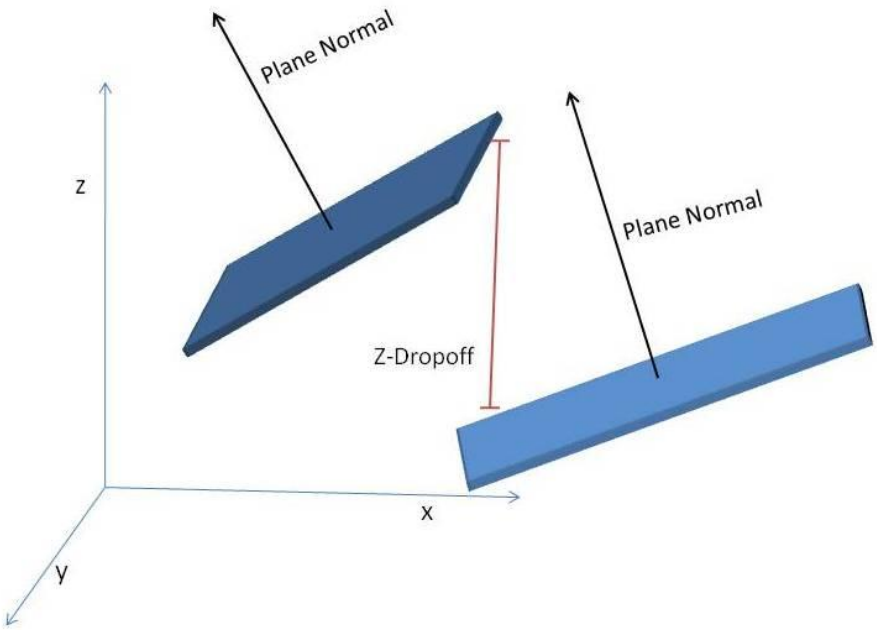


Figure 3.20 Example of *z*-value drop-off between two acceptable planes

To correctly identify these drop-off areas, a new definition of an edge in *G* is employed. An edge is now said to exist between two vertices in *G* if the two vertices are adjacent and the *z*-value drop-off between the two best-fit planes is below a pre-determined threshold. For this algorithm, the *z*-value drop-off is found using the borders of each best-fit plane. For comparing a current plane *P* to the plane *R* to the right of it, the average *z*-value of the right border for *P* is compared to the average *z*-value of the right border for *R*. Similarly, the average *z*-value of the bottom border for *P* is compared to that of the top border for *D* to find the *z*-value drop-off between those planes. If the drop-off is below the pre-defined threshold, and all other criteria discussed above are met, there exists an edge between the two vertices representative of those planes. Figure 3.21 shows the flow of this process. Using this technique produces the results shown in Figure 3.22; the mapping of the traversable region results to the original image is provided for clarification. One can see that there are green-colored acceptable planes clearly bordering on the teal-colored largest traversable region. These acceptable planes are not part of the connected subgraph due to a sharp drop-off in *z*-value between these regions and the larger region. In examining the original image, one can deduce that these drop-offs correspond to the edge of the wood palette in the center of the scene. Though the area on

the palette and the area immediately surrounding it are independently traversable, movement from one to the other would prove impossible or dangerous.

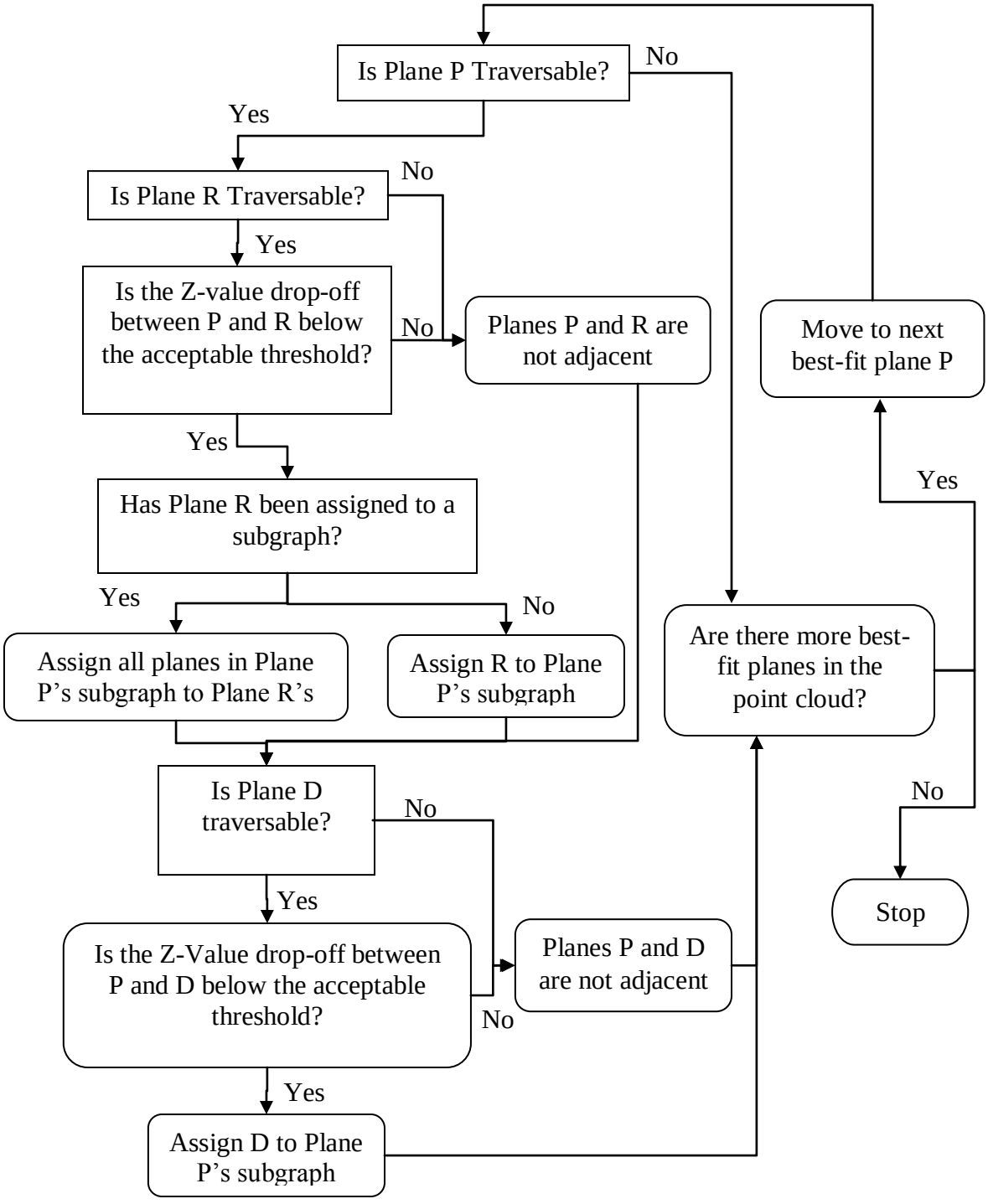


Figure 3.21 Flowchart for finding edges between vertices consisting of best-fit planes in the point cloud

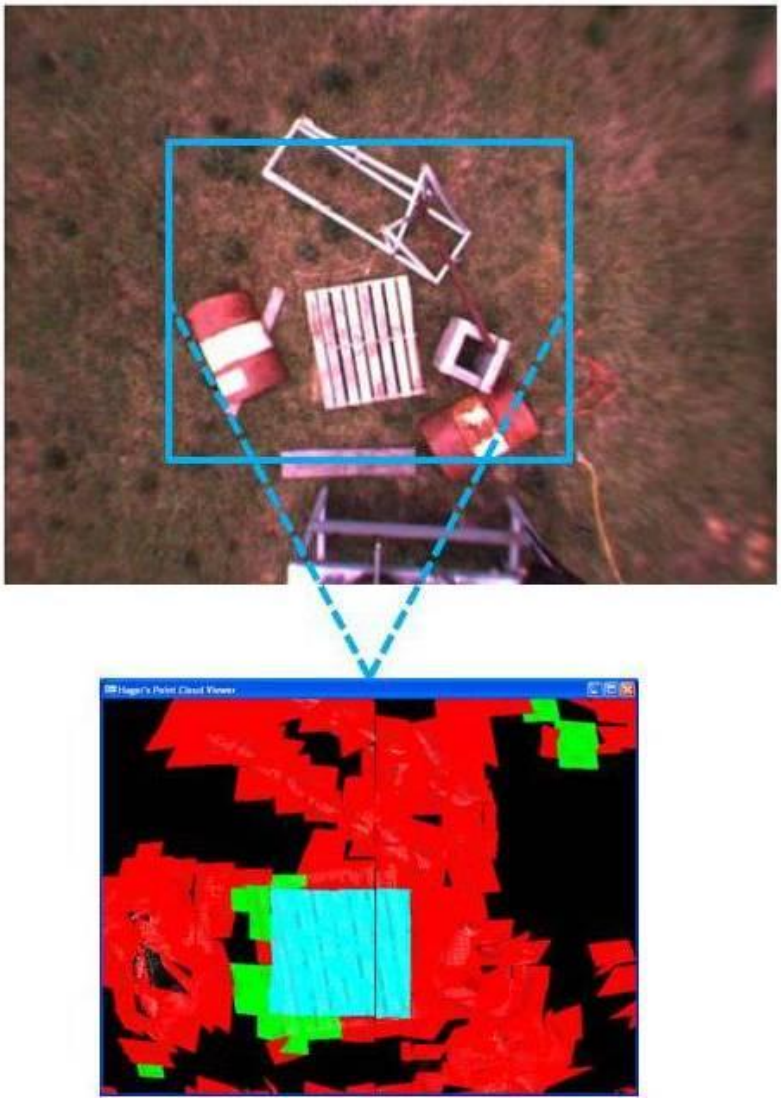


Figure 3.22 Largest traversable region output using *z*-value drop-off as mapped onto 2D image of scene

It was important to determine the best *z*-value drop-off threshold that would produce consistent results across several inputs. To accomplish this, several values were experimented with on separate point cloud data sets. It was found that a drop-off threshold of 0.04 works well on most inputs. It also is helpful to look at the angle difference of the normals of the best-fit planes to ensure they do not exceed some extreme case threshold; this angle difference threshold is set at 7.

One final modification that was made to the graph theory approach to finding the largest traversable regions was the definition of the size of a region. It was determined that basing a region size solely on the number of planes contained in that region was an

imperfect strategy. With the best-fit planes approach taken in [6], not all planes are representative of a same-size area. One plane may contain a high concentration of points that describe a relatively small real-world space, whereas another may contains a lower concentration of points describing a larger real-world area. For this reason, it is preferable to describe a region's size by the real-world area it encompasses. To accomplish this, the SubgraphSize array was changed to an array of float variables. Each time a plane *P* was added to a subgraph, the size of the corresponding SubgraphSize array entry was incremented by the value of the area of *P*, giving a more realistic reflection of the traversable region's size.

It should be noted that using the *z*-value drop-off approach for finding traversable regions makes the description of the region itself somewhat more complex. In the first implementation, two adjacent planes that lie within the same subgraph would inevitably have an edge lying between them. However, that is not true of the output from the new method. Consider the diagram in Figure 3.23, where planes 8, 9, 13, and 14 make up the largest traversable region. If there exists a drop-off between planes 9 and 14, this region is still fully connected; however, for the robot to travel from a point in plane 9 to one in plane 14, it must do so by traversing through planes 8 and 13. With this in mind, it is important for the operator to realize where potential hazards lie *within* fully connected, traversable regions. This issue will be addressed further in Chapter 5.

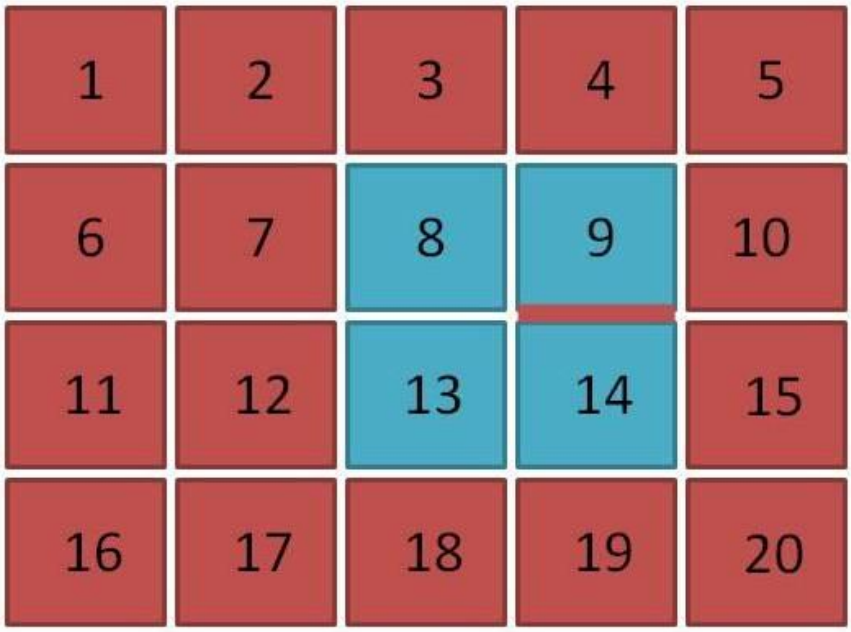


Figure 3.23 Illustration of traversable region containing a *z*-value drop-off

Chapter 4
Vision-based Tracking of a Ground Robot

In this chapter, the problem of tracking a ground robot through the use of visual information is considered. In contrast to the stereo visual data analyzed in Chapter 3, the tracking discussed in this chapter is done with the use of monocular (single-camera) vision. As discussed previously, a marker tracking system was chosen for this task, and the ARToolkit software was selected for its effectiveness at the task and its free availability. This chapter discusses the work done to use this software for determining the 3D distance from the camera to the marker, as well as its horizontal distance from the camera's optical axis. In addition, the results of some preliminary tests using a prototype camera system are presented, along with an analysis of the necessary camera and lens parameters needed for the final version of the system.

4.1 Calculating Distance and Position of Marker

In our experiments, the ARToolkit software typically did a fine job of locating markers within an image. The code used here for tracking the robot is modified from the "SimpleTest" example application provided with the ARToolkit package. In its original form, this application runs through a loop that grabs each frame from a video camera, searches it for the presence of the marker shown in Figure 4.1, and overlays that marker with a virtual 3D object. A screenshot taken from this application is shown in Figure 4.2. The blue cube in the image is the virtual object drawn by OpenGL, which is placed on the location where the marker was found. The software was tested at 15 frames per second; it consistently located the marker in each frame at this rate with no noticeable delay. Note that there are two markers in the image, but only one was specified as the object of interest, and thus the only one with the virtual object overlaid on it. The markers used for this example are approximately 0.2 meters on each side, though the ARToolkit software

is capable of detecting markers of various sizes; this will be explored further in Section 4.3.

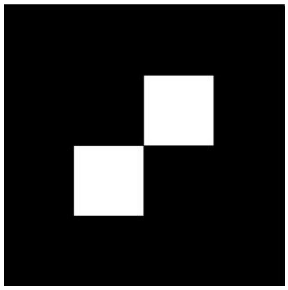


Figure 4.1 Marker which is searched for by ARToolkit in tracking software

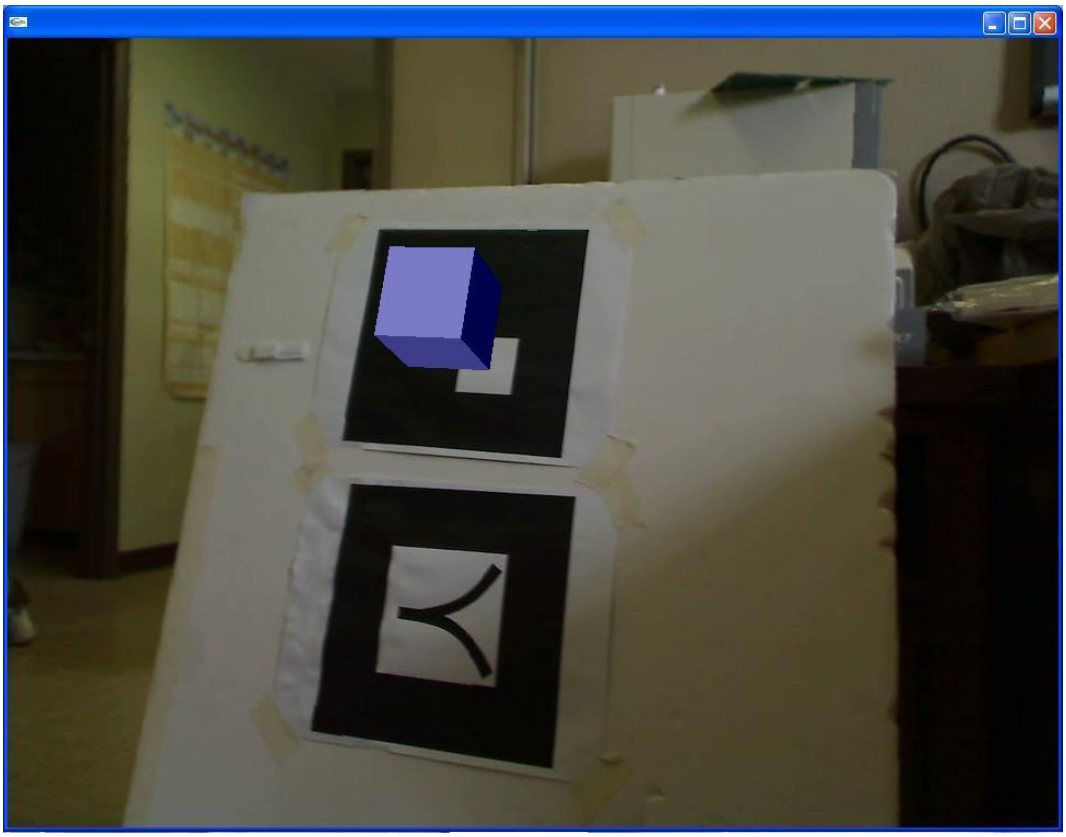


Figure 4.2 Screenshot from ARToolkit's SimpleTest application

For each iteration of the loop, there are two primary functions from the ARToolkit API that serve to locate and describe the markers in the frame. These functions are C-callable and are critical for use in applications utilizing ARToolkit's marker-tracking capability. The first of these to be called is arDetectMarker, which takes as input the

image being examined and a threshold value for converting the color image to binary. It then outputs the number of markers found in the image, and an array of ARMarkerInfo structures. These structures contain the following data about the markers found in the image:

Table 4.1. List of marker parameters defined in ARMarkerInfo structure **[24]**.

Type	Name	Description
Int	Area	Number of pixels in labeled region
Int	Id	Marker identifying number
Int	Dir	Direction of rotation about the marker
Double	Cf	Confidence value
Double	Pos[2]	Center of marker (in screen coordinates)
Double	Line[4][3]	Line equations for four sides of the marker
Double	Vertex[4][2]	Edge points of the marker

After the arDetectMarker function has been run, the resulting ARMarkerInfo array is passed to the arGetTransMat function. This function calculates the position and orientation of the camera relative to the tracking mark. Using the assumption that the marker is in the *x*-*y* plane with the *z*-axis pointing upward from the marker, it outputs the physical center, width, and transformation matrix for the marker [24]. In the SimpleTest example application, the marker position and orientation information is then passed on to a Draw function that overlays the virtual 3D object onto the position in the image where the marker is located.

The transformation matrix output from the arGetTransMat function is used by the ARToolkit software for positioning the virtual objects in the frame seen by the human user [25]. It serves to translate the coordinate system of the marker to that of the camera; these coordinate systems are shown in Figure 4.3. The camera coordinate system coincides with that used by OpenGL for displaying graphics in images, and the transformation matrix accurately positions 3D objects on the markers in the image. However, it does not take the real-world sizes of the markers or the field-of-view properties of the camera into account. Therefore, additional work is necessary to correlate

the position of the marker as found by the ARToolkit software with the marker's position relative to real-world distances.

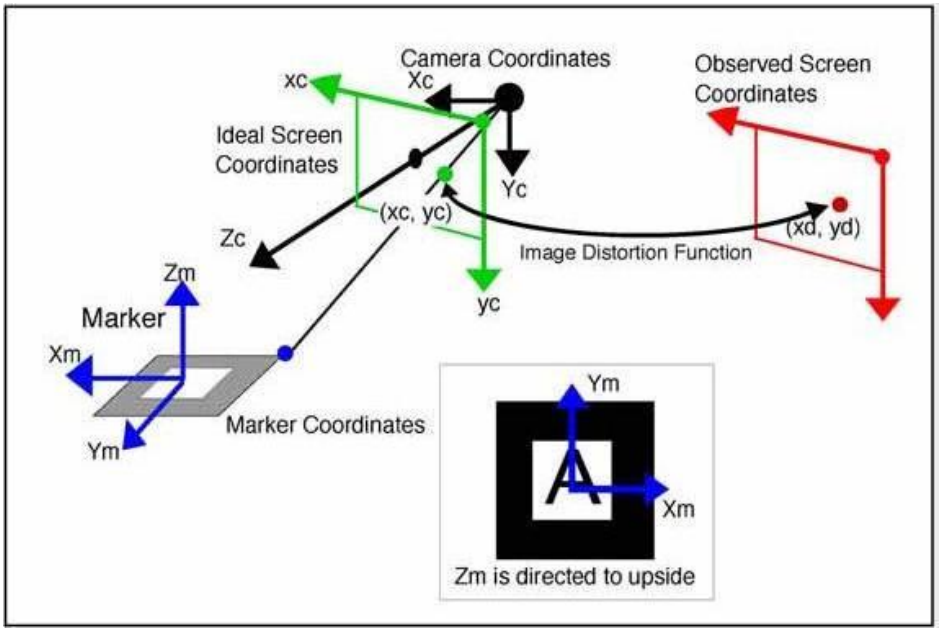


Figure 4.3 Diagram of ARToolkit coordinate systems

The tracking software developed by this author computes the distance and position of the ground robot by modifying the SimpleTest application. Given that the marker to be placed on the robot would be of a known size and pattern, it was decided that it should be possible to estimate the real-world distance to the robot from the data output by the arDetectMarker function. In fact, given a known marker size and the specifications for the camera being used, the distance to the marker can be estimated using just the Area parameter of the ARMarkerInfo structure.

In attempting to find these values, it is important to define several things about an image. Consider the diagram in Figure 4.4 that illustrates an example scene with the camera pointing at an area on the ground containing the ground robot.

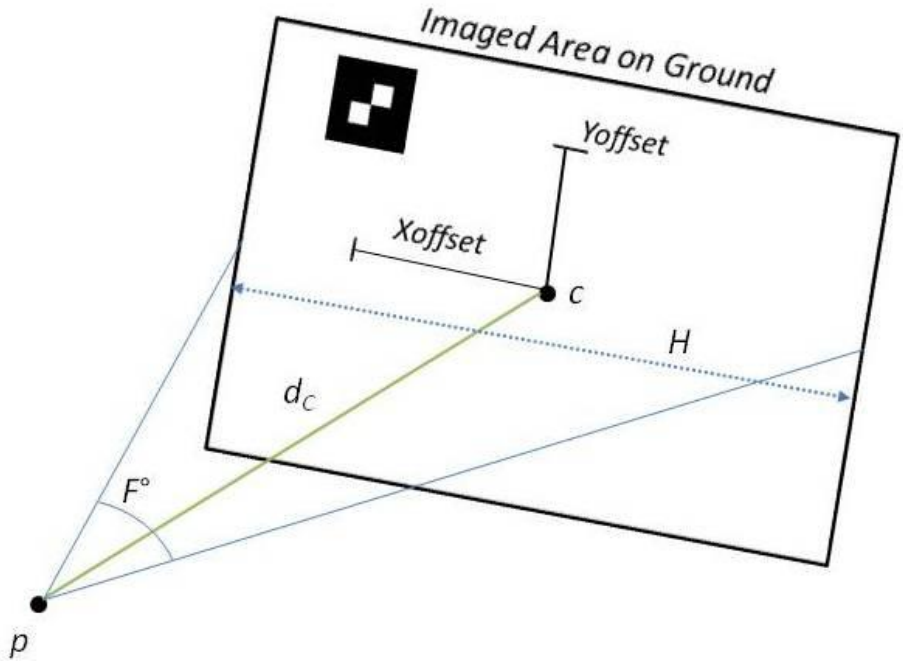


Figure 4.4 Diagram of field of view for camera looking at scene with ground robot marker

In the diagram, d_C is the real-world distance from the camera's position p to the point C on the ground, which appears at the center of the image, F is the horizontal field of view angle for the camera, and H represents the real-world width of the area on the ground that is viewable by the camera. The large rectangle in the figure represents the portion of the ground that the camera sees. The y_{offset} and x_{offset} values correspond to the distances along each axis from C to the center of the marker. The horizontal ground area H can be represented as follows:

$$H = 2 * \tan\left(\frac{F}{2}\right) * d_C \tag{7}$$

The 2D image area T can be represented by the equation below, where y is the number of vertical pixels in the image and x is the number of horizontal pixels:

$$T = \left(\frac{y}{x}\right) * H^2 \tag{8}$$

From (7) and (8) one can derive the following equation:

$$T = \left(\frac{y}{x}\right) * \left(2 * \tan\left(\frac{F}{2}\right) * d_C\right)^2 \tag{9}$$

The Area parameter of the ARMarkerInfo structure contains the number of pixels p_m in the image that make up the marker. Given this value, along with the total number of

pixels p_t contained in the image, one can compute the value of the 2D image area T viewable by the camera if the real-world marker size A is known. That is accomplished using the following equation:

$$T = A * \frac{p_t}{p_m} \tag{10}$$

By inserting the definition for the 2D image area given in (9), one can produce the following equation, which computes the height of the camera from the ground:

$$d_C = \frac{1}{2*\tan\left(\frac{F}{2}\right)} \sqrt{\frac{4}{3} * A * \frac{p_t}{p_m}} \tag{11}$$

When the marker is not centered in the image, the distance computation requires some additional steps. To find the distance to the marker in this case, the offset value must first be found. This offset is defined as the real-world distance from the center of the 2d image area to the marker location. The ARMarkerInfo structure's *pos* parameter provides the pixel coordinates of the center of the marker. Using these pixel coordinates, one can find the offset value if some correlation between pixels and real-world size is known. Using (10), the 2d Image Area can be ascertained; by inserting this value into (8), the Horizontal Area of the image is found. Knowing this value, along with the ratio of horizontal pixels to vertical pixels, one can determine the real-world width W_p and height H_p of pixels within the plane of the marker. From this, the following equations produce the offset values of the marker from the image center in the *x* and *y* directions, where x_{center} and y_{center} are the pixel coordinates of the image center:

$$X_{Offset} = W_p * (Pos[0] - X_{center}) \tag{12}$$

$$Y_{Offset} = H_p * (Pos[1] - Y_{center}) \tag{13}$$

These offset values are sent to the central processing unit on the helicopter to determine what flight commands are necessary to keep the helicopter centered above the ground robot. In addition, they allow the tracking software to compute the actual distance from the camera to the robot, even when the robot is not centered in the frame. Using the distance d_C to the center of the 2d image that lies within the plane of the marker, along with the distance equation as applied to the offset values, the following equation produces the value of the distance d_m from the camera to the marker:

$$d_m = \sqrt{d_C^2 + X_{Offset}^2 + Y_{Offset}^2} \quad (14)$$

4.2 Vision Tracking Test

An experiment using the modified ARToolkit software and a marker was set up to test the accuracy and consistency of the tracking system described above. To accomplish this, a rectangular grid was constructed with evenly distributed intersections, as shown in Figure 4.5 (the grid appears curved in this figure due to lens distortion). This grid contains 63 points (or intersections), where each point is almost exactly 0.1016 meters from each adjacent point. The center of the grid, marked by the red dot, is numbered as (0, 0), and all points on the grid are identified as (*x*, *y*), where *x* and *y* are the offset values on each axis from the center point. For example, the point immediately to the right of the center is referred to as (1, 0), the point immediately to the left is (-1, 0), the point above is (0, -1), and the point below the center is (0, 1).

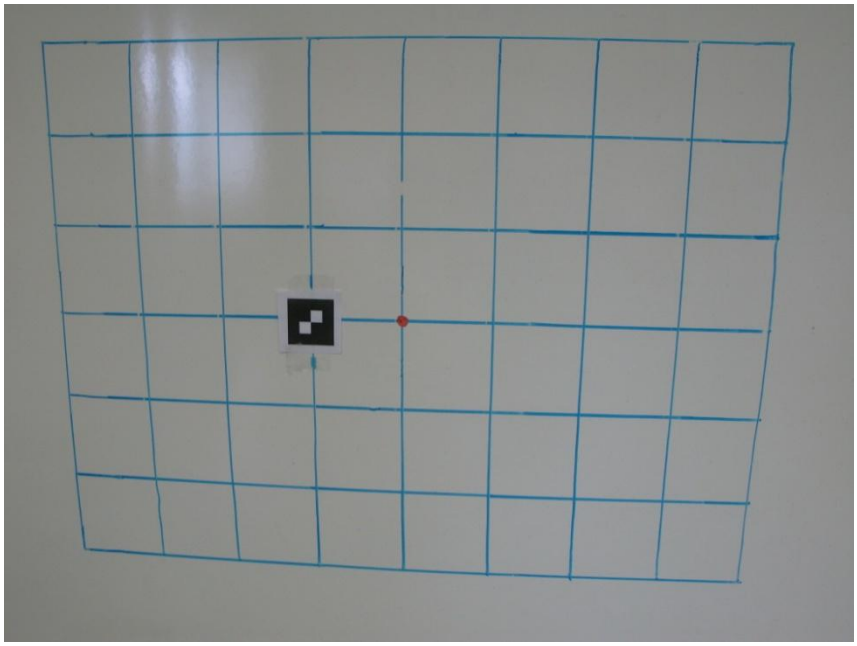


Figure 4.5 Photograph of grid used in tracking experiment

The camera used by the tracking system was then set up facing the grid. It was elevated to be level with the center of the grid, and positioned such that its focal plane would be close to parallel to the plane of the grid. However, some error in positioning inevitably exists, just as it would exist in a real scenario where a camera is mounted on a helicopter. The camera was positioned 81" (2.06 meters) from the center of the grid. A photograph of the camera setup used for the experiment is shown below (Figure 4.6). The software searches the image grid for the ARToolkit marker; a screenshot showing the grid as seen by the camera is given in Figure 4.7. Note that the marker in this frame has been overlaid by the 3D object to signify to the experimenter that the software has found the correct marker. Once the software locates the marker, it uses the equations derived in the previous section to determine the distance to the marker and the distances in both the x and y directions from the marker to the center of the frame. These values are displayed to the user; a screenshot of this display is given in Figure 4.8. This display shows how many markers were detected in the image, the number of pixels composing the marker of interest, the pixel coordinates of the marker's center, the real-world distance to the marker, the corresponding real-world sizes of the image pixels, and the values for x_{offset} and y_{offset}.

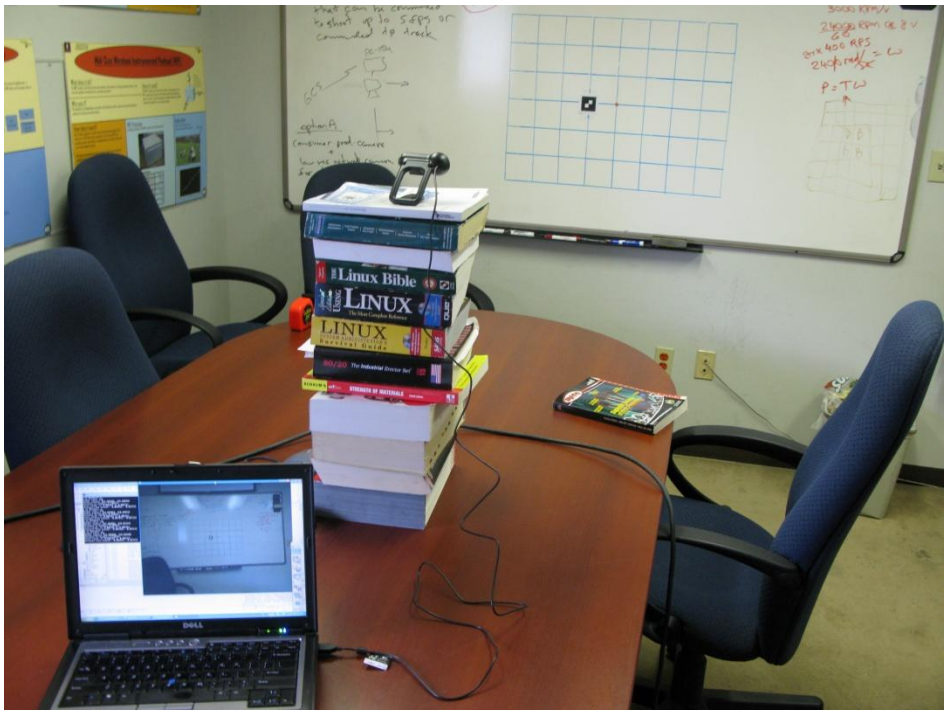


Figure 4.6 Camera station setup for tracking experiment

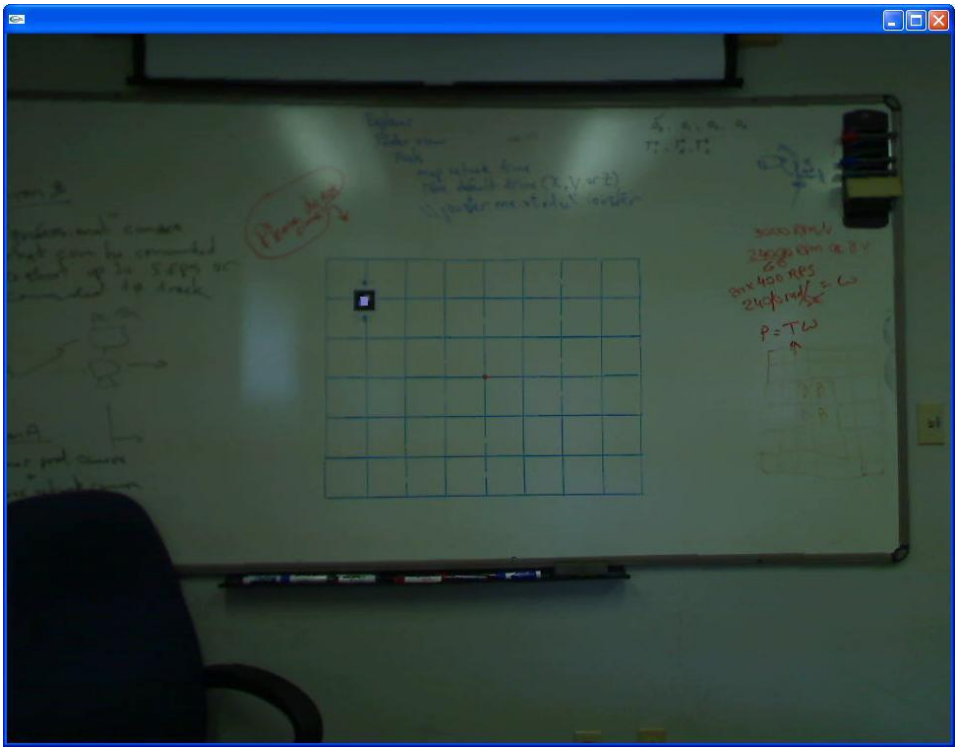


Figure 4.7 Screenshot of grid as seen by tracking camera

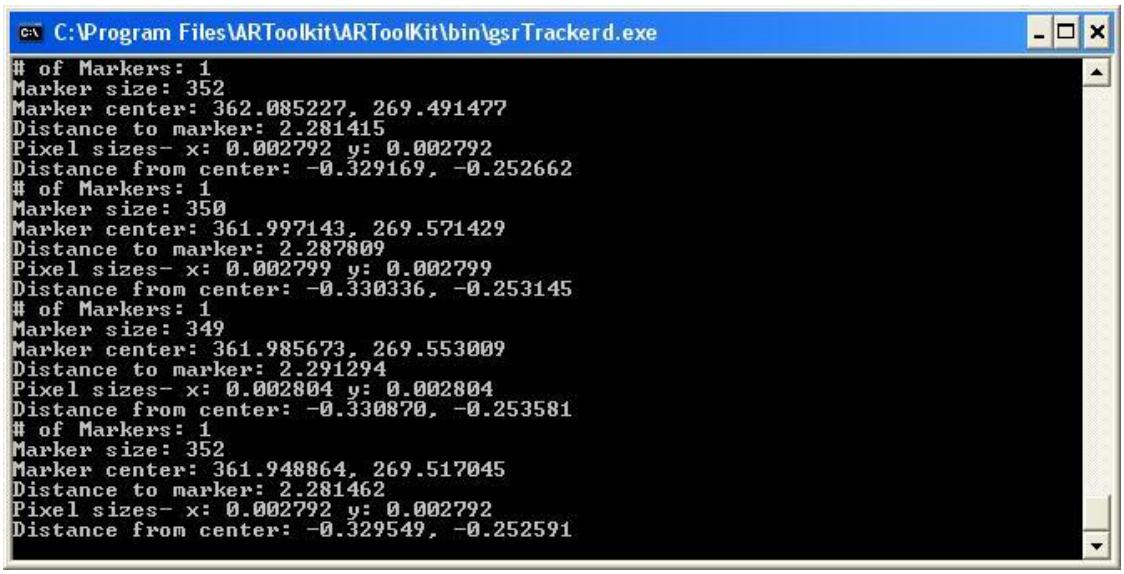


Figure 4.8 Screenshot of output given by tracking software

Using this numbering system, a small marker was placed at various points on the grid. The marker used here measures 0.0508 meters on each side, for an area of $0.002581 m^2$. At the distance used in the experiment, this would be roughly equivalent to tracking the marker that will be used on the actual mission at a distance of 50 meters when using a 2 megapixel camera with a 30° horizontal field of view. The method used for characterizing these scaling factors is discussed in the next section. The values computed by the tracking software for the offsets in the *x* and *y* directions at each of these positions were recorded. These computed positions were then compared to the actual real-world offset distances, and the corresponding errors were also recorded. The % error was calculated by dividing the absolute error by the total horizontal (in the case of x_{offset}) or vertical (in the case of y_{offset}) image area. The results of this test are shown in Table 4.2 below.

Table 4.2. Results of marker tracking experiment.

Position	X_Offset (m)				Y_Offset (m)			
(x,y)	Computed	Actual	Absolute Error	Error (%)	Computed	Actual	Absolute Error	Error (%)
(0,0)	0.019	0.019	0.000	0.0	-0.026	-0.026	0.000	0.0
(0,-1)	0.012	0.019	-0.007	0.3	-0.146	-0.128	0.018	1.1
(-1,0)	-0.099	-0.083	-0.016	0.7	-0.027	-0.026	0.001	0.1
(2,0)	0.239	0.222	0.017	0.7	-0.035	-0.026	0.009	0.5
(0,2)	0.016	0.019	-0.003	0.1	0.190	0.177	-0.013	0.8
(2,-2)	0.242	0.222	0.020	0.8	-0.258	-0.229	0.029	1.8
(-2,2)	-0.207	-0.184	-0.023	0.9	0.204	0.177	-0.027	1.6
(-3,-2)	-0.329	-0.286	-0.043	1.7	-0.253	-0.229	0.024	1.5
(3,2)	0.341	0.324	0.017	0.7	0.185	0.177	-0.008	0.5
(4,-3)	0.424	0.425	-0.001	0.0	-0.310	-0.331	-0.021	1.3
(-4,3)	-0.413	-0.425	0.012	0.5	0.298	0.279	-0.019	1.2

As these results show, the tracking algorithm manages to compute the marker's distance from the center with reasonable accuracy. All errors are less than 2%, with only 3 points producing errors larger than 1.5%. These errors are largely attributable to small differences in the angles between the camera's focal plane and the plane of the grid. In addition, the pixel count for the marker is often slightly smaller than the actual relative size of the marker to the rest of the scene; the reason for this discrepancy is unknown. However, some small errors are acceptable in the context of the purpose for this application.

Considering the likelihood that the helicopter's camera position will not provide a perfectly nadir view of the ground, it is important that the system be able to function despite the presence of some errors. This means that the central computer on-board the helicopter should not over-react to offset values provided by the tracker. When the robot is found to be significantly off-center from the helicopter, minor flight adjustments should be made. In addition, the use of geo-rectification will allow the system to compensate for helicopter movements. This can be done with IMU data from the helicopter, which provides pose information for the helicopter. By performing the

corresponding rotations on the position of the marker in the image, the system can ascertain the flight commands necessary to keep the helicopter in a hover over the ground robot. After each flight command is issued, the position of the robot is re-assessed, and further adjustments made if necessary. Given the relatively low speeds at which the ground robot will be traversing the terrain, and the stability afforded by autonomous flight, this approach is appropriate, making the tracker system presented here adequate for fulfilling the mission requirements. Though the software is capable of running at 15 frames per second without any delays, this speed may be unnecessary for the same reasons. Ultimately, it will be necessary to quantify at what speed the robot needs to be tracked once the speed at which it will be moving and the level of resistance of the helicopter to turbulence has been assessed.

4.3 Characterization of Camera Required for Tracking

Though the tracking software presented here is sufficient for tracking the ground robot gathering samples from radioactive terrain, this is only true if it is provided with adequate visual data from the camera on-board the UAV. In this chapter, some key parameters for the camera are examined, and a set of tests is run to simulate various values for these parameters. Based on the results of these tests, a characterization of the camera suited for this mission is presented.

The camera used for the results shown in this paper is a Logitech QuickCam Pro 9000. This camera was found to have a horizontal field of view of approximately 61.7°, and is capable of resolutions up to 960×720 when used with the ARToolkit software. These two parameters, field of view and resolution, are of paramount importance when considering the ability to locate a specific-sized marker from a certain distance, and are thus the focus of the tests described below. When attempting to increase the functional range of the marker detection system, higher resolution becomes preferable, and a narrower horizontal field of view is helpful as well.

The term resolution refers to the number of pixels used to represent a scene captured by a camera. Those with inadequate resolution cannot reliably resolve small objects in an image. If the marker in an image is not represented by enough pixels, it

becomes impossible for the ARToolkit software to locate it in an image. Consider two images, where image *A* contains half the number of pixels as image *B*. In this instance, a marker in image *B* is represented by the same number of pixels as a marker in image *A* that is twice its size in the real world. This property is used to simulate higher resolution images in the test below by scaling the defined real-world size of the marker being detected; the same camera was used for each test.

The horizontal field of view (HFOV) angle factors heavily into a camera's ability to locate far-away objects as well. In photography, zooming in on an object simply involves increasing the focal length of the lens, which in turn narrows the HFOV angle for the image. Figure 4.9 shows two images taken of the same scene with two different HFOV angles. The image on the left has a 75° HFOV angle, whereas the right image has a 47° HFOV angle. The test below simulates varying the HFOV angle by changing the pre-defined value for this parameter as appears in equation 11. The resulting values for 2D image area will no longer be valid when these values are altered, but the computed distance to the center of the area will be true for a camera with the new simulated HFOV, assuming the marker is in the center of the image.

Figure 4.9 Comparison of images taken of same scene with 75° (left) and 47° (right) HFOV angles

A test was constructed to find the edge of the range at which cameras with different resolutions and HFOV angles could detect a marker of a certain size. This marker size was set as 0.305 meters on each side, for a total area of 0.093 meters^2. This size comes from an estimate made by the project's ground robot operations expert on the maximum size for a marker that can be placed on the robot. Using this marker size, for each set of parameters the experimenter moved the camera further from the marker until

the software ceased locating it consistently. Consistent location of the marker is defined as it being found in more than 3 frames per second, as this was determined to be more than sufficient for choosing what flight commands must be issued to the helicopter. Two distances are given; static distance is defined as the maximum detectable distance when the camera is held steady, and vibration distance is the maximum detectable distance when the camera is being shaken similar to the vibrations likely to be experienced on the helicopter. The vibrations experienced by the camera were measured at over 3.5 g's. The larger distances were simulated using smaller markers. In addition, the experimenter made an effort to keep the marker as close to the center of the image as possible, thereby producing the best approximation of distance between the camera and marker. The results of this test are presented in Table 4.3.

Table 4.3. Results of camera characterization test. S.D. and V.D. stand for Static Distance and Vibration distance, respectively.

HFOV	**61.7°**		**45°**		**30°**	
Resolution	**S.D. (m)**	**V.D. (m)**	**S.D. (m)**	**V.D. (m)**	**S.D. (m)**	**V.D. (m)**
640×480	19	17	27	26	42	40
960×720	28	22	42	31	62	52
1600x1200	48	31	70	44	104	69
2300x1725	70	42	94	56	150	88

The cells highlighted in green illustrate those that achieved simulation results suitable for the mission constraints. Those in yellow are nearly sufficient, but lack the necessary vibration distance. It should be noted that the vibration results are highly dependent on the shutter speed of the camera. If the shutter speed is too low, the vibrations of the helicopter will distort the images, preventing the software from resolving the markers. With the ProCam software, the camera's exposure can be adjusted with a slide control. The exposure is inversely proportional to the shutter speed, so the exposure used in the tests was the lowest at which the marker could still be resolved with

the existing lighting conditions. The exposure used for the results gathered above was approximately 1/64 seconds.

From these test results, a basic characterization of the camera needed to satisfy the mission constraints is formed. Assuming a similar shutter speed to that of the camera used in the tests, one can choose a camera that carries a resolution and HFOV angle that falls under the acceptable results range from those in the table above. It is also important to select a camera with a large enough sensor to resolve imagery in the scene. If these criteria are met, the camera should be capable of locating the ground robot from a distance over 40 meters, while on-board a helicopter.

Chapter 5
Providing Operator Feedback

To safely control a ground vehicle from a remote location, it is critical to keep the human operator apprised of any important information. This chapter focuses on the human interface aspect of the situational awareness system developed for this mission. Though most of the development done for this feedback is not wholly original, it is presented here for its role in the presentation of data gathered using the methods discussed in the previous chapters.

5.1 Providing Feedback Concerning Utility Cable Detection

The first aspect of operator feedback addressed here deals with the utility cable detection technique. Each time an image is analyzed for utility cables, the areas of the image found to be likely utility cables are highlighted. In the initial implementation of this process, all such lines are colored red. However, it was decided that in cases where two different utility cables exist, each line should be colored differently. For this reason, after the technique developed in Section 3.2 is used to find all likely utility cables, the line with the most votes, along with any lines with approximately the same θ value as the most voted-for line, is colored red. All other likely utility cables in the image, which have a significantly different θ value, are colored purple. This allows the operator to quickly distinguish separate lines based on visual cues.

As mentioned in Section 3.2, it is necessary for the mission plan to include taking two stereo images of a given terrain scene. The assumption made here is that an image is taken, and then the helicopter is rotated 45° counter-clockwise, and a second stereo image is taken. If this is done correctly, the θ value for any utility cables in the first image will be increased by 45° for the second image. Therefore, if any lines were hidden from detection in the first data set due to a θ value approaching 0°, these lines should be detected in the second image. However, there is also a possibility that a utility cable that

was detected in the first image could have its θ value approach 0° in the second image, potentially hiding it from detection in the second case. Assuming the human operator will base a suitability decision on the results from the second image, the system should provide a warning if a utility cable from the first image has "disappeared" in the second.

To accomplish this task, a PowerLine structure is created that holds the parameter values for two lines in an image: the most voted-for line (referred to as pl_1), and the most voted-for line that has a θ value significantly different from pl_1 (this line is referred to as pl_2). When a new image is being analyzed, the PowerLine structure will contain parameter values for pl_1 and pl_2 as they were found for the previous image. The utility cable detection routine then calls the function ImageAnalysis, which compares the utility cables found in the current image with the lines pl_1 and pl_2 that were found in the previous image. If the most voted-for lines pl_1 and pl_2 in the current image are sufficiently close to the estimated values of pl_1 or pl_2 from the previous image after a 45° rotation, they are presumed to be the same utility cables. However, if a line in the current image does not lie sufficiently close to the estimated values for the old pl_1 or pl_2 after rotation, this indicates that the corresponding utility cable from the first image is hidden in the second image. If this is the case, a warning message appears to the operator, warning him that a utility cable is likely hidden in the current view. Figure 5.1 shows an example of this occurrence. The first image, shown on the left, contains a utility cable with $\theta \approx 135°$. The second image contains the same utility cable, but the θ value has been rotated 45° to 180° (which is equivalent to 0°). Because the utility cable was not detected in the second image, a warning appears alerting the operator to this fact. By using this form of feedback, the system allows an operator to base decisions for suitability of a region on more informed data. The inherent weakness of the stereo vision system is overcome when this feedback technique is used in conjunction with proper mission planning.

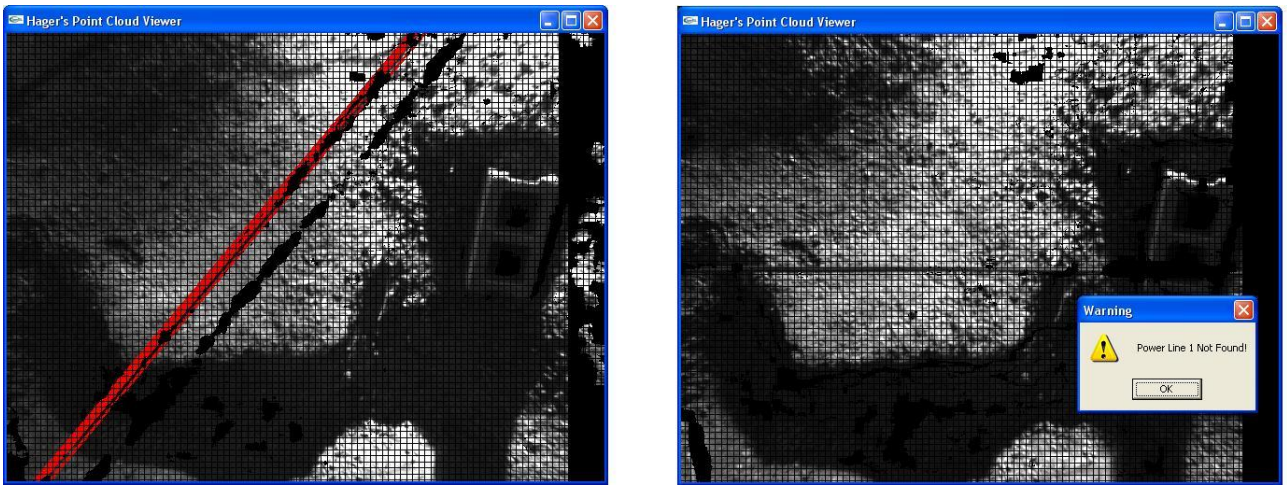


Figure 5.1 Demonstration of two image utility cable detection with a warning for an undetected utility cable

5.2 Traversable Region Feedback

Another aspect of locating suitable deployment sites is the largest traversable region location detailed in Section 3.3. As discussed previously, the acceptable best-fit planes are colored green and those which are unacceptable are colored red. In addition, the planes found to belong to the largest traversable region are marked with a teal color. However, as mentioned in Section 3.3, simply marking those planes that constitute the largest traversable region is insufficient. Doing so fails to alert the operator of *z*-value drop-offs *within* the traversable region.

To provide such critical information, a set of four Boolean values are added to the Plane structure: R_safe, D_safe, L_safe, and U_safe. These correspond to acceptable *z*-value drop-offs between the plane and its neighbor on the right, below, to the left, and above, respectively. When the planes are drawn by the system, these Boolean values are then checked for any sharp drop-offs. If a plane is found to have such a drop-off, a narrow red quadrilateral is drawn at the corresponding border. Figure 5.2 shows a scene that has been drawn with these drop-off indicators.

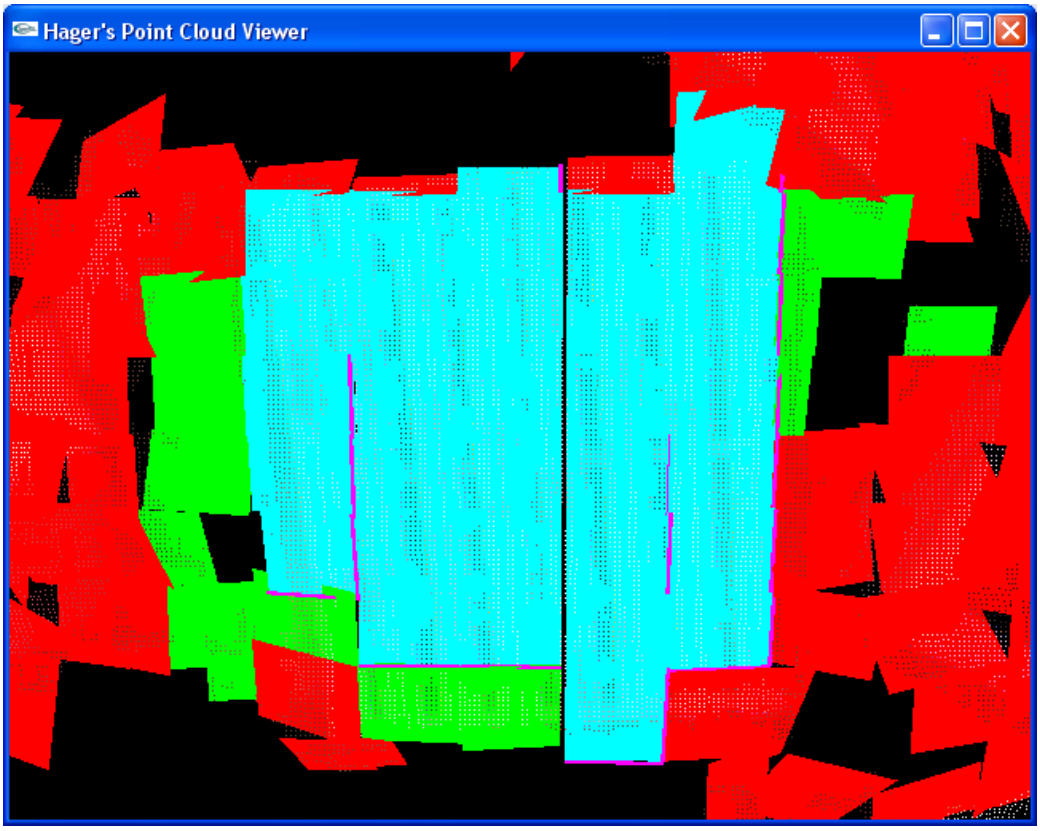


Figure 5.2 Traversable region display with drop-offs within the region marked

By marking these drop-offs within the region, the system can give the operator sufficient information regarding the suitability of a potential deployment site for the ground robot. In addition, real-time feedback can be given after the robot has been deployed if the current location for the robot can be mapped onto the corresponding area of the point cloud. If this is done, one can be alerted regarding any potential unsafe maneuvers from the robot's current position. To simulate this scenario, the mouse is used to select an initial deployment position for the robot. The user can then maneuver the simulated robot around the terrain using the keyboard's arrows. Each press of an arrow moves the robot to the next adjacent plane in the corresponding direction, and the plane on which the robot currently lies is marked with a blue color. Each time such a move is made, the software then analyzes the current position of the robot for any immediately hazardous moves. The operator is notified if any such dangerous maneuvers are imminent. A screenshot depicting an example of this notification is presented in Figure 5.3.

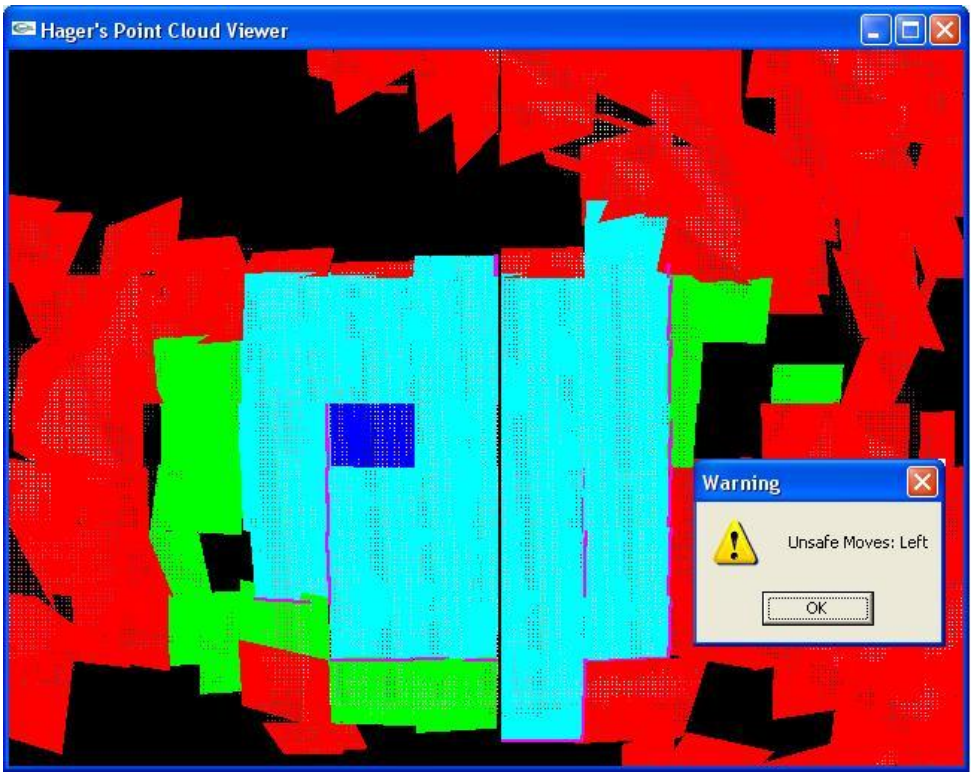


Figure 5.3 Traversable region display with drop-off warning to user

5.3 Vision-based Tracking Feedback

In addition to simulating a robot's location through keyboard presses, a prototype has been developed that combines the vision-based tracking procedure discussed in Chapter 4 with the traversable region analysis presented in Chapter 3. When this tracking interface feature is enabled, the user selects a log file location upon starting the point cloud viewing application. This log file corresponds to one that is written to by the visual tracking application. Upon analyzing each video frame, the tracking system updates the log file to contain the location of the ground robot. The point cloud viewer application continually checks this file for the robot's location. It then finds the best-fit plane that contains the point at which the robot is located and colors it blue.

The full operation of this system requires both applications to be running simultaneously. While the human operator is observing the 3D traversable region feedback, the tracking system will be locating the position of the marker in real-time. Meanwhile, the human operator is able to see where in the 3D plane set the robot is currently located as this

information is updated automatically. This operation is based on the assumption that the coordinate system for the 3D range data corresponds to that of the image used for tracking. While this may not be true in the final implementation of the UAV image systems, it serves to demonstrate the usability of this situational awareness system.

5.4 Instructions for Use

Using the situational awareness is designed to be simple for human operators. This section gives a brief overview of the setup and use of the system. The first step is the installation of OpenGL and GLUT (Graphics Library User Toolkit). These are installed by placing the corresponding DLL files into the Windows System folder, and by placing the library files in the "include" directory for the compiler being used. If the user wishes to use the vision-based tracking system simultaneously with the point cloud viewer, the config.h file should be modified such that the "USE_TRACKER" value is 1, and the project should be re-compiled. Otherwise, the user may simulate marker movements using the keyboard by setting this configuration value to 0 before compiling the project.

If the vision-based tracking is to be used, a USB or Firewire camera should be connected to the computer, and the "gsrtracker" executable is run. Once this execution has begun, the point cloud viewer application should be run. The user may then select the log file being written to by the tracking application, and then select a point cloud file to open. If the ground robot movements are instead being simulated by keyboard input, the user does not select a log file.

Once a point cloud file has been selected, pressing the 'P' key will highlight any detected utility cables as red or purple. If a new point cloud is opened after one has had the utility cable detection run on it, the system will check for continuity between the utility cables in the two images. This is done based on an assumed 45° rotation, as discussed in the previous section. To view the traversable region, the user toggles the best-fit plane display with the 'B' key and presses the 'F5' key. The largest traversable region is displayed with a teal color, and all potentially dangerous drop-offs are indicated with the use of red lines. If the vision-based tracking system is running concurrently, the current location of the marker in the scene is indicated with a blue-colored plane. If the

user is simulating marker movement, an initial deployment point is chosen by clicking the left mouse button, and the position is changed through the use of keyboard arrow presses. Table 5.1 gives a complete overview of keyboard commands with the point cloud viewer.

Table 5.1. Keyboard input for Point cloud viewer; modified from **[6]**

Keyboard Input	**Effect**
2	Displays 2D image of scene
3	Displays 3D point cloud
5	Overlays 3D points onto 2D image of scene
B	Toggles display of best-fit planes when in 3D mode
I	Open a point cloud file
C	Captures a scene from Bumblebee camera
O	Outputs the scene to a PPM file
M	Selects planes in 2D mode that are defined as acceptable
T	Toggles sticky selection
P	Runs utility cable detection on point cloud
←→	Translates points in the *x* direction; simulates robot movement in the *x* direction if displaying traversable region
↑↓	Translates points in the y direction; simulates robot movement in the *y* direction if displaying traversable region
Z	Translates points in the negative *z* direction
X	Translates points in the positive *z* direction
D	Affects the yaw of the scene in the positive direction
A	Affects the yaw of the scene in the negative direction
S	Affects the pitch of the scene in the positive direction
W	Affects the pitch of the scene in the negative direction
E	Affects the roll of the scene in the negative direction
Q	Affects the roll of the scene in the positive direction
0	Resets the view to the default
F1	Colors best-fit planes according to slope
F2	Colors best-fit planes according to average angular difference of the surrounding planes
F3	Colors best-fit planes according to acceptability
F4	Colors best-fit planes according to kurtosis
F5	Colors best-fit planes according to traversability
Left mouse button	Click and drag to affect pitch and yaw of scene; click to choose initial deployment point for robot if displaying traversable region
Left mouse button + Shift	Click and drag to affect pitch and roll of scene
Right mouse button	Click and drag to translate scene
Right mouse button + Shift	Click and drag to slowly translate scene

These methods of providing operator feedback form the basis of a safe ground robot deployment mission. By notifying human operators of the presence of utility cable obstacles and the location of the largest traversable terrain regions, the system enables the operator to make the best choice for a deployment site for the robot. In addition, by providing real-time feedback on the robot's proximity to potential hazards, the safe operation of the robot is facilitated.

Chapter 6
Conclusion

This thesis has detailed the research and development of a situational awareness system for safe operation of a ground sampling robot that will be deployed from an unmanned helicopter. In this chapter, the findings of this work are discussed, and suggestions for future work in this field are made. Though several separate problems are addressed in this thesis, they combine to form the basis for safely deploying and operating a ground robot from a UAV, even with the human operator at a remote location.

6.1 Summary of Findings

The utility cable detection algorithm presents a method for detecting the presence of these dangerous obstacles. Using the parameter values found in Section 3.2, the detection system found nearly all cables that were draped across a sample terrain scene in an experiment. Although some basic limitations of stereo ranging prevent the camera system from locating utility cables that appear horizontal relative to the image, a method for mission planning has been laid out that overcomes this weakness.

Additionally, the system finds the traversable regions in a set of point cloud data in current experiments. The regions are defined to be traversable based on the slope of the best-fit planes found using a multiple linear regression technique, and the possibility of movement between planes is dependent on elevation drop-off between them. Using graph theory principles, a method for determining the largest traversable region of the terrain is developed as well.

One aspect of the point cloud analysis that will eventually become critical is the speed at which it is performed. Although this thesis was intended to demonstrate the concepts of using these techniques to produce useful information about the terrain, it did not focus on the speed of implementation. However, the execution times for critical

aspects of the system are presented in Table 6.1 to show where bottlenecks exist. The times shown were those gathered using a Pentium M 2GHz processor with 2GB of RAM. The "Open" function represents reading in a point cloud file from the hard drive. The time to generate a point cloud using the Point Grey software system is less than 1 second. At present, the algorithms discussed – which have not been optimized for speed – are capable of reading in and analyzing a point cloud file at a rate of 3 -5 per minute. Though this is significantly slower than the rate at which the ground robot's location is tracked, this speed may be sufficient for the mission. This then relies on a proper method for translating the current position of the robot tracking camera to the position from which the point cloud image was gathered. This performance is also dependent on the processing platform that is used on the helicopter; at present, a Mini PC form factor is being considered with processing power similar to that used to produce the results below.

Table 6.1. Execution Time for Point Analysis Tasks.

Point Cloud File		Task Execution Time (s)				
Image ID	Size (KB)	Open	Regression w/ Kurtosis	Regression w/o Kurtosis	Traversability Analysis	Cable Detection
30_H1	11,842	8.625	11.781	4.093	0.000	0.266
90_H1	11,872	8.734	11.688	4.140	0.000	0.469
Cross_45_1	11,710	8.625	11.500	4.078	0.000	0.469
Cross_90_1	11,420	8.422	11.219	4.094	0.000	0.984
2.4 (from [6])	5,152	3.516	2.937	1.453	0.000	0.485

The vision-based tracking explored in Chapter 4 locates a ground robot through the use of monocular vision and a marker placed on top of the robot. Based on the experimental results that were presented in Table 4.2, the tracking system should prove suitable for the mission constraints. If the camera parameters fall within the requirements laid out in Table 4.3, it should be capable of locating the target marker on top of the robot. However, once a camera is selected, it should obviously be tested rigorously to ensure that this is the case. One key aspect to keep in mind is the shutter speed at which it is operating. Though a fast shutter speed is preferable to overcome some of the helicopter's vibration issues, the aperture for the camera must be capable of gathering

enough light to see the objects in the image with a lower exposure time. For this reason, the camera should be tested for vibrations at various lighting conditions.

6.2 Suggestions for Future Work

Though this thesis presents a good baseline for a situational awareness system for operating a ground robot deployed from a UAV, there is room for future work to be done on this topic. With regard to the utility cable detection, the system is not designed to overcome some of the limitations of stereo vision when more than two separate utility cables are present in the image. Therefore, it would useful to research some ways to improve the system for cases where several different utility cables exist. Some additional statistical methods for locating non-traversable areas of the terrain could also be examined, as well as further testing of the existing method on some additional terrain examples.

With regard to the stereo system used to acquire the data used for the tasks discussed here, some improvements could be considered as well. This includes provisions for operating in various lighting conditions, stabilization of the camera on a vibrating helicopter, and determination of the accuracy of the camera when used with lenses of different focal lengths at various distances from the target area. In addition, it would be beneficial to explore alternative means for gathering point cloud data, specifically with the use of LIDAR technology. Though the weight of these systems may prove prohibitive for use on an unmanned helicopter, some may work well within the mission constraints. The situational awareness system developed for this thesis should be highly portable to use with a LIDAR system as well, as the output from such systems is very similar to that of stereo vision cameras.

The vision-based tracking system can be further researched and improved upon as well. One such area for improvement is that of occlusion. Currently, the tracking system will not locate the target if it is partially blocked from sight. It may prove necessary to conduct research on some techniques for overcoming this issue. In addition, though the tracking algorithm currently operates at a sufficient rate, some portions of the code may need to be optimized for execution time if a significant slow-down is observed with

higher resolution cameras. At present, the tracking system's distance computations do not assume that the marker is not skewed with respect to the camera; if such skew exists, some error will result. Though this error may not be prohibitive for issuing proper flight commands, this can be eliminated by taking the pose of the robot into account when performing the calculations, which would require using the output of the arGetTransMat function.

Though the operator feedback currently provided is sufficient for demonstrating the use of the situational awareness system, some aspects could still be improved on. A more user-friendly interface that includes dynamic toolbars for accessing features would help facilitate easier use of the system. In addition, a system that correlates the location of the robot from the tracking portion to a specific area of the point cloud may prove useful. This would allow the operator to be assured of the safety of the robot through constant updates provided by the system. To properly implement this, some research needs to be performed on methods for mapping real-time tracking output to a point cloud data set that is only periodically updated. This may be done with the use of GPS or IMU (inertial measurement unit) sensors on the helicopter, though the precision of these may not be suitable for this task. Alternatively, odometer information from the ground robot could be used to track its current location on the terrain.

Some time-related issues may need to be addressed in future research as well. Though the execution times presented in Table 6.1 may be satisfactory for the mission, some speed-up may be possible by optimizing the multiple linear regression code or the file interface code. Additionally, an efficient method for transmitting the data from the UAV to the ground station is needed. Though radios such as the 900MHz Digi are capable of data transmission up to 115.2 Kbps, this speed is not sufficient for transmitting an entire point cloud to the ground station in a timely manner. At this rate, it would take about 13 minutes to send a single point cloud file to the ground station, assuming a resolution and density typical of the experiment conducted in Chapter 3. Although the COFDM video radio is capable of much faster data rates (over 2048Kbps), it would still take over 45 seconds to transmit such a point cloud file. For this reason, it will be imperative that the point cloud analysis takes place on the UAV itself, and then a smaller file containing a summary of results be sent to the ground station.

By implementing some of these suggestions for future work, a more robust and easily usable situational awareness system can be developed. However, the methods discussed in this thesis constitute some significant contributions to the safe deployment and operation of a ground robot from a UAV. This in turn lays the groundwork for interesting additions to this area of work in the future, some of which are already being researched by other students in the Unmanned Systems Laboratory.

References

[1] "What is OpenCV?" *OpenCVWiki.* [Online] 2006. [Cited: April 10, 2009.] http://opencv.willowgarage.com/wiki/#Welcome.2BAC8-Introduction.WhatisOpenCV.3F.

[2] Khronos Group. *OpenGL.* [Online] 1997. [Cited: April 10, 2009.] http://www.opengl.org/.

[3] HIT Lab NZ. "How Does ARToolkit Work?" *ARToolkit.* [Online] [Cited: March 12, 2009.] http://www.hitl.washington.edu/artoolkit/documentation/userarwork.htm.

[4] Ayache, N. and Hansen, C. "Rectification of Images for Binocular and Trinocular Stereovision." In *Proceedings of the 9th International Conference on Pattern Recognition.* (Rome, Italy, Nov. 14-17, 1988). 11-16.

[5] Sharkasi, A. Stereo Vision Based Aerial Mapping Using GPS and Inertial Sensors. M.S. Thesis, Dept. of Mechanical Engineering, Virginia Tech. 2008.

[6] Klomparens, D. Automated Landing Site Evaluation for Semi-Autonomous Unmanned Aerial Vehicles. M.S. Thesis, Bradley Dept. of Electrical and Computer Engineering, Virginia Tech. 2008.

[7] Point Grey Research. Triclops Stereo Vision System Manual. 2003.

[8] Del-Cerro, J., Barrientos, A., Campoy, P, and Garcia, P. "An Autonomous Helicopter Guided by Computer Vision for Inspection of Overhead Power Cable." In *Proceedings of the 2002 IEEE/RSJ Conference on Intellgent Robots and Systems* . (Sept. 30 - Oct. 5, 2002). 69-78.

[9] Yang, G., and Gillies, D. "Hough Transform." Lecture Notes on Computer Vision, Dept. of Computing, Imperial College. London, UK. 2004.

[10] Li, Z., Liu, Y., Hayward, R. Zhang, J., and Cai, J. "Knowlege-based Power Line Detection for UAV Surveillance and Inspection Systems." In *23rd International Conference on Image and Vision Computing New Zealand.* (Christchurch, New Zealand, Nov. 26-28, 2008). IEEE, 1-6.

[11] Golightly, D. and Jones, D. "Visual control of an unmanned aerial vehicle for power line inspection." In *Proceedings of the 12th International Conference on Advanced Robotics.* (Seattle, WA, July 18-20, 2005). 288-295.

[12] Vosselman, G., and Dijkman, S. "3D Building Model Reconstruction From Point Clouds and Ground Plans." In *Proceedings of the International Archives of Photogrammetry and Remote Sensing.* (Annapolis, MD, 2001). Vol. XXXIV-3, No. W4, 22-24.

[13] Katsoulas, D., and Bergen, L. "Efficient 3D Vertex Detection in Range Images Acquired with a Laser Sensor." In *Proceedings of the 23rd DAGM Symposium on Pattern Recognition* . (Munich, Germany, Sept. 12-14, 2001). Springer Berlin, Frelburg, Germany, Vol. 2191/2001, 116-123.

[14] Zhou, M., Xia, B., Su, G., Tang, L., and Li, C. "Study on the Target Feature Extraction from Lidar Point Clouds." In *Proceedings of the International Archives of the Photogrammetry, Remote Sensing, and Spatial Information Sciences.* Vol.

XXXVII, B3b. 309-312.

[15] Pfeifer, Norbert. "Powerline Mapping." Presentation at International School on Lidar Technology (IIT Kanpur, India). Vienna University of Technology, Austria. 2008.

[16] Yilmaz, A., Javed, O., and Shah, M. "Object Tracking: A Survey." *ACM Computing Surveys.* 2006. ACM Press, New York, NY, Vol. 38, 4. Article 13.

[17] Comport, A., Marchand, E., and Chaumette, F. "A Real-time Tracker for Markerless Augmented Reality." In *Proceedings of the 2nd IEEE and ACM International Symposium on Mixed and Augmented Reality.* (Oct. 7-10, 2003). IEEE Computer Society, Washington, D.C., 36-45.

[18] Tomasi, C. and Kanade, T. "Shape and Motion From Image Streams Under Orthography: a Factorization Method." *International Journal of Computer Vision.* 1992. Kluwer Academic Publishers, The Netherlands, Vol. 9, 2, 137-154.

[19] Fiala, M. "ARTag, a Fiducial Marker System Using Digital Techniques." In *Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition* . (San Diego, CA, June 20-25, 2005). IEEE Computer Society, Washington, D.C., 590-596.

[20] Sementille, A., Lourenco, L., Brega, J., and Rodello, I. "A Motion Capture System Using Passive Markers." In *Proceedings of the 2004 ACM SiGGRAPH International Conference on Virtual Reality Continuum and Its Applications in Industry* . (Singapore, June 16-18, 2004). ACM Publications, New York, NY, 440-447.

[21] Kato, H. and Billinghurst, M. "Marker Tracking and HMD Calibration for a Video-Based Augmented Reality Conferencing System." In *Proceedings of the 2nd IEEE and ACM International Workshop on Augmented Reality* . (San Francisco, CA, Oct. 20-21, 1999). IEEE Computer Society, Washington, D.C., 85-94.

[22] Kanbara, M., Yokoya, N., and Takemura, H. "A Stereo Vision-Based Augmented Reality System with Marker and Natural Feature Tracking." In *Proceedings of the 7th International Conference on Virtual Systems and Multimedia* . (Berkeley, CA, Oct. 25-27, 2001). IEEE Computer Society, Washington, D.C., 455-462.

[23] Fisher, R., Perkins, S., Walker, A., and Wolfart, E. "Sobel Edge Detector." *Image Processing Learning Resources.* [Online] 2003. [Cited: March 15, 2009.] http://homepages.inf.ed.ac.uk/rbf/HIPR2/sobel.htm.

[24] HIT Lab NZ. ARToolkit API Documentation. *ARToolkit.* [Online] 2006. [Cited: April 7, 2009.] http://artoolkit.sourceforge.net/apidoc/index.html.

[25] HIT Lab NZ. "Coordinate Systems." *ARToolkit.* [Online] [Cited: April 18, 2009.] http://www.hitl.washington.edu/artoolkit/documentation/cs.htm.

[26] Rusdorf, S. and Brunnett, G. "Real Time Tracking of High Speed Movements in the Context of a Table Tennis Application." In *Proceedings of the ACM Symposium on Virtual Reality Software and Technology.* (Monterey, CA, Nov. 7-9, 2005). ACM Publications, New York, NY, 192-200.

Appendix A
Source Code

This appendix contains source code used for each aspect of the situational awareness system detailed in this thesis. This system currently encompasses two different solutions: the deployment site assessment discussed in Chapter 3, and the vision-based tracking application presented in Chapter 4. The source code files developed by this author for each of these solutions are given in the following pages. The deployment site decision code that analyzes the point cloud uses the files in Table A.1.

Table A.1. Description of files used in deployment site suitability analysis

Cloud.h	Header file for reading and analyzing point clouds
Cloud.cpp	Contains functions for reading in and performing analysis on point clouds
Render.h	Header file for drawing visual elements
Render.cpp	Contains functions for drawing planes, points, and other scene elements
Driver.cpp	Driver file for program; handles user input and the flow of the program
Config.h	Configuration header file
Bumblebee.h	Header file for interfacing with the Bumblebee camera
Bumblebee.cpp	Functions that allow use of the Bumblebee camera

Each of these files is included below. All were originally written by Dylan Klomparens and modified by this author for use with the situational awareness system, except for the Driver.cpp file which was developed by Jeff Molofee and modified by Klomparens and this author. The file gsrtracker.c, which is used for the marker tracking discussed in

Chapter 4, is included as well. As discussed earlier, this file builds on the SimpleTest application that was provided by ARTookit. This author has modified the code to compute distances to the marker and provide corresponding feedback to the user.

Cloud.h

```
#pragma once

#include <iostream>
#include <fstream>
#include <sstream>
#include <cmath>
#include <cassert>
#include <list>
#include <limits>
#ifdef WIN32
#include <windows.h>
#endif
#include <GL/gl.h>
#include <GL/glu.h>
#include "config.h"
using namespace std;

struct Point
{
	Point();
	float x, y, z, r, g, b; // Represents x, y, z coords in the real world, along with red, green, and
blue color.
	int i, j, d; // i is the pixel row. j is the pixel column. d is the pixel disparity.
	long ref;   // Reference to position in Point array
	float h, grade;   // Hough transform output
	bool on_edge;
};

struct Vector
{
	Vector();
	float x, y, z;
};

struct HitList
{
public:
	bool Contains(pair<int, int> Item);
```

Cloud.h

```
	void Toggle(pair<int, int> Item);
	void Add(pair<int, int> Item);
	void Remove(pair<int, int> Item);
	void Clear();
private:
	list<pair<int, int>> Hits;
};

struct Plane
{
	Plane();
	Point P1, P2, P3, P4; // The 4 corners of the plane.
	Vector Normal; // The normal vector of the plane.
	float AngleFromZ; // The angle between the normal of the plane and the z-axis, in degrees.
	float AverageAdjacent; // The largest angular difference between the plane and all of the adjacent
planes.
	float Kurtosis; // A measure of how well the points fit the plane.
	int Name; // The "name" (or identifier) of the plane, used for selection in rendering.
	bool Hit;
	int Subgraph; // The identifier of the connected subgraph to which the plane belongs
	bool Marked;
	bool R_safe, D_safe, L_safe, U_safe;
};

struct PPMImage
{
	PPMImage();
	PPMImage(const PPMImage& x);
	PPMImage& operator=(const PPMImage& x);
	void Clear();
	~PPMImage();
	bool DrawBox(long X1, long Y1, long X2, long Y2);
	long Height;
	long Width;
	long MaxColor;
	time_t Timestamp;
	GLubyte* Data;
};
```

Cloud.h

```
struct Texture
{
	Texture();
	Texture(const Texture& x);
	Texture& operator=(const Texture& x);
	void Compose(PPMImage& Image);
	~Texture();
	GLuint ID;
	GLubyte* Data;
	int Width;
	int Height;
};

struct Cloud
{
	Cloud();
	Cloud(const Cloud& x);
	Cloud& operator=(const Cloud& x);
	bool Open(string FileName);
	bool Save();
	void ComposeGrid(int R, int C);
	void MultipleLinearRegression(Plane*& P, int PixelLowerX, int PixelUpperX, int PixelLowerY, int
PixelUpperY);
	void ProximityEvaluation();
//	void CalculateQuality(float ImportanceOfSteepness, float ImportanceOfProximitySimilarity);
	float CompareNormals(int OriginRow, int OriginColumn, int TargetRow, int TargetColumn, float& Count);
	void RangeCount();
	void LineDetect();
	void Clear();
	void ZeroParameters();
	void DeleteBestFitPlanes();
	~Cloud();

	long PointCount;
	bool Has2D;
	Point* Points;
	PPMImage Image;
```

Cloud.h

```
	HitList Hits;
	Plane*** BestFitPlanes;
	int Columns; // The total number of columns that the point cloud is divided into for analysis.
	int Rows; // The total number of rows that the point cloud is divided into for analysis.
	float DeltaX;
	float DeltaY;
	Texture Tex;
	int LargestSubgraph;
	int curr_x;
	int curr_y;
};

struct PowerLine
{
	PowerLine();
	int r_1, t_1, r_2, t_2;
	bool pl_1_found, pl_2_found;
	void ImageCompare(int most_t, int most_t_alt, unsigned int ntheta);
};

Vector& operator/=(Vector& v, const float Scalar);
float len(const Vector& v);
float AngularDifference(const Vector& u, const Vector& v);
ostream& operator<<(ostream& Out, const Point& P);

// Contains information about utility cables detected in the previous
//    image for comparison with the current image
extern PowerLine pline;
```

Cloud.cpp

```
#include "cloud.h"

const float PI = 4.0f * atan(1.0f);
PowerLine pline;

PowerLine::PowerLine() : r_1(-1), t_1(-1), r_2(-1), t_2(-1), pl_1_found(false), pl_2_found(false)
{

}

Point::Point() : x(0), y(0), z(0), r(0), g(0), b(0), i(0), j(0), d(0), ref(-1), h(0), grade(0),
on_edge(false)
{

}

Vector::Vector() : x(0), y(0), z(0)
{

}

bool HitList::Contains(pair<int, int> Item)
{
	for(list<pair<int, int>>::const_iterator i = Hits.begin(); i != Hits.end(); i++)
		if(Item == *i)
			return true;
	return false;
}

void HitList::Toggle(pair<int, int> Item)
{
	Contains(Item) ? Remove(Item) : Add(Item);
}

void HitList::Add(pair<int, int> Item)
{
	if(Contains(Item)) return;
	Hits.push_back(Item);
```

Cloud.cpp

```
}

void HitList::Remove(pair<int, int> Item)
{
	for(list<pair<int, int>>::iterator i = Hits.begin(); i != Hits.end(); i++)
		if(Item == *i)
		{
			Hits.erase(i);
			return;
		}
}

void HitList::Clear()
{
	Hits.clear();
}

Texture::Texture() : ID(0), Data(0), Width(0), Height(0)
{

}

Texture::Texture(const Texture& x)
{
	Data = 0;
	*this = x;
}

Texture& Texture::operator=(const Texture& x)
{
	if(Data)
		delete [] Data;
	ID = x.ID;
	Width = x.Width;
	Height = x.Height;
	if(x.Data)
	{
		long ByteCount = x.Width * x.Height * 3 * sizeof(GLubyte);
```

Cloud.cpp

```
			Data = new GLubyte[ByteCount];
			memcpy(Data, x.Data, ByteCount);
		}
		return *this;
}

void Texture::Compose(PPMImage& Image)
{
		Height = Image.Height;
		Width = Image.Width;
		if(Data)
			delete [] Data;
		long ByteCount = Width * Height * 3 * sizeof(GLubyte);
		Data = new GLubyte[ByteCount];
		memcpy(Data, Image.Data, ByteCount);
}

Texture::~Texture()
{
		if(Data)
			delete [] Data;
}

Plane::Plane() : AngleFromZ(0), Kurtosis(0), AverageAdjacent(0), Name(0), Hit(false), Subgraph(0),
Marked(false), R_safe(false), D_safe(false), L_safe(false), U_safe(false)
{

}

PPMImage::PPMImage()
{
		Data = 0;
		Clear();
}

PPMImage::PPMImage(const PPMImage& x)
{
		Data = 0;
```

```
	*this = x;
}

PPMImage& PPMImage::operator=(const PPMImage& x)
{
	if(&x == this) // Check for self assignment.
		return *this;
	if(Data) // Delete original data.
		delete [] Data;
	Height = x.Height; // Deep copy the new data.
	Width = x.Width;
	MaxColor = x.MaxColor;
	Timestamp = x.Timestamp;
	if(x.Height * x.Width > 0 && x.Data)
	{
		int Limit = x.Height * x.Width * 3;
		Data = new GLubyte[Limit];
		memcpy(Data, x.Data, Limit * sizeof(GLubyte));
	}
	return *this;
}

// Draws a black box at the coordinates given.
// Returns false if the box is not within the image boundries or no image data exists.
bool PPMImage::DrawBox(long X1, long Y1, long X2, long Y2)
{
	if(Data == 0 || X1 < 0 || Y1 < 0 || X2 < 0 || Y2 < 0 || X1 >= Width || X2 >= Width || Y1 >= Height ||
Y2 >= Height)
	{
		assert(sizeof("Preconditions for PPMImage::DrawBox not met.") == 0);
		return false;
	}
	long TopLeft = 3 * (Width * Y1 + X1);
	long TopRight = 3 * (Width * Y1 + X1 + (X2 - X1));
	long BottomLeft = 3 * (Width * Y2 + X1);
	long BottomRight = 3 * (Width * Y2 + X1 + (X2 - X1));
	long RowStep = 3 * Width;
	for(long p = TopLeft; p <= TopRight; p++)
```

```
		Data[p] = 0;
	for(long p = BottomLeft; p <= BottomRight; p++)
		Data[p] = 0;
	for(long p = TopLeft; p <= BottomLeft; p += RowStep)
	{
		Data[p] = 0;
		Data[p + 1] = 0;
		Data[p + 2] = 0;
	}
	for(long p = TopRight; p <= BottomRight; p += RowStep)
	{
		Data[p] = 0;
		Data[p + 1] = 0;
		Data[p + 2] = 0;
	}
	return true;
}

void PPMImage::Clear()
{
	Height = 0;
	Width = 0;
	MaxColor = 0;
	Timestamp = 0;
	if(Data)
		delete [] Data;
	Data = 0;
}

PPMImage::~PPMImage()
{
	if(Data)
		delete [] Data;
}


// This function compares the power line parameters from the current image with those
// in the last image. It searches for each power line from the first image to exist in
```

Cloud.cpp

```
// the second image with an increase in theta by 45 degrees
void PowerLine::ImageCompare(int most_t, int most_t_alt, unsigned int ntheta)
{
	int t_expected = -1;
	int t_alt_expected = -1;
	int old_t_diff;
	int old_t_alt_diff;
	bool old_t_found = false;
	bool old_t_alt_found = false;
	const int t_diff_thresh = 5;

	pl_1_found = false;
	pl_2_found = false;

	// Check if there were power lines detected in a previous image
	if(t_1 != -1)
		t_expected = (t_1 + 10) % ntheta;
	if(t_2 != -1)
		t_alt_expected = (t_2 + 10) % ntheta;


	old_t_diff = abs(most_t - t_expected);
	if(old_t_diff > (ntheta / 2))
	{
		if(most_t < (ntheta / 2))
			old_t_diff = (ntheta - t_expected) + most_t;
		else
			old_t_diff = (ntheta - most_t) + t_expected;
	}

	// Check if pl_1 from first image corresponds to pl_1 in the second image
	if((old_t_diff < t_diff_thresh || abs((int)(old_t_diff - ntheta/2)) < t_diff_thresh)
		&& most_t > -1 && t_expected > -1)
		pl_1_found = true;

	old_t_diff = abs(most_t_alt - t_expected);
	if(old_t_diff > (ntheta / 2))
	{
```

Cloud.cpp

```
		if(most_t_alt < (ntheta / 2))
			old_t_diff = (ntheta - t_expected) + most_t_alt;
		else
			old_t_diff = (ntheta - most_t_alt) + t_expected;
	}

	// Check if pl_1 from first image corresponds to pl_2 in the second image
	if((old_t_diff < t_diff_thresh || abs((int)(old_t_diff - ntheta/2)) < t_diff_thresh) && most_t_alt >
-1
		&& t_expected > -1)
		pl_1_found = true;

	old_t_alt_diff = abs(most_t - t_alt_expected);
	if(old_t_alt_diff > (ntheta / 2))
	{
		if(most_t < (ntheta / 2))
			old_t_alt_diff = (ntheta - t_alt_expected) + most_t;
		else
			old_t_alt_diff = (ntheta - most_t) + t_alt_expected;
	}

	// Check if pl_2 from first image corresponds to pl_1 in the second image
	if((old_t_alt_diff < t_diff_thresh || abs((int)(old_t_alt_diff - ntheta/2)) < t_diff_thresh)
		&& most_t > -1 && t_alt_expected > -1)
		pl_2_found = true;

	old_t_alt_diff = abs(most_t_alt - t_alt_expected);
	if(old_t_alt_diff > (ntheta / 2))
	{
		if(most_t_alt < (ntheta / 2))
			old_t_alt_diff = (ntheta - t_alt_expected) + most_t_alt;
		else
			old_t_alt_diff = (ntheta - most_t_alt) + t_alt_expected;
	}

	// Check if pl_2 from first image corresponds to pl_2 in the second image
	if((old_t_alt_diff < t_diff_thresh || abs((int)(old_t_alt_diff - ntheta/2)) < t_diff_thresh)
		&& most_t_alt > -1 && t_alt_expected > -1)
```

```
			pl_2_found = true;
}

Cloud::Cloud()
{
	ZeroParameters();
}

Cloud::Cloud(const Cloud& x)
{
	ZeroParameters();
	*this = x;
}

Cloud& Cloud::operator=(const Cloud& x)
{
	// Check for self assignment
	if(&x == this)
		return *this;
	// Delete pre-existing memory
	Clear();
	// Copy new memory
	PointCount = x.PointCount;
	Has2D = x.Has2D;
	LargestSubgraph = x.LargestSubgraph;
	curr_x = x.curr_x;
	curr_y = x.curr_y;
	old_x = x.old_x;
	old_y = x.old_y;
	DeltaX = x.DeltaX;
	DeltaY = x.DeltaY;
	Columns = x.Columns;
	Rows = x.Rows;
	Image = x.Image;
	Hits = x.Hits;
	Tex = x.Tex;
	//LogFileName = x.LogFileName;
```

Cloud.cpp

```
	if(PointCount > 0) // Check that data exists to be copied.
		Points = new Point[PointCount];
	for(int i = 0; i < PointCount; i++)
		Points[i] = x.Points[i];
	if(x.BestFitPlanes)
	{
		BestFitPlanes = new Plane**[Rows];
		for(int i = 0; i < Rows; i++)
		{
			BestFitPlanes[i] = new Plane*[Columns];
			for(int j = 0; j < Columns; j++)
				BestFitPlanes[i][j] = x.BestFitPlanes[i][j] ? new Plane(*x.BestFitPlanes[i][j]) :
0;
		}
	}
	return *this;
}

// This function opens a portable pixel map file. The xyz
// point information is embedded as comments within the image.
// The format of a PPM goes as follows:
// 1st line: "P6"
// 2nd line: "# time = <timestamp>"
// 3rd line: "# point count = <point count>"
// Next <point count> lines: "# <point>"
// After that: "<width> <height>" of the image
// After that: "<max color value>" which is the range of color
// values from 0 to <max color value>. This is usually 255.
// Subsequent data are triples of RGB values.
bool Cloud::Open(string FileName)
{
	Clear();
	int q;
	ifstream f(FileName.c_str(), ios_base::binary);
	if(f.fail())
	{
		f.close();
		return false;
```

Cloud.cpp

```
	}
	string Buffer;
	getline(f, Buffer); // Read in "P6"
	if(Buffer == "P6")	// If image does not have P6 header, it does not have a corresponding 2D
image
		Has2D = true;
	else
		Has2D = false;

	f >> Buffer >> Image.Timestamp; // Read in "#time= <timestamp>"
	f >> Buffer >> PointCount; // Read in "#pointcount= <point count>"
	Points = new Point[PointCount];
	for(long c = 0; c < PointCount; c++) // Read in each point, with the format "# r g b x y z i j d"
	{
		Point& p = Points[c];
		f >> Buffer >> p.x >> p.y >> p.z >> p.r >> p.g >> p.b >> p.i >> p.j;
		if(Has2D)
			f >> p.d;
	}
	f >> Image.Width >> Image.Height; // Read in the image height and width
	f >> Image.MaxColor; // Read in the maximum value of the color range for the image data
	f.get(); // Skip over the new line character (0x0A)

	// The "times 3" for the ByteCount is here because the color information
	// is represented as RGB. One value per color, per height, per width.
	long ByteCount = Image.Width * Image.Height * 3 * int(sizeof(GLubyte));
	Image.Data = new GLubyte[ByteCount];
	//f.read((char*)(Image.Data), ByteCount);
	long b;
	for(b = 0; b < ByteCount; b++)
		f.read((char*)(&Image.Data[b]), 1);
	assert(b == ByteCount);
	f.close();

	if(PointCount == 0)
		return true;

	Tex.Compose(Image);
```

```
	glGenTextures(1, &Tex.ID);
	glBindTexture(GL_TEXTURE_2D, Tex.ID);
	glTexImage2D(GL_TEXTURE_2D, 0, 3, Tex.Width, Tex.Height, 0, GL_RGB, GL_UNSIGNED_BYTE, Tex.Data);
	glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
	glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
	return true;
}

// Writes a PPM to the hard drive. Will work on both Linux and Windows with consistant newline output.
bool Cloud::Save()
{
	if(Image.Data == 0)
		return false;

	stringstream ss;
	ss << Image.Timestamp;
	string FileName;
	ss >> FileName;
	FileName.append(".ppm");

	const char NewLine = char(0x0A);
	ofstream f(FileName.c_str(), ios_base::binary);
	if(f.fail())
	{
		f.close();
		return false;
	}
	f << "P6" << NewLine;
	f << "#time= " << Image.Timestamp << NewLine;
	f << "#pointcount= " << PointCount << NewLine;
	for(long c = 0; c < PointCount; c++)
		f << "# " << Points[c] << NewLine;
	f << Image.Width << ' ' << Image.Height << NewLine;
	f << Image.MaxColor << NewLine;
	f.write((char*)Image.Data, Image.Height * Image.Width * 3 * sizeof(GLubyte));
	f.close();
	return true;
}
```

Cloud.cpp

```
void Cloud::ComposeGrid(int R, int C)
{
	int r, c;
	if(R <= 0 || C <= 0) return;
	Rows = R;
	Columns = C;
	// Calculate the size of each square segment.
	DeltaX = (float)Image.Width / (float)Columns;
	DeltaY = (float)Image.Height / (float)Rows;
	DeleteBestFitPlanes();
	BestFitPlanes = new Plane**[Rows];
	for(r = 0; r < Rows; r++)
	{
		BestFitPlanes[r] = new Plane*[Columns];
		for(c = 0; c < Columns; c++)
			BestFitPlanes[r][c] = 0;
	}
	int LowerX, UpperX, LowerY, UpperY;
	for(r = 0; r < Rows; r++)
	{
		for(c = 0; c < Columns; c++)
		{
			LowerX = int(c * DeltaX);
			UpperX = int(LowerX + DeltaX);
			LowerY = int(r * DeltaY);
			UpperY = int(LowerY + DeltaY);
			MultipleLinearRegression(BestFitPlanes[r][c], LowerX, UpperX, LowerY, UpperY);
		}
	}
	ProximityEvaluation();
}

void Cloud::MultipleLinearRegression(Plane*& P, int PixelLowerX, int PixelUpperX, int PixelLowerY, int
PixelUpperY)
{
	P = 0;
	float A[3][3];
```

Cloud.cpp

```
float b[3];
// Clear A and b.
for(int r = 0; r < 3; r++)
{
	for(int c = 0; c < 3; c++)
		A[r][c] = 0;
	b[r] = 0;
}

// For matrix A, the sums are kept in the lower diagonal portion of the matrix
// and then copied afterwards to make the matrix symmetric.
float Count = 0;
bool BoundsInitialized = false;
float WorldLowerX, WorldLowerY, WorldUpperX, WorldUpperY;
for(int i = 0; i < PointCount; i++)
{
	Point& p = Points[i];
	if(p.j >= PixelLowerX && p.j <= PixelUpperX && p.i >= PixelLowerY && p.i <= PixelUpperY)
	{
		if(!BoundsInitialized)
		{
			WorldLowerX = p.x;
			WorldUpperX = p.x;
			WorldLowerY = p.y;
			WorldUpperY = p.y;
			BoundsInitialized = true;
		}
		else
		{
			if(p.x < WorldLowerX) WorldLowerX = p.x;
			if(p.x > WorldUpperX) WorldUpperX = p.x;
			if(p.y < WorldLowerY) WorldLowerY = p.y;
			if(p.y > WorldUpperY) WorldUpperY = p.y;
		}
		A[0][0] += p.x * p.x;
		A[0][1] += p.x * p.y;
		A[0][2] += p.x;
		A[1][1] += p.y * p.y;
```

Cloud.cpp

```
                A[1][2] += p.y;
                b[0] += p.x * p.z;
                b[1] += p.y * p.z;
                b[2] += p.z;
                Count += 1.0;
            }
        }
        if(Count < 4)
            return;
        A[2][2] = Count;
        A[1][0] = A[0][1];
        A[2][0] = A[0][2];
        A[2][1] = A[1][2];

        float A_inverse[3][3];
        float Inverse_Determinant = 1.0f / (A[0][0] * (A[2][2] * A[1][1] - A[2][1] * A[1][2]) - A[1][0] *
(A[2][2] * A[0][1] - A[2][1] * A[0][2]) + A[2][0] * (A[1][2] * A[0][1] - A[1][1] * A[0][2]));
        A_inverse[0][0] = (A[1][1] * A[2][2] - A[1][2] * A[2][1]) * Inverse_Determinant;
        A_inverse[0][1] = (A[0][2] * A[2][1] - A[0][1] * A[2][2]) * Inverse_Determinant;
        A_inverse[0][2] = (A[0][1] * A[1][2] - A[0][2] * A[1][1]) * Inverse_Determinant;
        A_inverse[1][0] = (A[1][2] * A[2][0] - A[1][0] * A[2][2]) * Inverse_Determinant;
        A_inverse[1][1] = (A[0][0] * A[2][2] - A[0][2] * A[2][0]) * Inverse_Determinant;
        A_inverse[1][2] = (A[0][2] * A[1][0] - A[0][0] * A[1][2]) * Inverse_Determinant;
        A_inverse[2][0] = (A[1][0] * A[2][1] - A[1][1] * A[2][0]) * Inverse_Determinant;
        A_inverse[2][1] = (A[0][1] * A[2][0] - A[0][0] * A[2][1]) * Inverse_Determinant;
        A_inverse[2][2] = (A[0][0] * A[1][1] - A[0][1] * A[1][0]) * Inverse_Determinant;

        float x[3];
        x[0] = A_inverse[0][0] * b[0] + A_inverse[0][1] * b[1] + A_inverse[0][2] * b[2];
        x[1] = A_inverse[1][0] * b[0] + A_inverse[1][1] * b[1] + A_inverse[1][2] * b[2];
        x[2] = A_inverse[2][0] * b[0] + A_inverse[2][1] * b[1] + A_inverse[2][2] * b[2];

        P = new Plane;
        P->Normal.x = x[0];
        P->Normal.y = x[1];
        P->Normal.z = x[2];
        P->Normal /= len(P->Normal);
        // Find the angle between the best-fit plane's normal vector and the z-axis.
```

Cloud.cpp

```
	Vector Z_Axis;
	Z_Axis.x = 0;
	Z_Axis.y = 0;
	Z_Axis.z = 1;
	P->AngleFromZ = AngularDifference(P->Normal, Z_Axis);
	P->P1.x = WorldLowerX;
	P->P1.y = WorldLowerY;
	P->P1.z = x[0] * P->P1.x + x[1] * P->P1.y + x[2];
	P->P2.x = WorldUpperX;
	P->P2.y = WorldLowerY;
	P->P2.z = x[0] * P->P2.x + x[1] * P->P2.y + x[2];
	P->P3.x = WorldUpperX;
	P->P3.y = WorldUpperY;
	P->P3.z = x[0] * P->P3.x + x[1] * P->P3.y + x[2];
	P->P4.x = WorldLowerX;
	P->P4.y = WorldUpperY;
	P->P4.z = x[0] * P->P4.x + x[1] * P->P4.y + x[2];
	// Find the mean (to be used in the kurtosis calculation.)
#if USE_KURTOSIS
	float Mean = 0;
	for(int i = 0; i < PointCount; i++)
	{
		Point& p = Points[i];
		if(p.j >= PixelLowerX && p.j <= PixelUpperX && p.i >= PixelLowerY && p.i <= PixelUpperY)
		{
			float BestFitZ = x[0] * p.x + x[1] * p.y + x[2];
			Mean += (BestFitZ - p.z);
		}
	}
	Mean /= Count;
	// Find the kurtosis of the plane.

	float m_2 = 0;
	float m_4 = 0;
	float Temp;
	for(int i = 0; i < PointCount; i++)
	{
		Point& p = Points[i];
```

Cloud.cpp

```
			if(p.j >= PixelLowerX && p.j <= PixelUpperX && p.i >= PixelLowerY && p.i <= PixelUpperY)
			{
				float BestFitZ = x[0] * p.x + x[1] * p.y + x[2];
				Temp = (BestFitZ - p.z) - Mean;
				m_2 += pow(Temp, 2);
				m_4 += pow(Temp, 4);
			}
		}
		m_2 /= Count;
		m_4 /= Count;
		P->Kurtosis = (m_4 / pow(m_2, 2)) - 3;
#endif
}

void Cloud::ProximityEvaluation()
{
		for(int r = 0; r < Rows; r++)
		{
			for(int c = 0; c < Columns; c++)
			{
				float Count = 0;
				float AverageAdjacent = 0;
				Plane* P = BestFitPlanes[r][c];
				if(P == 0) continue;
				AverageAdjacent += CompareNormals(r, c, r - 1, c - 1, Count);
				AverageAdjacent += CompareNormals(r, c, r - 1, c, Count);
				AverageAdjacent += CompareNormals(r, c, r - 1, c + 1, Count);
				AverageAdjacent += CompareNormals(r, c, r, c - 1, Count);
				AverageAdjacent += CompareNormals(r, c, r, c + 1, Count);
				AverageAdjacent += CompareNormals(r, c, r + 1, c - 1, Count);
				AverageAdjacent += CompareNormals(r, c, r + 1, c, Count);
				AverageAdjacent += CompareNormals(r, c, r + 1, c + 1, Count);
				if(Count == 0)
					AverageAdjacent = numeric_limits<float>::infinity();
				P->AverageAdjacent = AverageAdjacent / Count;
			}
		}
}
```

Cloud.cpp

```
float Cloud::CompareNormals(int OriginRow, int OriginColumn, int TargetRow, int TargetColumn, float& Count)
{
	if(TargetRow < 0 || TargetColumn < 0 || TargetRow >= Rows || TargetColumn >= Columns) return 0;
	Plane* x = BestFitPlanes[OriginRow][OriginColumn];
	Plane* y = BestFitPlanes[TargetRow][TargetColumn];
	if(x == 0 || y == 0) return 0;
	Count++;
	return AngularDifference(x->Normal, y->Normal);
}

// This function locates the "largest traversable region" of the point cloud,
// and finds any drop-offs within this area
void Cloud::RangeCount()
{
	float* SubgraphSize;		// Array of subgraphs
	float SteepnessThreshold;	// Acceptable thresholds for the traversable regions
	float MaxAdjacentThreshold;
	const float MaxDropoff = 7;
	const float MaxZDiff = 0.07;
	int GreenCount = 0;
	int SubgraphID = 1;
	int CurrentID;
	int Largest = 0;
	float T_area;
	float density;
	//float PlaneArea = 0;
	SubgraphSize = new float [Rows*Columns];

	int mid_r, mid_c;
	mid_r = Rows / 2;
	mid_c = Columns / 2;

	Plane* T = BestFitPlanes[mid_r][mid_c];
	while(!T)
	{
		T = BestFitPlanes[mid_r][++mid_c];
	}
```

Cloud.cpp

```
T_area = (T->P2.x - T->P1.x)*(T->P3.y - T->P1.y);
density = (PointCount / (Rows*Columns));

SteepnessThreshold = 2100 / density;
MaxAdjacentThreshold = 2100 / density;

// Initialize Subgraph array
for(int i = 0; i < Rows*Columns; i++)
      SubgraphSize[i] = 0;

for(int r = 0; r < Rows-1; r++)
{
      for(int c = 0; c < Columns-1; c++)
      {
            float r_angle, d_angle, r_zdiff, d_zdiff;
            Plane* P = BestFitPlanes[r][c];
            if(!P) continue;

            // Check if this is a green square
            if(P->AngleFromZ < SteepnessThreshold && P->AverageAdjacent < MaxAdjacentThreshold)
            {
                  if(P->Subgraph == 0)    // Check if this has been assigned to a subgraph already
                  {
                        P->Subgraph = SubgraphID;
                        SubgraphID++;
                  }
                  // Increment the size of the subgraph
                  SubgraphSize[P->Subgraph]+= (P->P2.x - P->P1.x)*(P->P3.y - P->P1.y);
                  CurrentID = P->Subgraph;
                  Plane* R = BestFitPlanes[r][c+1];
                  Plane* D = BestFitPlanes[r+1][c];

                  if(R) // Look at plane to right of current plane
                  {
                        r_angle = AngularDifference(P->Normal, R->Normal);
                        r_zdiff = abs((P->P2.z + P->P3.z)/2.0 - (R->P4.z + R->P1.z)/2.0);
                        if(R->AngleFromZ < SteepnessThreshold && r_angle < MaxDropoff
```

```
                && r_zdiff < MaxZDiff && R->AverageAdjacent < MaxAdjacentThreshold)
        {
                // Check if R has already been assigned to a subgraph
                if(R->Subgraph != 0 && R->Subgraph != P->Subgraph)
                {
                        // If R is assigned already, assign all planes currently assigned
                        // to P's subgraph to that of R
                        for(int i=0; i < Rows;i++)
                        {
                                for(int j=0; j < Columns;j++)
                                {
                                        Plane *Q = BestFitPlanes[i][j];
                                        if(!Q) continue;
                                        if(Q->Subgraph == CurrentID)
                                                BestFitPlanes[i][j]->Subgraph = R->Subgraph;


                                }
                        }
                        P->Subgraph = R->Subgraph;

                        SubgraphSize[R->Subgraph] += SubgraphSize[CurrentID];
                        SubgraphSize[CurrentID] = 0;
                }
                else
                {
                        BestFitPlanes[r][c+1]->Subgraph = P->Subgraph;
                }
                P->R_safe = true;
                BestFitPlanes[r][c+1]->L_safe = true;
        }
}

if(D) // Look at plane below P
{
        d_angle = AngularDifference(P->Normal, D->Normal);
        d_zdiff = abs((P->P4.z + P->P3.z)/2.0 - (D->P2.z + D->P1.z)/2.0);
        if(D->AngleFromZ < SteepnessThreshold && d_angle < MaxDropoff
```

```
                                    && d_zdiff < MaxZDiff && D->AverageAdjacent < MaxAdjacentThreshold)
                              {
                                    BestFitPlanes[r+1][c]->Subgraph = P->Subgraph;
                                    P->D_safe = true;
                                    BestFitPlanes[r+1][c]->U_safe = true;
                              }
                        }

                  }

                  P->Marked = true;
                  BestFitPlanes[r][c]->Subgraph = P->Subgraph;
                  BestFitPlanes[r][c]->R_safe = P->R_safe;
                  BestFitPlanes[r][c]->D_safe = P->D_safe;

            }
      }

      for(int i=1; i < Rows*Columns; i++)
      {
            if(SubgraphSize[i] > SubgraphSize[Largest])
                  Largest = i;
      }

      LargestSubgraph = Largest;

      delete []SubgraphSize;

}



void Cloud::GetCurrentPos(string LogFileName)
{
      float distance, x_offset, y_offset;

      ifstream f(LogFileName.c_str(), ios_base::binary);
      f >> distance >> x_offset >> y_offset;
```

```
	f.close();

	for(int r = 0; r < Rows; r++)
	{
		for(int c = 0; c < Columns; c++)
		{
			Plane* P = BestFitPlanes[r][c];
			if(!P) continue;

			if((x_offset > P->P1.x) && (x_offset < P->P3.x)
				&& (y_offset > P->P1.y) && (y_offset < P->P3.y))
			{
				curr_x = c;
				curr_y = r;
			}
		}
	}

}

// This function searches the point cloud for possible power lines
void Cloud::LineDetect()
{
	long high_point = 0;
	long min = 0;
	int max_i, max_j, min_i, min_j;
	float max_z;
	float mean_z = 0;
	int Gy_m, Gx_m;
	long curr_point = 0;
	float edge_threshold = 60;
	int min_points = 130;
	int most_points = 0;
	int most_points_alt = 0;
	float divisor, angle_div;
	Point* TDPoints;
	const unsigned int ntheta = 90;
	const unsigned int nrho = 300;
```

Cloud.cpp

```
int rho;
unsigned int count;
float max_rho;
// Accumulator array for Hough transform
unsigned int Accumulator[ntheta][nrho];

for(int i=0;i<ntheta;i++)
{
	for(int j=0; j<nrho; j++)
		Accumulator[i][j] = 0;
}

angle_div = PI * (2 / (float)ntheta);

max_i = 0;
max_j = 0;
min_i = 500;
min_j = 500;
max_z = 0;
for(long i = 0; i < PointCount; i++)
{
	Point& p = Points[i];
	if(p.i > max_i)
		max_i = p.i;

	if(p.j > max_j)
		max_j = p.j;

	if(p.i < min_i)
		min_i = p.i;

	if(p.j < min_j)
		min_j = p.j;

	if(p.z > max_z)
		max_z = p.z;
}
```

```
max_j++;
max_i++;

max_rho = sqrt(pow(max_j,2.0)+pow(max_i,2.0));
divisor = max_rho / nrho;

TDPoints = new Point[max_i * max_j];



for(int i = 0; i < PointCount; i++)
{
	Point& p = Points[i];
	mean_z = mean_z*((float)(i)/((float)(i)+1)) + p.z/((float)(i)+1);
}

for(int i = 0; i < max_i; i++)
{
	for(int j = 0; j < max_j; j++)
	{
		TDPoints[i*max_j + j].z = 0.5;
		TDPoints[i*max_j + j].ref = -1;
	}
}

// Each point's h-value is defined as the difference of the mean
// z-value and the z-value of the point, multiplied by 30
for(int i = 0; i < PointCount; i++)
{
	Point& p = Points[i];
	curr_point = p.i*max_j + p.j;
	TDPoints[curr_point] = p;
	TDPoints[curr_point].ref = i;
	TDPoints[curr_point].h = (mean_z - TDPoints[curr_point].z) * 30;
	if(TDPoints[curr_point].h < -10)
		TDPoints[curr_point].h = -10;
}
```

```
	count = 0;

	for(int i = 0; i < max_i; i++)
	{
		for(int j = 0; j < max_j; j++)
		{
			if(TDPoints[i*max_j + j].z == 0.5)
				count++;
		}
	}

	count = 0;

	for(int i = min_i+3; i < (max_i-1); i++)
	{
		for(int j = min_j+3; j < (max_j-1); j++)
		{
			curr_point = i*max_j + j;

			// The Sobel edge detection routine is run on each point
			Gx_m = TDPoints[curr_point+max_j-1].h +  2*TDPoints[curr_point+max_j].h +
TDPoints[curr_point+1+max_j].h
				- (TDPoints[curr_point-1-max_j].h + 2*TDPoints[curr_point-max_j].h +
TDPoints[curr_point-max_j+1].h);

			Gy_m = TDPoints[curr_point-max_j+1].h + 2*TDPoints[curr_point+1].h +
TDPoints[curr_point+1+max_j].h
				- (TDPoints[curr_point-1-max_j].h + 2*TDPoints[curr_point-1].h +
TDPoints[curr_point+max_j-1].h);

			TDPoints[curr_point].grade = abs(abs(Gx_m)+abs(Gy_m));

			// Points found to be on an edge are run through the
			// Hough transform, and the Accumulator array is incremented
			if((TDPoints[curr_point].grade > edge_threshold))
			{
				count++;
				for(int t = 0; t < ntheta; t++)
```

```
                        {
                              rho = (j*cos(t*angle_div)+i*sin(t*angle_div)) / divisor;
                              if((rho < nrho) && (rho >= 0))
                              {
                                    Accumulator[t][rho] ++;
                              }
                        }
                        TDPoints[curr_point].on_edge = true;
                        //TDPoints[curr_point].r = 255;
                        //TDPoints[curr_point].b = 255;
                        //TDPoints[curr_point].g = 255;
                  }
                  else
                  {
                        //TDPoints[curr_point].r = 0;
                        //TDPoints[curr_point].b = 0;
                        //TDPoints[curr_point].g = 0;
                  }
            }
}


count = 0;
int most_t = -1;  // The parameters of the most voted-for
int most_r = -1;  //   line in the image
int most_t_alt = -1;// The parameters of the most voted-for line
int most_r_alt = -1;//   that has a sufficiently different angle
int diff_t;
int diff_r;
unsigned char alt_color;

// This algorithm searches for the most voted-for line in the image
for(int t = 0; t < ntheta; t++)
{
      for(int r = 0; r < nrho; r++)
      {
            if(Accumulator[t][r] > most_points)
            {
```

```
                        most_points = Accumulator[t][r];
                        most_t = t;
                        most_r = r;
                  }
            }
      }

      for(int t = 0; t < ntheta; t++)
      {
            for(int r = 0; r < nrho; r++)
            {
                  diff_t = abs(t-most_t);
                  diff_r = abs(r-most_r);
                  if(diff_t > (ntheta / 2))
                  {
                        if(most_t < (ntheta / 2))
                              diff_t = (ntheta - t) + most_t;
                        else
                              diff_t = (ntheta - most_t) + t;
                  }

                  // Check if current set of line parameters have
                  //    received a sufficient amount of votes
                  if((Accumulator[t][r] > min_points) && (((Accumulator[t][r] > ((most_points*4)/7)) &&
(diff_r > 50))
                        || (Accumulator[t][r] > ((most_points*5)/6))))
                  {
                        count++;

                        // If angle is sufficiently different, this represents
                        // a possible second power line
                        if((diff_r > 50) || (diff_t > 20))
                        {
                              alt_color = 255;
                              if(Accumulator[t][r] > most_points_alt)
                              {
                                    most_points_alt = Accumulator[t][r];
                                    most_t_alt = t;
```

```
                                    most_r_alt = r;
                              }
                        }
                        else
                        {
                              alt_color = 0;
                        }

                        // All points that lie on the selected line parameters
                        //    are highlighted for the user to see
                        for(int i=0; i < max_i; i++)
                        {
                              for(int j = 0; j< max_j; j++)
                              {
                                    if(r == (int)((j*cos(t*angle_div)+i*sin(t*angle_div)) / divisor))
                                    {
                                          curr_point = i*max_j + j;

                                          TDPoints[curr_point].r = 255;
                                          TDPoints[curr_point].g = 0;
                                          TDPoints[curr_point].b = alt_color;
                                          //TDPoints[curr_point].z = 2.0;

                                    }
                              }
                        }

                  }
            }
      }

// Current line parameters are compared with those of the last image
pline.ImageCompare(most_t, most_t_alt, ntheta);

// If the lines from the previous image are not found, a warning is
//     issued to the user
if(pline.t_1 != -1 && !pline.pl_1_found)
      MessageBox(NULL, "Power Line 1 Not Found!", "Warning", MB_ICONWARNING);
```

```
	if(pline.t_2 != -1 && !pline.pl_2_found)
		MessageBox(NULL, "Power Line 2 Not Found!", "Warning", MB_ICONWARNING);

	if(Accumulator[most_t][most_r] > min_points)
	{
		pline.r_1 = most_r;
		pline.t_1 = most_t;
	}
	else
	{
		pline.r_1 = -1;
		pline.t_1 = -1;
	}

	pline.r_2 = most_r_alt;
	pline.t_2 = most_t_alt;


	for(int i = 0; i < max_i; i++)
	{
		for(int j = 0; j < max_j; j++)
		{
			curr_point = i*max_j + j;
			if(TDPoints[curr_point].ref >= 0)
				Points[TDPoints[curr_point].ref] = TDPoints[curr_point];
		}
	}


	delete [] TDPoints;
}

void Cloud::Clear()
{
	DeleteBestFitPlanes();
	delete [] Points;
	ZeroParameters();
```

```
}

void Cloud::ZeroParameters()
{
	Image.Clear();
	PointCount = 0;
	Points = 0;
	BestFitPlanes = 0;
	DeltaX = 0;
	DeltaY = 0;
	Columns = 0;
	Rows = 0;
	LargestSubgraph = 0;
	curr_x = -1;
	curr_y = -1;
	old_x = -1;
	old_y = -1;
	//LogFileName = "";
}

void Cloud::DeleteBestFitPlanes()
{
	if(BestFitPlanes == 0)
		return;
	for(int r = 0; r < Rows; r++)
	{
		for(int c = 0; c < Columns; c++)
		{
			delete BestFitPlanes[r][c];
		}
		delete [] BestFitPlanes[r];
	}
	delete [] BestFitPlanes;
	BestFitPlanes = 0;
}

Cloud::~Cloud()
{
```

Cloud.cpp

```
	Clear();
}

Vector& operator/=(Vector& v, const float Scalar)
{
	v.x /= Scalar;
	v.y /= Scalar;
	v.z /= Scalar;
	return v;
}

float len(const Vector& v)
{
	return sqrt(v.x * v.x + v.y * v.y + v.z * v.z);
}

float AngularDifference(const Vector& u, const Vector& v)
{
	return acos(u.x * v.x + u.y * v.y + u.z * v.z) * (180.0f / PI);
}

ostream& operator<<(ostream& Out, const Point& p)
{
	Out << p.x << ' ' << p.y << ' ' << p.z << ' ' << p.r << ' ' << p.g << ' ' << p.b << ' ' << p.i << ' '
<< p.j << ' ' << p.d;
	return Out;
}
```

Render.h

```
#pragma once

#ifdef WIN32
#include <windows.h>
#endif
#include <GL/gl.h>
#include <GL/glut.h>
#include <GL/glui.h>
#include <vector>
#include <list>
#include <sstream>
#include "config.h"
#include "cloud.h"
#include "bumblebee.h"

#pragma comment(lib, "opengl32.lib")
#pragma comment(lib, "glu32.lib")

enum Representation
{
	REPRESENTATION_IS_2D,
	REPRESENTATION_IS_3D,
};

struct Cam
{
	Cam();
	float x, y, z, roll, pitch, yaw;
	bool ShowBestFitPlanes;
	bool Overlay;
	Representation Rep;
};

enum ColorType
{
	TOP_PICKS,
	SLOPE,
	AVERAGE_ANGLE,
```

Render.h

```
	THRESHOLD,
	KURTOSIS,
	RANGE,
};

void DrawFlightVector(const Point& Destination);
void ColorScene(bool SelectAcceptable);
void ResizeScene(GLsizei width, GLsizei height);
void OnDraw();
string Selection(bool Toggle);

extern Cam Camera;
extern vector<Cloud*> Clouds;
extern int MouseX, MouseY;
extern ColorType CurrentColorType;
extern string LogFileName;
```

Render.cpp

```
#include "render.h"

Point Dest; // FIX THIS!!!

Cam Camera;
string LogFileName;
vector<Cloud*> Clouds;
ColorType CurrentColorType;


Cam::Cam() : x(0), y(0), z(0), roll(0), pitch(0), yaw(0), ShowBestFitPlanes(true), Overlay(true),
Rep(REPRESENTATION_IS_2D)
{

}

void DrawFlightVector(const Point& Destination)
{
	glColor3f(1, 1, 1);
	glBegin(GL_LINES);
	glVertex3f(0, 0, 0);
	glVertex3f(Destination.x, Destination.y, Destination.z);
	glEnd();
}

void ColorScene(bool SelectAcceptable)
{
	float SteepnessThreshold; // degrees
	float MaxAdjacentThreshold; // degrees
	const int TopPickCount = 3;
	float density;
	LPCSTR unsafe;
	for(vector<Cloud*>::iterator i = Clouds.begin(); i != Clouds.end(); i++)
	{
		Cloud* C = (*i);

		density = C->PointCount / (C->Rows * C->Columns);
		SteepnessThreshold = 2100 / density;
```

```
		MaxAdjacentThreshold = 2100 / density;

		for(int r = 0; r < C->Rows; r++)
		{
			for(int c = 0; c < C->Columns; c++)
			{
				Plane* P = C->BestFitPlanes[r][c];
				if(!P) continue;
				else if(CurrentColorType == TOP_PICKS)
				{
					// TODO: implement this...
					// list<Plane*>
				}
				else if(CurrentColorType == SLOPE || CurrentColorType == AVERAGE_ANGLE ||
CurrentColorType == KURTOSIS)
				{
					float Scale;
					if(CurrentColorType == SLOPE)
						Scale = P->AngleFromZ / 90; // Show the slope as a red gradient
					else if(CurrentColorType == AVERAGE_ANGLE)
						Scale = P->AverageAdjacent / 180; // Show the max angle of an adjacent
planes as a red gradient
					else if(CurrentColorType == KURTOSIS)
						Scale = P->Kurtosis;
					if(Scale > 1) Scale = 1;
					if(Scale < 0) Scale = 0;
					float Color = 1 - Scale;
					P->P1.r = Color; P->P1.g = 0; P->P1.b = 0;
					P->P2.r = Color; P->P2.g = 0; P->P2.b = 0;
					P->P3.r = Color; P->P3.g = 0; P->P3.b = 0;
					P->P4.r = Color; P->P4.g = 0; P->P4.b = 0;
				}
				else if(CurrentColorType == THRESHOLD || CurrentColorType == RANGE)
				{
					bool Acceptable = false;
					if(CurrentColorType == RANGE && P->Subgraph == C->LargestSubgraph)
					{
						P->P1.r = 0; P->P1.g = 1; P->P1.b = 1;
```

```
                                    P->P2.r = 0; P->P2.g = 1; P->P2.b = 1;
                                    P->P3.r = 0; P->P3.g = 1; P->P3.b = 1;
                                    P->P4.r = 0; P->P4.g = 1; P->P4.b = 1;
                              }
                              else
                              {

                                    if(P->AngleFromZ < SteepnessThreshold && P->AverageAdjacent <
MaxAdjacentThreshold)

                                    {
                                          if(SelectAcceptable)
                                          {
                                                pair<int, int> p(c, r);
                                                C->Hits.Add(p);
                                          }
                                          Acceptable = true;
                                    }
                                    P->P1.r = !Acceptable; P->P1.g = Acceptable; P->P1.b = 0;
                                    P->P2.r = !Acceptable; P->P2.g = Acceptable; P->P2.b = 0;
                                    P->P3.r = !Acceptable; P->P3.g = Acceptable; P->P3.b = 0;
                                    P->P4.r = !Acceptable; P->P4.g = Acceptable; P->P4.b = 0;
                              }
                              // If the plane is at the current location of the robot,
                              // color it blue
                              if(CurrentColorType == RANGE && r == C->curr_y && c == C->curr_x)
                              {
                                    P->P1.r = 0; P->P1.g = 0; P->P1.b = 1;
                                    P->P2.r = 0; P->P2.g = 0; P->P2.b = 1;
                                    P->P3.r = 0; P->P3.g = 0; P->P3.b = 1;
                                    P->P4.r = 0; P->P4.g = 0; P->P4.b = 1;
                              }

                        }
                  }
            }

            // If the robot location has moved to a plane bordering
            //    a drop-off or an unsafe plane, a warning is issued
```

```
			//	to the user
			if(CurrentColorType == RANGE && C->curr_x != -1 && C->curr_y != -1 &&
				(C->curr_x != C->old_x || C->curr_y != C->old_y))
			{
				Plane* P = C->BestFitPlanes[C->curr_y][C->curr_x];
				if(P)
				{
					if(!P->R_safe)
						MessageBox(NULL, "Unsafe Moves: Right", "Warning", MB_ICONWARNING);

					if(!P->D_safe)
						MessageBox(NULL, "Unsafe Moves: Down", "Warning", MB_ICONWARNING);

					if(!P->U_safe)
						MessageBox(NULL, "Unsafe Moves: Up", "Warning", MB_ICONWARNING);


					if(!P->L_safe)
						MessageBox(NULL, "Unsafe Moves: Left", "Warning", MB_ICONWARNING);
				}

				C->old_y = C->curr_y;
				C->old_x = C->curr_x;
			}
		}
}

// Note: this function is for debug purposes only.
// It draws all points of a cloud within the specified
// coordinates, and the best-fit plane.
void DrawBoundedElements(int x, int y, float LowerX, float UpperX, float LowerY, float UpperY)
{
	if(Clouds.size() < 1) return;
	Cloud* C = Clouds[0];
	float M = float(C->Image.MaxColor);
	glBegin(GL_POINTS);
	for(int q = 0; q < C->PointCount; q++)
	{
```

```
			Point& p = C->Points[q];
			if(p.j >= LowerX && p.j <= UpperX && p.i >= LowerY && p.i <= UpperY)
			{
				glColor3f(p.r / M, p.g / M, p.b / M);
				glVertex3f(p.x, p.y, p.z);
			}
		}
		glEnd();
		Plane* p = C->BestFitPlanes[x][y];
		if(p == 0) return;
		glBegin(GL_QUADS);
		glColor3f(p->P1.r, p->P1.g, p->P1.b); glVertex3f(p->P1.x, p->P1.y, p->P1.z);
		glColor3f(p->P2.r, p->P2.g, p->P2.b); glVertex3f(p->P2.x, p->P2.y, p->P2.z);
		glColor3f(p->P3.r, p->P3.g, p->P3.b); glVertex3f(p->P3.x, p->P3.y, p->P3.z);
		glColor3f(p->P4.r, p->P4.g, p->P4.b); glVertex3f(p->P4.x, p->P4.y, p->P4.z);
		glEnd();
}

// Known issues with this function:
// It will only draw the first cloud in the Clouds list.
void DrawRaw(bool Selection, bool Draw_3D_Overlay)
{
		const float Depth = 5;
		if(Clouds.size() < 1) return;
		Cloud* C = Clouds[0];

		if(!Selection)
		{
			glEnable(GL_TEXTURE_2D);
			glBindTexture(GL_TEXTURE_2D, C->Tex.ID);
			glColor3f(1, 1, 1);
		}

		const float Scale = 0.01f;
		float LowerX, UpperX, LowerY, UpperY;
		float Width = (float)C->Image.Width;
		float Height = (float)C->Image.Height;
		float X_Offset = Width / 2.0f;
```

Render.cpp

```
	float Y_Offset = Height / 2.0f;
	// Draw the planes with texture.
	int Name = 0;
	for(float x = 0; x < C->Columns; x++)
	{
		for(float y = 0; y < C->Rows; y++)
		{
			LowerX = x / (float)C->Columns;
			LowerY = y / (float)C->Rows;
			UpperX = (x + 1) / (float)C->Columns;
			UpperY = (y + 1) / (float)C->Rows;
			if(Selection) glLoadName(Name);
			pair<int, int> P;
			P.first = int(x);
			P.second = int(y);
			if(!Selection && C->Hits.Contains(P))
			{
				glDisable(GL_TEXTURE_2D);
				glColor3f(0, 0, 1);
				glBegin(GL_QUADS);
				glVertex3f((LowerX * Width - X_Offset) * Scale, (LowerY * Height - Y_Offset) *
Scale, Depth);
				glVertex3f((UpperX * Width - X_Offset) * Scale, (LowerY * Height - Y_Offset) *
Scale, Depth);
				glVertex3f((UpperX * Width - X_Offset) * Scale, (UpperY * Height - Y_Offset) *
Scale, Depth);
				glVertex3f((LowerX * Width - X_Offset) * Scale, (UpperY * Height - Y_Offset) *
Scale, Depth);
				glEnd();
				glColor3f(1, 0, 0);
				glBegin(GL_QUADS);
				glVertex3f((LowerX * Width - X_Offset) * Scale, (LowerY * Height - Y_Offset) *
Scale, Depth-0.001);
				glVertex3f((LowerX * Width - X_Offset) * Scale, (LowerY * Height - Y_Offset+1) *
Scale, Depth-0.001);
				glVertex3f((UpperX * Width - X_Offset) * Scale, (LowerY * Height - Y_Offset+1) *
Scale, Depth-0.001);
```

Render.cpp

```
                        glVertex3f((UpperX * Width - X_Offset) * Scale, (LowerY * Height - Y_Offset) *
Scale, Depth-0.001);
                        glEnd();
                        if(Draw_3D_Overlay) DrawBoundedElements(int(y), int(x), LowerX * Width, UpperX *
Width, LowerY * Height, UpperY * Height);
                        //glRasterPos3f((LowerX*Width - X_Offset) * Scale, (UpperY * Height - Y_Offset) *
Scale, Depth-.2);
                        //glutBitmapCharacter(GLUT_BITMAP_TIMES_ROMAN_24, 'D');
                        glColor3f(1, 1, 1);
                        glEnable(GL_TEXTURE_2D);
                  }
                  else
                  {
                        glBegin(GL_QUADS);
                        if(!Selection) glTexCoord2f(LowerX, LowerY); glVertex3f((LowerX * Width - X_Offset)
* Scale, (LowerY * Height - Y_Offset) * Scale, Depth);
                        if(!Selection) glTexCoord2f(UpperX, LowerY); glVertex3f((UpperX * Width - X_Offset)
* Scale, (LowerY * Height - Y_Offset) * Scale, Depth);
                        if(!Selection) glTexCoord2f(UpperX, UpperY); glVertex3f((UpperX * Width - X_Offset)
* Scale, (UpperY * Height - Y_Offset) * Scale, Depth);
                        if(!Selection) glTexCoord2f(LowerX, UpperY); glVertex3f((LowerX * Width - X_Offset)
* Scale, (UpperY * Height - Y_Offset) * Scale, Depth);
                        glEnd();
                  }
                  Name++;
            }
      }

      if(!Selection)
            glDisable(GL_TEXTURE_2D);
}

void DrawCloud()
{
      glBegin(GL_POINTS);
      for(vector<Cloud*>::const_iterator c = Clouds.begin(); c != Clouds.end(); c++)
      {
            float M = float((*c)->Image.MaxColor);
```

```
			for(int i = 0; i < (*c)->PointCount; i++)
			{
				Point& p = (*c)->Points[i];
				glColor3f(p.r / M, p.g / M, p.b / M);
				glVertex3f(p.x, p.y, p.z);
			}
		}
		glEnd();
}

void DrawPlanes(bool Selection = false)
{
		for(vector<Cloud*>::iterator i = Clouds.begin(); i != Clouds.end(); i++)
		{
			Cloud* const c = *i;
			if(c->BestFitPlanes == 0) continue;
			for(int row = 0; row < c->Rows; row++)
			{
				for(int col = 0; col < c->Columns; col++)
				{
					Plane* p = c->BestFitPlanes[row][col];
					if(p == 0) continue;
					if(Selection) glLoadName(p->Name);
					//if(CurrentColorType != RANGE)			// This is making everything teal
						p->Hit ? glColor3f(0, 0, 1) : glColor3f(p->P1.r, p->P1.g, p->P1.b);

					glBegin(GL_QUADS);
					glVertex3f(p->P1.x, p->P1.y, p->P1.z);
					glVertex3f(p->P2.x, p->P2.y, p->P2.z);
					glVertex3f(p->P3.x, p->P3.y, p->P3.z);
					glVertex3f(p->P4.x, p->P4.y, p->P4.z);

					if(CurrentColorType == RANGE && p->Subgraph == c->LargestSubgraph)
					{
						// Draw narrow purple quadrilaterals at dropoff locations
						//    if user is looking at the traversable region display
						glColor3f(1,0,1);
```

Render.cpp

```
                                    if(!(p->R_safe))
                                    {
                                          glVertex3f(p->P2.x, p->P2.y, p->P2.z-0.02);
                                          glVertex3f(p->P2.x-0.008, p->P2.y, p->P2.z-0.02);
                                          glVertex3f(p->P3.x-0.008, p->P3.y, p->P3.z-0.02);
                                          glVertex3f(p->P3.x, p->P3.y, p->P3.z-0.02);
                                    }
                                    if(!(p->D_safe))
                                    {
                                          glVertex3f(p->P4.x, p->P4.y, p->P4.z-0.02);
                                          glVertex3f(p->P4.x, p->P4.y-0.008, p->P4.z-0.02);
                                          glVertex3f(p->P3.x, p->P3.y-0.008, p->P3.z-0.02);
                                          glVertex3f(p->P3.x, p->P3.y, p->P3.z-0.02);
                                    }


                              }
                              glEnd();
                              glColor3f(1, 1, 1);
                        }
                  }
#if USE_TRACKER
                  c->GetCurrentPos(LogFileName);
                  if(c->curr_x != c->old_x || c->curr_y != c->old_y)
                        ColorScene(true);
#endif

            }


}

void OnDraw()
{
      glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Clear screen and depth buffer
      glLoadIdentity(); // Reset the current model view matrix
      glScalef(1, -1, -1);
      glTranslatef(Camera.x, Camera.y, Camera.z);
```

```
	glRotatef(Camera.pitch, 1, 0, 0);
	glRotatef(Camera.yaw,   0, 1, 0);
	glRotatef(Camera.roll,  0, 0, 1);

	DrawFlightVector(Dest); // FIX THIS!!!

	if(Camera.Rep == REPRESENTATION_IS_2D)
		DrawRaw(false, Camera.Overlay);
	else if(Camera.Rep == REPRESENTATION_IS_3D)
	{
		DrawCloud();
		if(Camera.ShowBestFitPlanes)
			DrawPlanes();
	}

	// Currently we've been drawing to the back buffer, we need
	// to swap the back buffer with the front one to make the image visible
	glutSwapBuffers();
}

// Known issues:
// 2D representation selection will only work with first cloud in the Clouds list.
string Selection(bool Toggle)
{
	const int BUFFER_SIZE = 512;

	// Mark all the planes as not hit and name them.
	int Name = 0;
	if(!Toggle) Clouds[0]->Hits.Clear();
	for(vector<Cloud*>::iterator i = Clouds.begin(); i != Clouds.end(); i++)
		for(int x = 0; x < (*i)->Columns; x++)
			for(int y = 0; y < (*i)->Rows; y++)
				if((*i)->BestFitPlanes[x][y])
				{
					if(!Toggle) (*i)->BestFitPlanes[x][y]->Hit = false;
					(*i)->BestFitPlanes[x][y]->Name = Name;
					Name++;
				}
```

Render.cpp

```
	GLuint Buffer[BUFFER_SIZE]; // Create the selection buffer and zero it
	memset(Buffer, 0, sizeof(GLuint) * BUFFER_SIZE);
	GLint viewport[4]; // Create a viewport
	glGetIntegerv(GL_VIEWPORT, viewport); // Set the viewport
	glSelectBuffer(BUFFER_SIZE, Buffer);// Set the select buffer
	glRenderMode(GL_SELECT); // Put OpenGL in select mode
	glInitNames(); // Initialize the name stack
	glPushName(0); // Push a fake ID on the stack to prevent load error

	glMatrixMode(GL_PROJECTION);
	glPushMatrix();
	glLoadIdentity();

	// Setup a pick matrix
	gluPickMatrix((GLdouble)MouseX, (GLdouble)(viewport[3] - MouseY), 1.0f, 1.0f, viewport);
	gluPerspective(45.0f, (GLfloat)(viewport[2] - viewport[0]) / (GLfloat)(viewport[3] - viewport[1]),
0.1f, 100.0f);
	glMatrixMode(GL_MODELVIEW);

	glLoadIdentity(); // Reset the current matrix
	glScalef(1, -1, -1);
	glTranslatef(Camera.x, Camera.y, Camera.z);
	glRotatef(Camera.pitch, 1, 0, 0);
	glRotatef(Camera.yaw,   0, 1, 0);
	glRotatef(Camera.roll,  0, 0, 1);
	// Draw to the pick matrix instead of our normal one
	if(Camera.Rep == REPRESENTATION_IS_2D)
		DrawRaw(true, false);
	else if(Camera.Rep == REPRESENTATION_IS_3D)
		DrawPlanes(true);


	glMatrixMode(GL_PROJECTION);
	glPopMatrix();
	glMatrixMode(GL_MODELVIEW);
	glPopName();
	GLint hits = glRenderMode(GL_RENDER); // Count the hits
```

Render.cpp

```
	if(Camera.Rep == REPRESENTATION_IS_2D)
	{
		if(hits > 0)
		{
			int Hit = Buffer[3];
			pair<int, int> P;
			P.first = Hit / Clouds[0]->Columns;
			Hit -= P.first * Clouds[0]->Columns;
			P.second = Hit;
			Clouds[0]->Hits.Toggle(P);
			Plane* p = Clouds[0]->BestFitPlanes[P.second][P.first];
			stringstream ItemInfo;
			if(p != 0)
			{
				ItemInfo << "Row: " << P.second << "Column: " << P.first << "Slope: " << p-
>AngleFromZ << ", max adjacent: " << p->AverageAdjacent << ", kurtosis: " << p->Kurtosis;
				ItemInfo << ", destination (" << p->P1.x << ", " << p->P1.y << ", " << p->P1.z <<
")";
				Dest.x = p->P1.x; // FIX THIS!!!
				Dest.y = p->P1.y;
				Dest.z = p->P1.z;
				return ItemInfo.str();
			}
		}
	}
	else if(Camera.Rep == REPRESENTATION_IS_3D)
	{
		// Find the object that was hit (if any)
		// Note: this is a very primitive way of processing the hit list for OpenGL selection.
		// This should probably be improved in the future to account for proper occlusion.
		if(hits > 0)
			for(vector<Cloud*>::iterator i = Clouds.begin(); i != Clouds.end(); i++)
				for(int x = 0; x < (*i)->Columns; x++)
					for(int y = 0; y < (*i)->Rows; y++)
						if((*i)->BestFitPlanes[x][y] && (*i)->BestFitPlanes[x][y]->Name ==
Buffer[3])
						{
```

Render.cpp

```
                                        Plane* p = (*i)->BestFitPlanes[x][y];
                                        p->Hit = !(p->Hit);
                                        if(CurrentColorType == RANGE)
                                              p->Hit = false;
                                        (*i)->curr_x = y;
                                        (*i)->curr_y = x;
                                        stringstream ItemInfo;
                                        ItemInfo << "Slope: " << p->AngleFromZ << ", average adjacent: "
<< p->AverageAdjacent << ", kurtosis: " << p->Kurtosis;
                                        ItemInfo << ", destination (" << p->P1.x << ", " << p->P1.y << ",
" << p->P1.z << ")";

                                        Dest.x = p->P1.x; // FIX THIS!!!
                                        Dest.y = p->P1.y;
                                        Dest.z = p->P1.z;
                                        return ItemInfo.str();
                                  }
            }
            return "";
}
```

Driver.cpp

```
#define _WIN32_WINNT 0x0400 // This program must be used with Windows NT or later version.

#include "config.h"
#if ANALYZE_MEMORY
#define _CRTDBG_MAP_ALLOC
#include <stdlib.h>
#include <crtdbg.h>
#endif

#include <windows.h>
#include <gl/glut.h>
#include <gl/glui.h>
#include <ctime>
#include <locale>
#include <iostream>
#include <fstream>
#include "resource.h"
#include "cloud.h"
#include "render.h"
#include "bumblebee.h"
using namespace std;

#define TWOD      0
#define THREED    1
#define ID_VIEW 0
#define ID_VIEW2 1
#define INVALID (-1)
int MouseX;
int MouseY;
int MouseButton;
bool MotionEnabled;
bool HasMoved;
bool ToggleSelection = true;
void GetLogFile();
ofstream Out;

void OnReshape(int w, int h)
{
```

```
	if(h == 0) h = 1;

	// Set the drawable region of the window
	glViewport(0, 0, w, h);

	// Set up the projection matrix
	glMatrixMode(GL_PROJECTION);
	glLoadIdentity();

	// Calculate the aspect ratio of the window
	gluPerspective(45.0f, (GLfloat)w/(GLfloat)h, 0.1f, 100.0f);

	// Go back to modelview matrix so we can move the objects about
	glMatrixMode(GL_MODELVIEW);
}

void OnSpecialKey(int Key, int x, int y)
{
	// If the user is viewing the largest traversable region
	//    display, the arrow keys function as robot move commands
	if(CurrentColorType == RANGE && Clouds[0]->curr_x != -1)
	{
		if(Key == GLUT_KEY_RIGHT)
			Clouds[0]->curr_x++;
		else if(Key == GLUT_KEY_LEFT)
			Clouds[0]->curr_x--;
		else if(Key == GLUT_KEY_UP)
			Clouds[0]->curr_y--;
		else if(Key == GLUT_KEY_DOWN)
			Clouds[0]->curr_y++;
	}
	else
	{
		if(Key == GLUT_KEY_RIGHT)
			Camera.x -= 0.3f;
		else if(Key == GLUT_KEY_LEFT)
			Camera.x += 0.3f;
		else if(Key == GLUT_KEY_UP)
```

Driver.cpp

```
                Camera.y -= 0.3f;
            else if(Key == GLUT_KEY_DOWN)
                Camera.y += 0.3f;
      }
}

void OnSpecialKeyUp(int Key, int x, int y)
{
      if(Key == GLUT_KEY_F1)
            CurrentColorType = SLOPE;
      else if(Key == GLUT_KEY_F2)
            CurrentColorType = AVERAGE_ANGLE;
      else if(Key == GLUT_KEY_F3)
            CurrentColorType = THRESHOLD;
      else if(Key == GLUT_KEY_F4)
            CurrentColorType = KURTOSIS;
      else if(Key == GLUT_KEY_F5)
            CurrentColorType = RANGE;
      ColorScene(false);
}

void OnKey(unsigned char Key, int x, int y)
{
      // Convert the key to lowercase
      Key = use_facet<ctype<string::value_type>>(locale()).tolower(Key);
      if(Key == 'z')
            Camera.z -= 0.3f;
      else if(Key == 'x')
            Camera.z += 0.3f;
      else if(Key == 'd')
            Camera.yaw -= 3;
      else if(Key == 'a')
            Camera.yaw += 3;
      else if(Key == 's')
            Camera.pitch -= 3;
      else if(Key == 'w')
            Camera.pitch += 3;
      else if(Key == 'e')
```

Driver.cpp

```
			Camera.roll -= 3;
		else if(Key == 'q')
			Camera.roll += 3;
}

void OnKeyUp(unsigned char Key, int x, int y)
{
		// Convert the key to lowercase
		Key = use_facet<ctype<string::value_type>>(locale()).tolower(Key);
		if(Key == '0')
		{
			Camera.x = 0;
			Camera.y = 0;
			Camera.z = 0;
			Camera.roll = 0;
			Camera.pitch = 0;
			Camera.yaw = 0;
			Camera.ShowBestFitPlanes = true;
		}
		else if(Key == '2')
			Camera.Rep = REPRESENTATION_IS_2D;
		else if(Key == '3')
			Camera.Rep = REPRESENTATION_IS_3D;
		else if(Key == '5')
			Camera.Overlay = !Camera.Overlay;
		else if(Key == 't')
			ToggleSelection = !ToggleSelection;
		else if(Key == 'b')
			Camera.ShowBestFitPlanes = !Camera.ShowBestFitPlanes;
		#if BUMBLEBEE_INSTALLED
		else if(Key == 'c')
		{
			for(vector<Cloud*>::iterator i = Clouds.begin(); i != Clouds.end(); i++)
				delete *i;
			Clouds.clear();
			Cloud* C = new Cloud();
			Bumblebee->Capture(*C);
			C->ComposeGrid(30, 30);
```

```
		// Test the assignment operator...
		Cloud* D = new Cloud();
		*D = *C;
		delete C;
		Clouds.push_back(D);
		ColorScene(CurrentColorType, false);
}
#endif
else if(Key == 'i')
{
		OPENFILENAME ofn; // common dialog box structure
		char szFile[1000];      // buffer for file name
		// Initialize OPENFILENAME
		ZeroMemory(&ofn, sizeof(ofn));
		ofn.lStructSize = sizeof(ofn);
		ofn.lpstrTitle = "Choose point cloud file: ";
		ofn.hwndOwner = 0; // hWnd;
		ofn.lpstrFile = szFile;
		// Set lpstrFile[0] to '\0' so that GetOpenFileName does not
		// use the contents of szFile to initialize itself.
		ofn.lpstrFile[0] = '\0';
		ofn.nMaxFile = sizeof(szFile);
		ofn.lpstrFilter = "Portable pixel map (*.ppm)\0*.ppm\0All (*.*)\0*.*\0";
		ofn.nFilterIndex = 1;
		ofn.lpstrFileTitle = NULL;
		ofn.nMaxFileTitle = 0;
		ofn.lpstrInitialDir = NULL;
		ofn.Flags = OFN_PATHMUSTEXIST | OFN_FILEMUSTEXIST;
		// Display the Open dialog box.
		if(GetOpenFileName(&ofn) == TRUE)
		{
			for(vector<Cloud*>::iterator i = Clouds.begin(); i != Clouds.end(); i++)
				delete *i;
			Clouds.clear();
			Cloud* C = new Cloud();
			DWORD t_start;
			t_start = GetTickCount();
			cout << "Opening file " << (char*)(ofn.lpstrFile) << endl;
```

```
                C->Open((char*)(ofn.lpstrFile));
                cout << "Open time = " << (GetTickCount() - t_start) << endl;
                t_start = GetTickCount();
                //if(C->Has2D)
                      C->ComposeGrid(30, 30);
                //else
                if(!C->Has2D)
                      Camera.Rep = REPRESENTATION_IS_3D;
                cout << "Analysis time = " << (GetTickCount() - t_start) << endl;
                cout << "Number of points = " << C->PointCount << endl;
                // Test the assignment operator...
                Cloud* D = new Cloud();
                *D = *C;
                delete C;
                Clouds.push_back(D);
                Clouds[0]->RangeCount();
                CurrentColorType = RANGE;
                ColorScene(false);

                //Clouds[0]->GetLogFile();

        }
}
else if(Key == 'o')
{
        if(!Clouds.empty())
                Clouds[0]->Save();
}
else if(Key == 'm')
        ColorScene(true);
else if(Key == 'p')
{
        Clouds[0]->LineDetect();
}
else if(Key == 'l')
{
        //Clouds[0]->GetCurrentPos();
        ColorScene(true);
```

```
	}
}

void OnIdle()
{
	glutPostRedisplay();
}

void OnMouse(int Button, int State, int x, int y)
{
//	Mouse wheel
//	const float ZoomScaleFactor = 150;
//	Camera.z -= NumberOfTurns / ZoomScaleFactor;
	if(State == GLUT_DOWN && (Button == GLUT_LEFT_BUTTON || Button == GLUT_RIGHT_BUTTON))
	{
		MouseX = x;
		MouseY = y;
		MouseButton = Button;
		MotionEnabled = true;
	}
	else
	{
		if(State == GLUT_UP && Button == GLUT_LEFT_BUTTON && !HasMoved)
		{
			string SectionInfo = Selection(ToggleSelection);

			glutSetWindowTitle(SectionInfo.c_str());
		}
		MotionEnabled = false;
	}
	HasMoved = false;
	//Clouds[0]->curr_x;
	ColorScene(false);
}

void OnMenuSelect(int option)
{
	switch(option){
```

Driver.cpp

```
		case TWOD :
			Camera.Rep = REPRESENTATION_IS_2D;
			break;

		case THREED :
			Camera.Rep = REPRESENTATION_IS_3D;
			break;
	}

}

void OnMotion(int x, int y)
{
	HasMoved = true;
	short Shift = GetAsyncKeyState(VK_SHIFT) & 0xff00;
	if(MotionEnabled)
	{
		const float TranslationScaleFactor = 500;
		const float RotationScaleFactor = 2;
		if(MouseButton == GLUT_LEFT_BUTTON)
		{
			if(Shift)
			{
				Camera.pitch -= (MouseY - y) / RotationScaleFactor;
				Camera.roll -= (MouseX - x) / RotationScaleFactor;
			}
			else
			{
				Camera.pitch -= (MouseY - y) / RotationScaleFactor;
				Camera.yaw -= (MouseX - x) / RotationScaleFactor;
			}
		}
		else if(MouseButton == GLUT_RIGHT_BUTTON)
		{
			if(Shift)
			{
				Camera.x -= (MouseX - x) / TranslationScaleFactor;
```

```
                        Camera.z -= (MouseY - y) / TranslationScaleFactor;
                }
                else
                {
                        Camera.x -= (MouseX - x) / TranslationScaleFactor;
                        Camera.y -= (MouseY - y) / TranslationScaleFactor;
                }
        }
    }
    MouseX = x;
    MouseY = y;
}


bool Initialize()
{
    WNDCLASS wc;
    Out.open("cout.txt");
    if(Out.is_open())
        cout.rdbuf(Out.rdbuf());
    #if BUMBLEBEE_INSTALLED
    InitializeBumblebee();
    #endif

    int OpenGL_argc = 1;
    char* OpenGL_argv = "program";
    glutInit(&OpenGL_argc, &OpenGL_argv);
    glutInitDisplayMode(GLUT_DEPTH | GLUT_RGBA | GLUT_DOUBLE);
    glutInitWindowSize(640, 480);
    glutCreateWindow("Hager's Point Cloud Viewer");


    glutDisplayFunc(OnDraw);
    glutReshapeFunc(OnReshape);
    glutSpecialFunc(OnSpecialKey);
    glutSpecialUpFunc(OnSpecialKeyUp);
    glutKeyboardFunc(OnKey);
    glutKeyboardUpFunc(OnKeyUp);
```

Driver.cpp

```
	glutIdleFunc(OnIdle);
	glutMouseFunc(OnMouse);



	glutMotionFunc(OnMotion);

#if USE_TRACKER
	GetLogFile();
#endif


	/*
	glutCreateMenu(OnMenuSelect);
	glutAddMenuEntry("2D", TWOD);
	glutAddMenuEntry("3D", THREED);
	glutAttachMenu(GLUT_RIGHT_BUTTON);
	*/
	/*
	HWND mywindow = FindWindow("GLUT", "Hager's Point Cloud Viewer");

	HMENU menubar = CreateMenu();
	InsertMenu(menubar, ID_VIEW, MF_STRING, ID_VIEW, "File");
	InsertMenu(menubar, ID_VIEW2, MF_STRING, ID_VIEW2, "View");
	SetMenu(mywindow, menubar);

	*/
	glLoadIdentity();
	glEnable(GL_DEPTH_TEST);


	//GLUI *glui = GLUI_Master.create_glui("Hager");
	return true;
}

void Finalize()
{
	for(vector<Cloud*>::iterator i = Clouds.begin(); i != Clouds.end(); i++)
```

Driver.cpp

```
			delete *i;
	#if ANALYZE_MEMORY
	_CrtDumpMemoryLeaks();
	#endif
	Out.close();
}

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow)
{
	if(Initialize())
		glutMainLoop();
	Finalize();
	return 0;
}

void GetLogFile()
{
	OPENFILENAME lfn;
	float distance, x_offset, y_offset;

	char szFile[1000];	// buffer for file name
	// Initialize OPENFILENAME
	ZeroMemory(&lfn, sizeof(lfn));
	lfn.lStructSize = sizeof(lfn);
	lfn.lpstrTitle = "Choose logfile for GSR position: ";
	lfn.hwndOwner = 0; // hWnd;
	lfn.lpstrFile = szFile;
	// Set lpstrFile[0] to '\0' so that GetOpenFileName does not
	// use the contents of szFile to initialize itself.
	lfn.lpstrFile[0] = '\0';
	lfn.nMaxFile = sizeof(szFile);
	lfn.lpstrFilter = "Text File (*.txt)\0*.txt\0All (*.*)\0*.*\0";
	lfn.nFilterIndex = 1;
	lfn.lpstrFileTitle = NULL;
	lfn.nMaxFileTitle = 0;
	lfn.lpstrInitialDir = NULL;
	lfn.Flags = OFN_PATHMUSTEXIST | OFN_FILEMUSTEXIST;
	GetOpenFileName(&lfn);
```

```
	LogFileName = lfn.lpstrFile;

	ifstream f(LogFileName.c_str(), ios_base::binary);

	f >> distance >> x_offset >> y_offset;

	f.close();

	/*
	FILE * logfile;
	char logfilename[15] = "log_data.txt";
	float distance, x_offset, y_offset;
	logfile = fopen(logfilename, "r");


	fscanf(logfile, "%f", &distance);
	fscanf(logfile, "%f", &x_offset);
	fscanf(logfile, "%f", &y_offset);
*/

}
```

Config.h

```
#pragma once

#define ANALYZE_MEMORY 1 // Print memory leaks to the output window?
#define REDIRECT_COUT 1 // Redirect standard out to "cout.txt"?
#define BUMBLEBEE_INSTALLED 0 // Has the bumblebee software been installed?
#define USE_TRACKER 0   // Use the GSR Tracking application
#define USE_KURTOSIS 0  // Calculate kurtosis of best-fit planes
```

Bumblebee.h

```
#pragma once
#include "config.h"
#if BUMBLEBEE_INSTALLED
#include <triclops.h>
#include <digiclops.h>
#include "cloud.h"
#include <sstream>
#include <iostream>
#include <ctime>
#include <list>
using namespace std;

#pragma comment(lib, "digiclops.lib")
#pragma comment(lib, "triclops.lib")

class BumblebeeCamera
{
public:
	bool Capture(Cloud& C);
	friend bool InitializeBumblebee();
private:
	TriclopsContext Triclops;
	DigiclopsContext Digiclops;
	bool Initialized;
	BumblebeeCamera();
	BumblebeeCamera(const BumblebeeCamera& B);
	void operator=(const BumblebeeCamera& B);
};

extern BumblebeeCamera* Bumblebee;
#endif
```

Bumblebee.cpp

```
#include "bumblebee.h"

#if BUMBLEBEE_INSTALLED

BumblebeeCamera* Bumblebee;

bool BumblebeeCamera::Capture(Cloud& C)
{
	if(!Initialized)
		return false;
	C.Clear();
	list<Point> Points;

	TriclopsInput StereoData;
	TriclopsInput ColorData;
	TriclopsImage16 DepthImage16;
	TriclopsColorImage ColorImage;

	// Grab the image set
	digiclopsGrabImage(Digiclops);
	// Grab the stereo data
	digiclopsExtractTriclopsInput(Digiclops, STEREO_IMAGE, &StereoData);
	// Grab the color image data
	digiclopsExtractTriclopsInput(Digiclops, RIGHT_IMAGE, &ColorData);
	// Preprocess the images
	triclopsPreprocess(Triclops, &StereoData);
	// Stereo processing
	triclopsStereo(Triclops);
	// Retrieve the interpolated depth image from the context
	triclopsGetImage16(Triclops, TriImg16_DISPARITY, TriCam_REFERENCE, &DepthImage16);
	triclopsRectifyColorImage(Triclops, TriCam_REFERENCE, &ColorData, &ColorImage);
	// Determine the number of pixels spacing per row
	int PixelInc = DepthImage16.rowinc / 2;
	for(int i = 0, k = 0; i < DepthImage16.nrows; i++)
	{
		unsigned short* Row = DepthImage16.data + i * PixelInc;
		for(int j = 0; j < DepthImage16.ncols; j++, k++)
		{
```

```
            int Disparity = Row[j];
            // Filter invalid points
            if(Disparity < 0xFF00)
            {
                  // Convert the 16 bit disparity value to floating point x, y, z
                  Point P;
                  triclopsRCD16ToXYZ(Triclops, i, j, Disparity, &P.x, &P.y, &P.z);
                  // Look at points within a range
                  if(P.z < 5.0)
                  {
                        P.i = i;
                        P.j = j;
                        P.d = Disparity;
                        P.r = (float)ColorImage.red[k];
                        P.g = (float)ColorImage.green[k];
                        P.b = (float)ColorImage.blue[k];
                        Points.push_back(P);
                  }
            }
      }
}
if(Points.size() == 0)
      return false;

C.PointCount = (long)Points.size();
C.Points = new Point[C.PointCount];
long q = 0;
for(list<Point>::iterator it = Points.begin(); it != Points.end(); it++, q++)
      C.Points[q] = *it;

// Now set the image data of the cloud
PPMImage& I = C.Image;
I.MaxColor = 255;
I.Height = ColorImage.nrows;
I.Width = ColorImage.ncols;
I.Timestamp = time(0);
I.Data = new GLubyte[3 * I.Height * I.Width];
for(q = 0; q < I.Height * I.Width; q++)
```

Bumblebee.cpp

```
	{
		I.Data[q * 3 + 0] = ColorImage.red[q];
		I.Data[q * 3 + 1] = ColorImage.green[q];
		I.Data[q * 3 + 2] = ColorImage.blue[q];
	}

	return true;
}

bool InitializeBumblebee()
{
	static BumblebeeCamera B;
	Bumblebee = &B;
	return B.Initialized;
}

// TODO: Make this more robust. If part of the initialization fails, then return with Initialized = false.
BumblebeeCamera::BumblebeeCamera()
{
	Initialized = false;
	// Open the Digiclops
	digiclopsCreateContext(&Digiclops);
	digiclopsInitialize(Digiclops, 0);
	// Get the camera module configuration
	digiclopsGetTriclopsContextFromCamera(Digiclops, &Triclops);
	// Set the digiclops to deliver the stereo image and right (color) image
	digiclopsSetImageTypes(Digiclops, STEREO_IMAGE | RIGHT_IMAGE);
	// Set the Digiclops resolution
	// Use 'HALF' resolution when you need faster throughput, especially for color images
//	digiclopsSetImageResolution(Digiclops, DIGICLOPS_HALF);
	digiclopsSetImageResolution(Digiclops, DIGICLOPS_FULL);
	// Start grabbing images with the camera
	digiclopsStart(Digiclops);
	// Set up some stereo parameters:
	// Set to 640x480 output images
	triclopsSetResolution(Triclops, 480, 640);
	// Set disparity range
	triclopsSetDisparity(Triclops, 1, 100);
```

```
	triclopsSetStereoMask(Triclops, 11);
	triclopsSetEdgeCorrelation(Triclops, 1);
	triclopsSetEdgeMask(Triclops, 11);
	// Turn on all validation
	triclopsSetTextureValidation(Triclops, 1);
	triclopsSetUniquenessValidation(Triclops, 1);
	// Turn on sub-pixel interpolation
	triclopsSetSubpixelInterpolation(Triclops, 1);
	// Make sure strict subpixel validation is on
	triclopsSetStrictSubpixelValidation(Triclops, 1);
	// Turn on surface validation
	triclopsSetSurfaceValidation(Triclops, 1);
	triclopsSetSurfaceValidationSize(Triclops, 200);
	triclopsSetSurfaceValidationDifference(Triclops, 0.5);
	Initialized = true;
}

BumblebeeCamera::BumblebeeCamera(const BumblebeeCamera& B)
{

}

void BumblebeeCamera::operator=(const BumblebeeCamera& B)
{

}

#endif
```

Gsrtracker.c

```
#ifdef _WIN32
#include <windows.h>
#endif
#include <stdio.h>
#include <stdlib.h>
#ifndef __APPLE__
#include <GL/gl.h>
#include <GL/glut.h>
#else
#include <OpenGL/gl.h>
#include <GLUT/glut.h>
#endif
#include <math.h>
#include <AR/gsub.h>
#include <AR/video.h>
#include <AR/param.h>
#include <AR/ar.h>

//
// Camera configuration.
//
#ifdef _WIN32
char             *vconf = "Data\\WDM_camera_flipV.xml";
#else
char             *vconf = "";
#endif

int              xsize, ysize;
int              thresh = 40;
int              count = 0;

char            *cparam_name    = "Data/camera_para.dat";
ARParam          cparam;

char            *patt_name      = "Data/patt.calib";
int              patt_id;
double           patt_width     = 80.0;
double           patt_center[2] = {0.0, 0.0};
```

Gsrtracker.c

```
double          patt_trans[3][4];

char        logfilename[15]   =     "log_data";
FILE * logfile;

float             x_center, y_center;
const float       target_size = 0.00258; // meters^2 (standard is 0.0410063), (small on tire is .0036),
(experiment is .00258064)
const float       tan_30 = 0.5973;     // tan(30.85 degrees)=0.5973
float             multiplier;                   // x_size / y_size
int                     total_pix;

static void   init(void);
static void   cleanup(void);
static void   keyEvent( unsigned char key, int x, int y);
static void   mainLoop(void);
static void   draw( void );

int main(int argc, char **argv)
{
      glutInit(&argc, argv);
      init();

    arVideoCapStart();
    argMainLoop( NULL, keyEvent, mainLoop );
      return (0);
}

static void   keyEvent( unsigned char key, int x, int y)
{
    /* quit if the ESC key is pressed */
    if( key == 0x1b ) {
        printf("*** %f (frame/sec)\n", (double)count/arUtilTimer());
        cleanup();
        exit(0);
    }
}
```

Gsrtracker.c

```
/* main loop */
static void mainLoop(void)
{
    ARUint8         *dataPtr;
    ARMarkerInfo    *marker_info;
    int             marker_num;
    int             j, k;
      float             distance, distance_image, image_area, real_pix_w, real_pix_h, x_off, y_off;


    /* grab a video frame */
    if( (dataPtr = (ARUint8 *)arVideoGetImage()) == NULL ) {
        arUtilSleep(2);
        return;
    }
    if( count == 0 ) arUtilTimerReset();
    count++;

    argDrawMode2D();
    argDispImage( dataPtr, 0,0 );

    /* detect the markers in the video frame */
    if( arDetectMarker(dataPtr, thresh, &marker_info, &marker_num) < 0 ) {
        cleanup();
        exit(0);
    }

    arVideoCapNext();

    /* check for object visibility */
    k = -1;
    for( j = 0; j < marker_num; j++ ) {
        if( patt_id == marker_info[j].id ) {
            if( k == -1 ) k = j;
            else if( marker_info[k].cf < marker_info[j].cf ) k = j;
        }
    }
    if( k == -1 ) {
```

```
            argSwapBuffers();
            // Print "Not found" to logfile
            logfile = fopen(logfilename, "a");
            fprintf(logfile, "Not found\n");
            fclose(logfile);
        return;
    }

    /* get the transformation between the marker and the real camera */
    arGetTransMat(&marker_info[k], patt_center, patt_width, patt_trans);

      // Print out the result for every other frame
      if((count % 1) == 0){
            // Compute size of 2d image area
            image_area = total_pix / marker_info[k].area * target_size;
            // Compute real-world size of pixels
            real_pix_w = sqrt(multiplier * image_area);
            real_pix_h = real_pix_w / multiplier;
            // Compute distance to center of image
            distance_image = (1 / (2*tan_30)) * real_pix_w;
            real_pix_w = real_pix_w / xsize;
            real_pix_h = real_pix_h / ysize;
            // Compute distances of marker to center of image
            //  for each axis
            x_off = real_pix_w * (marker_info[k].pos[0] - x_center);
            y_off = real_pix_h * (marker_info[k].pos[1] - y_center);
            // Compute total distance from camera to marker
            distance = sqrt((distance_image*distance_image)+(x_off*x_off)+(y_off*y_off));

            // Print results to screen
            printf("# of Markers: %i\n", marker_num);
            printf("Marker size: %d\n", marker_info[k].area);
            printf("Marker center: %f, %f\n", (double) marker_info[k].pos[0], (double)
marker_info[k].pos[1]);
            printf("Distance to marker: %f\n", distance);
            printf("Pixel sizes- x: %f y: %f\n", real_pix_w, real_pix_h);
            printf("Distance from center: %f, %f\n", x_off, y_off);
```

```
            // Print results to logfile
            logfile = fopen(logfilename, "w");
            fprintf(logfile, "%f %f %f\n", distance, x_off, y_off);
            fclose(logfile);

      }

    draw();

    argSwapBuffers();
}

static void init( void )
{
    ARParam  wparam;

      // Open logfile for printing results
      strcat(logfilename, ".txt");
      logfile = fopen(logfilename, "w");
      fprintf(logfile, "Distance, X_offset, Y_offset\n");
      fclose(logfile);

    /* open the video path */
    if( arVideoOpen( vconf ) < 0 ) exit(0);
    /* find the size of the window */
    if( arVideoInqSize(&xsize, &ysize) < 0 ) exit(0);
    printf("Image size (x,y) = (%d,%d)\n", xsize, ysize);

    /* set the initial camera parameters */
    if( arParamLoad(cparam_name, 1, &wparam) < 0 ) {
        printf("Camera parameter load error !!\n");
        exit(0);
    }
    arParamChangeSize( &wparam, xsize, ysize, &cparam );
    arInitCparam( &cparam );
    printf("*** Camera Parameter ***\n");
    arParamDisp( &cparam );
```

Gsrtracker.c

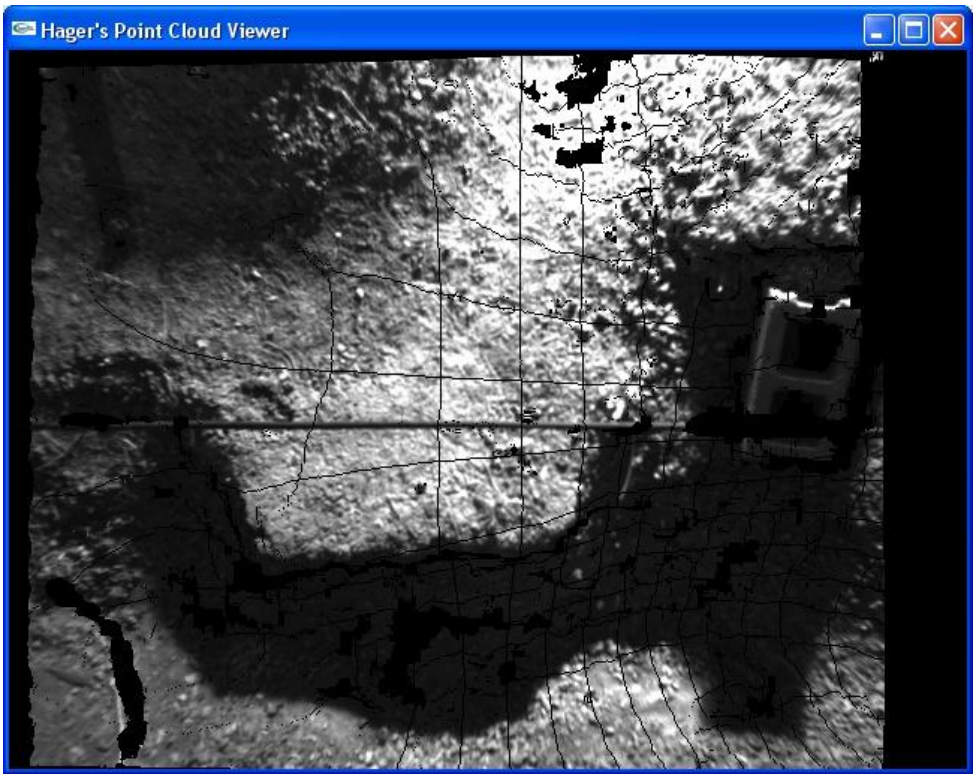
```
      // Set values of center pixels and total pixel size of image
      multiplier = (float)(xsize) / (float)(ysize);
      total_pix = xsize * ysize;
      x_center = xsize / 2;
      y_center = ysize / 2;

    if( (patt_id=arLoadPatt(patt_name)) < 0 ) {
        printf("pattern load error !!\n");
        exit(0);
    }

    /* open the graphics window */
    argInit( &cparam, 1.0, 0, 0, 0, 0 );
}

/* cleanup function called when program exits */
static void cleanup(void)
{
    arVideoCapStop();
    arVideoClose();
    argCleanup();
}

// Draws a virtual 3D object on top of the marker;
//    this is for display purposes to user only
static void draw( void )
{
    double    gl_para[16];
    GLfloat   mat_ambient[]     = {0.0, 0.0, 1.0, 1.0};
    GLfloat   mat_flash[]       = {0.0, 0.0, 1.0, 1.0};
    GLfloat   mat_flash_shiny[] = {50.0};
    GLfloat   light_position[]  = {100.0,-200.0,200.0,0.0};
    GLfloat   ambi[]            = {0.1, 0.1, 0.1, 0.1};
    GLfloat   lightZeroColor[]  = {0.9, 0.9, 0.9, 0.1};

    argDrawMode3D();
    argDraw3dCamera( 0, 0 );
    glClearDepth( 1.0 );
```

```
    glClear(GL_DEPTH_BUFFER_BIT);
    glEnable(GL_DEPTH_TEST);
    glDepthFunc(GL_LEQUAL);

    /* load the camera transformation matrix */
    argConvGlpara(patt_trans, gl_para);
    glMatrixMode(GL_MODELVIEW);
    glLoadMatrixd( gl_para );

    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glLightfv(GL_LIGHT0, GL_POSITION, light_position);
    glLightfv(GL_LIGHT0, GL_AMBIENT, ambi);
    glLightfv(GL_LIGHT0, GL_DIFFUSE, lightZeroColor);
    glMaterialfv(GL_FRONT, GL_SPECULAR, mat_flash);
    glMaterialfv(GL_FRONT, GL_SHININESS, mat_flash_shiny);
    glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient);
    glMatrixMode(GL_MODELVIEW);
    glTranslatef( 0.0, 0.0, 25.0 );
    glutSolidCube(30.0);
    glDisable( GL_LIGHTING );

    glDisable( GL_DEPTH_TEST );
}
```

Appendix B
Utility Cable Detection Results

The results produced by the utility cable detection experiment described in Section 3.2 are presented in this appendix, in figures B.1 – B.20. Each page contains two figures; the top figure is the point cloud gathered from one of the configurations, and the bottom figure presents the output of the utility cable detection performed on the point cloud data. In the output images, the primary detected cable is colored red. If a second cable is found in the image, it is colored purple. As can be seen, the detection algorithm consistently finds all utility cables except for those that are nearly horizontal ($\theta \approx 0°$). Examples where a near-horizontal utility cable was not found are seen in Figures B.1, B.2, and B.17.

(a)



(b)

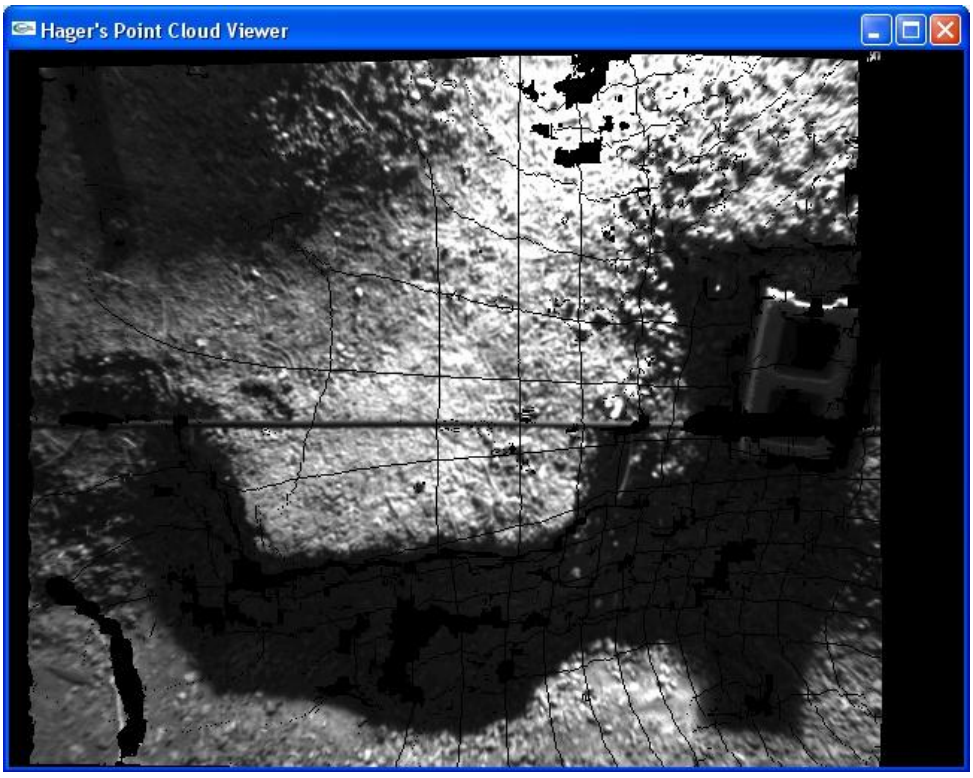
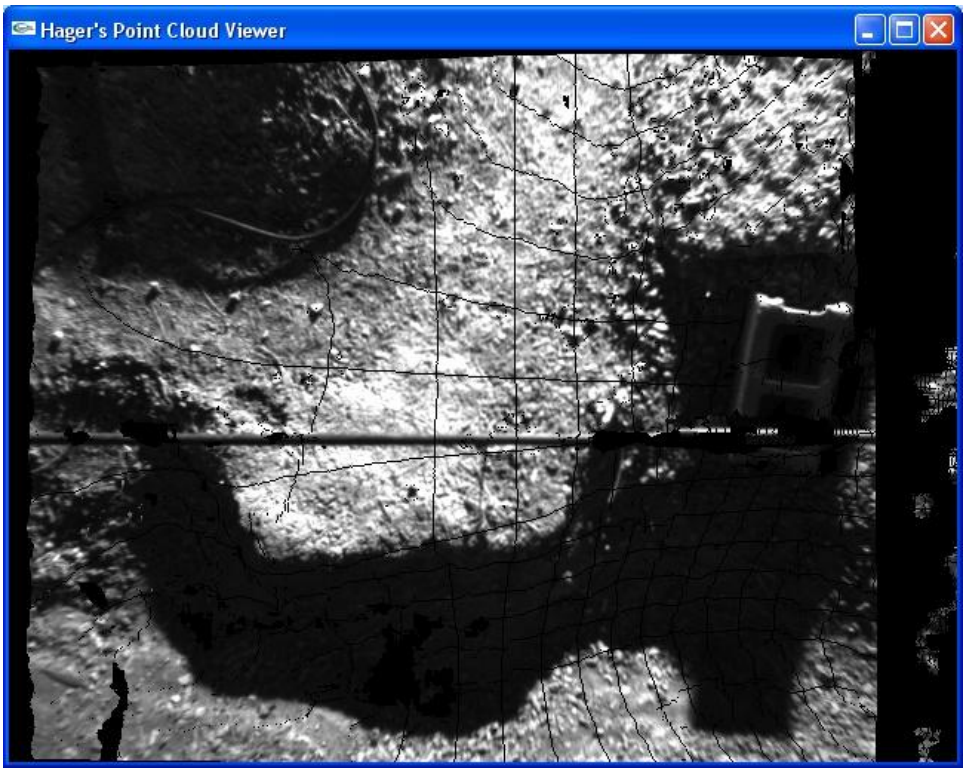


Figure B.1 Scene 0_H1. (a) Input range image. (b) Utility cable detection output.

(a)



(b)

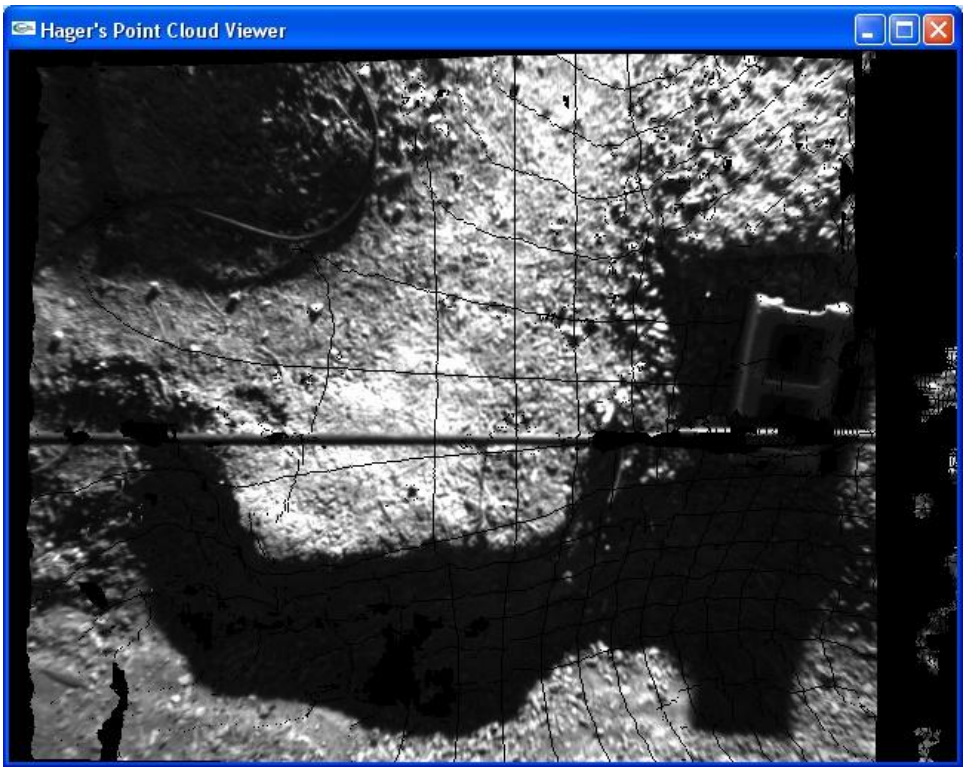


Figure B.2 Scene 0_H2. (a) Input range image. (b) Utility cable detection output.

(a)

(b)

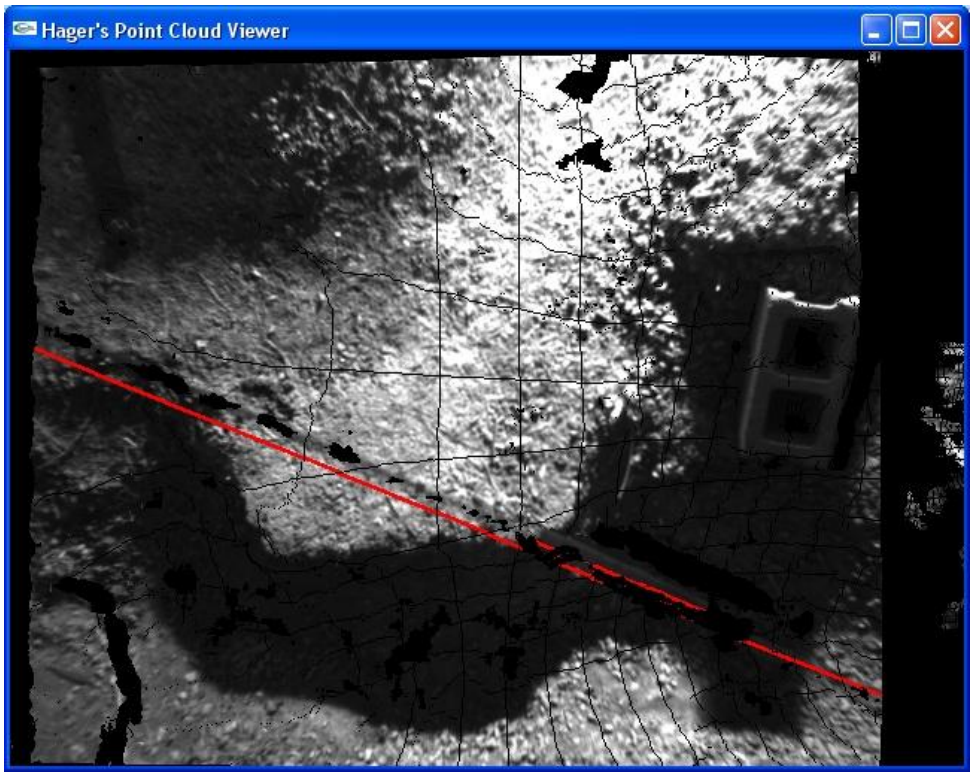


Figure B.3 Scene 30_H1. (a) Input range image. (b) Utility cable detection output.

(a)

(b)

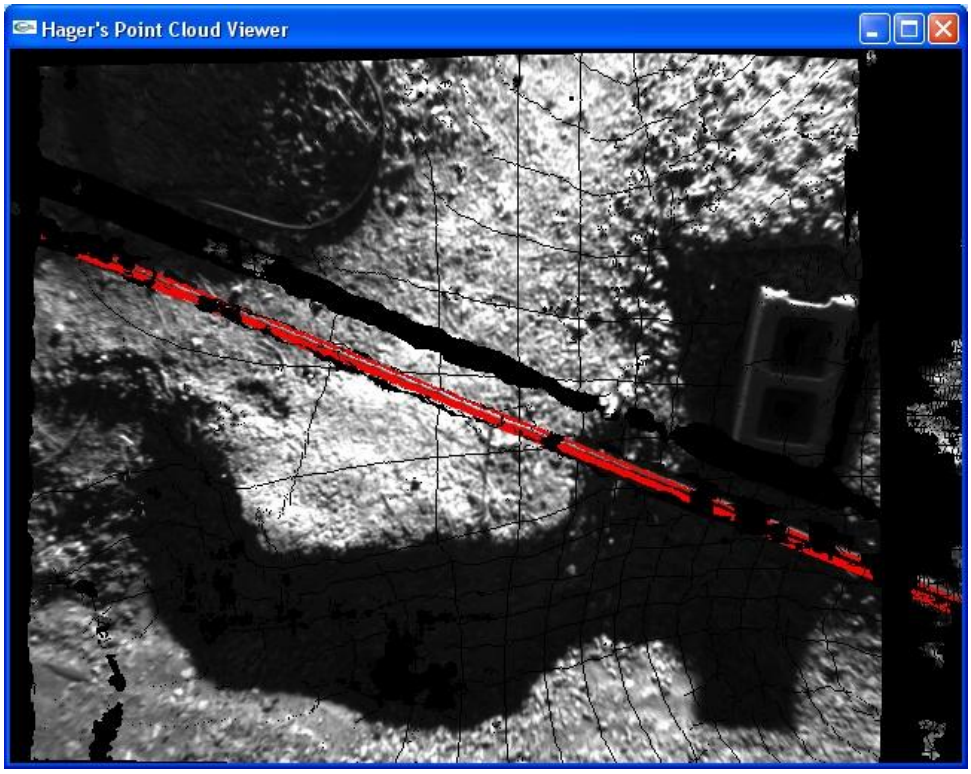


Figure B.4 Scene 30_H2. (a) Input range image. (b) Utility cable detection output.

(a)

(b)

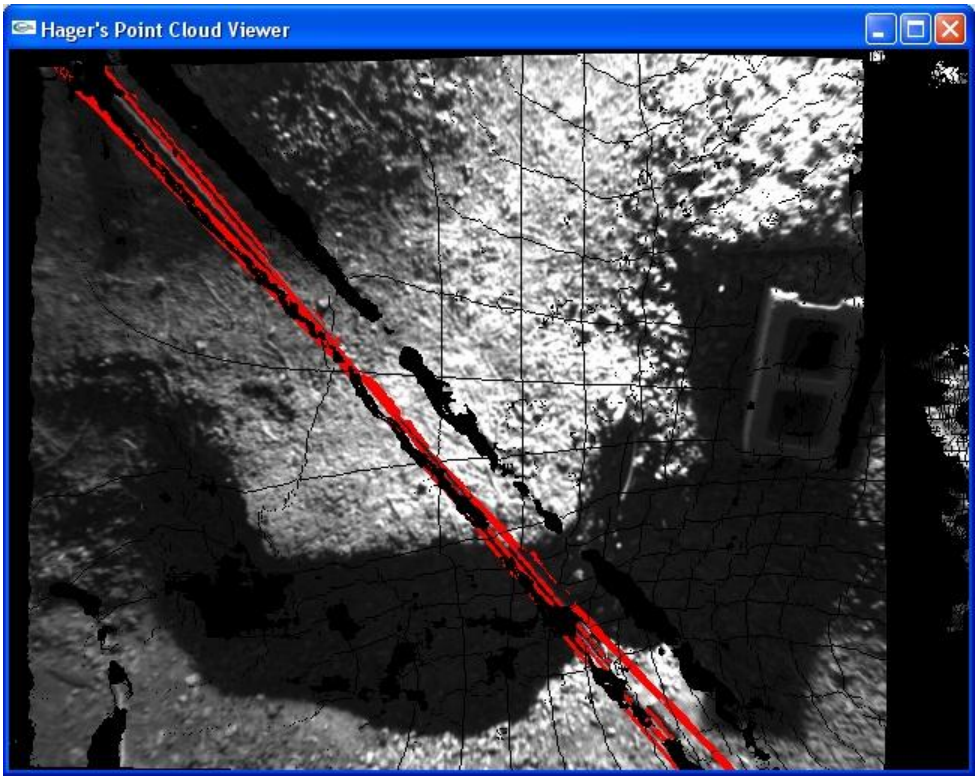
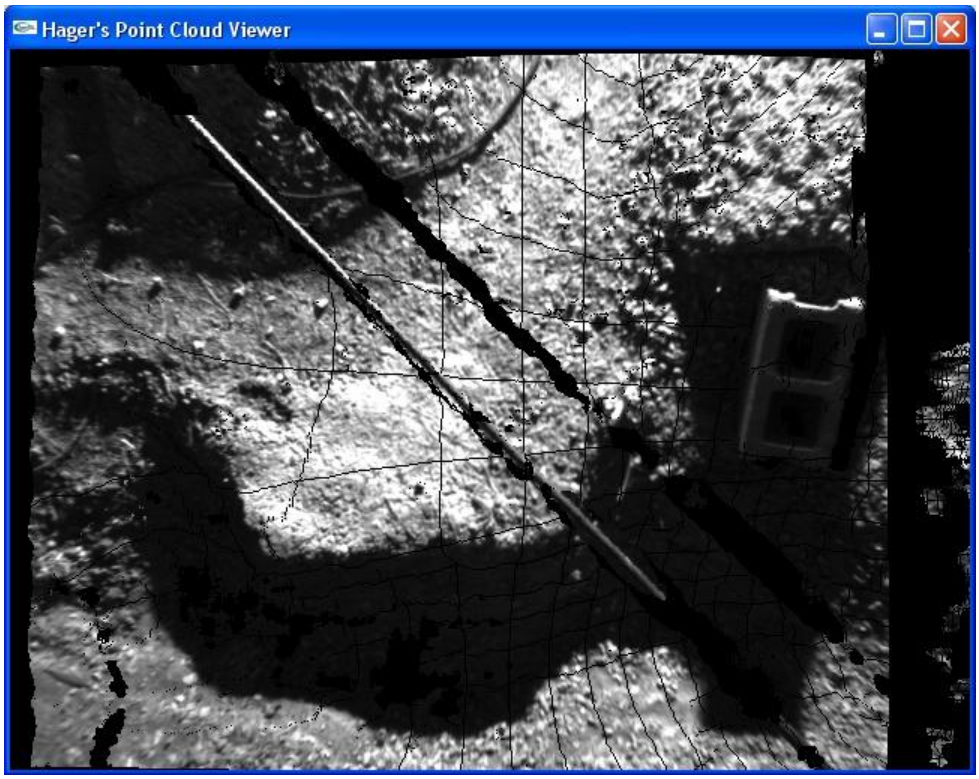


Figure B.5 Scene 45_H1. (a) Input range image. (b) Utility cable detection output.

(a)



(b)

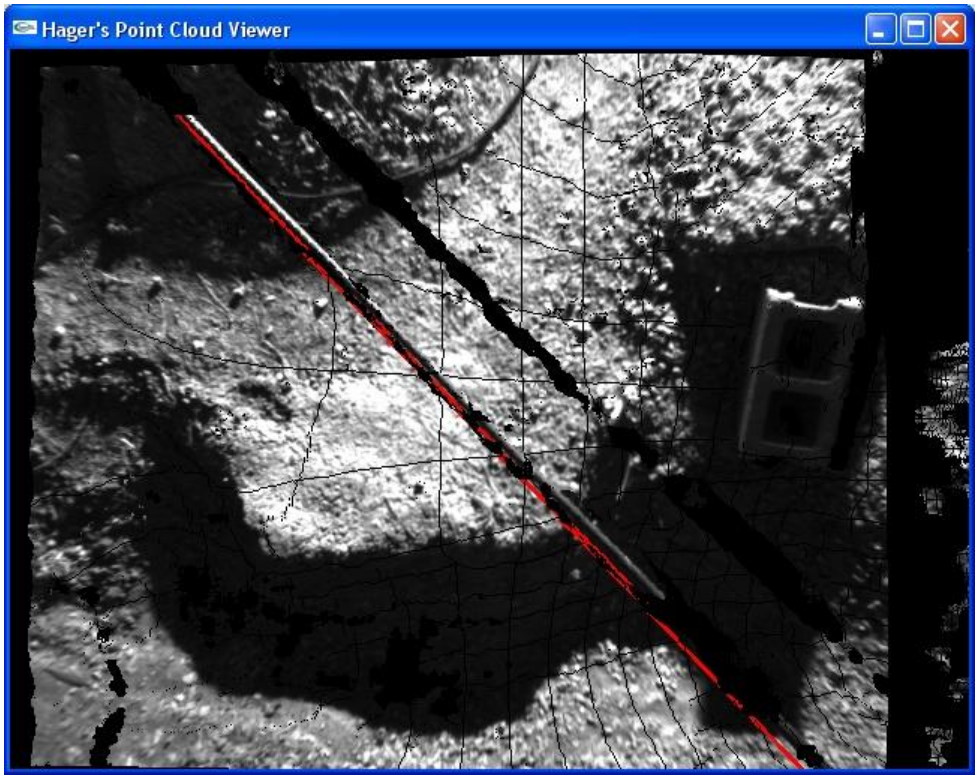


Figure B.6 Scene 45_H2. (a) Input range image. (b) Utility cable detection output.

(a)

(b)

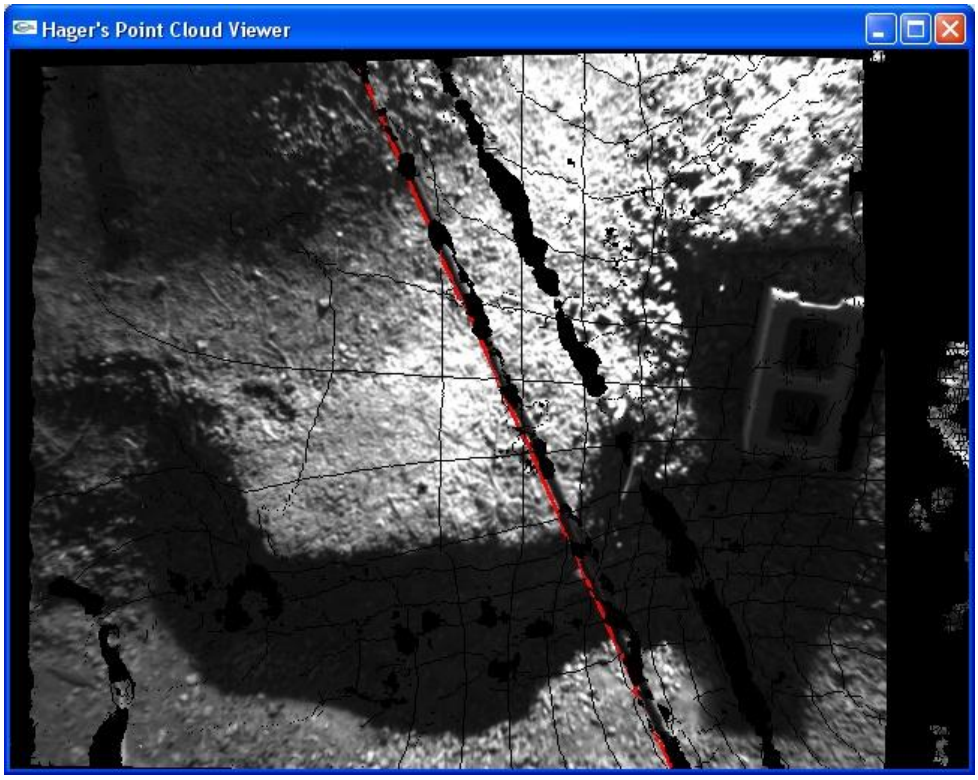
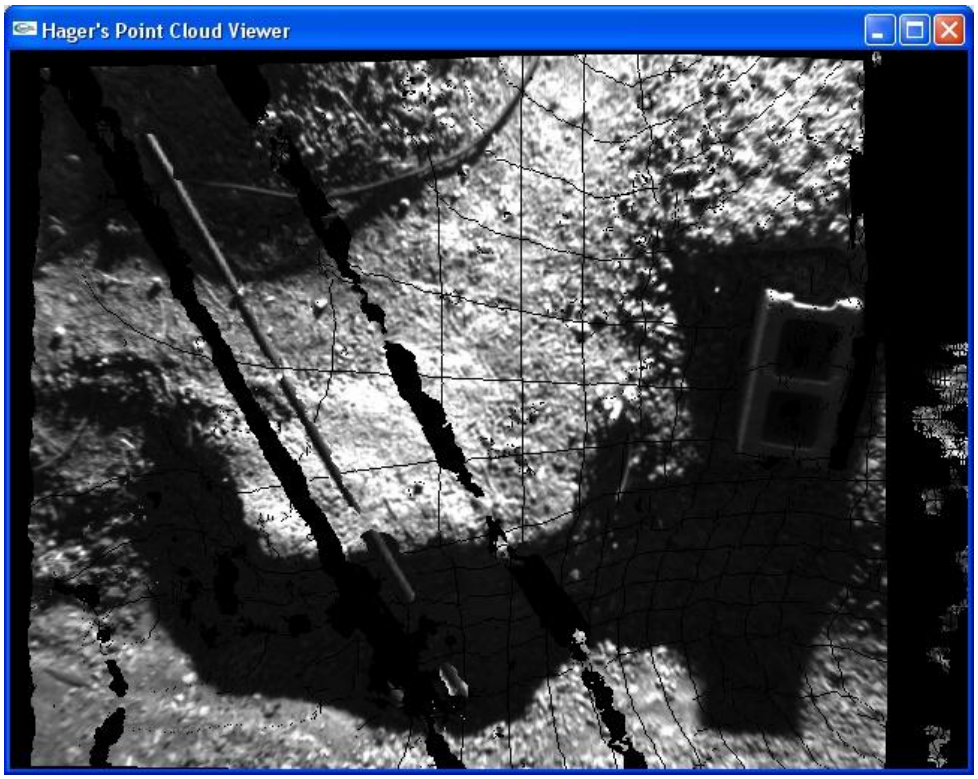


Figure B.7 Scene 60_H1. (a) Input range image. (b) Utility cable detection output.

(a)



(b)

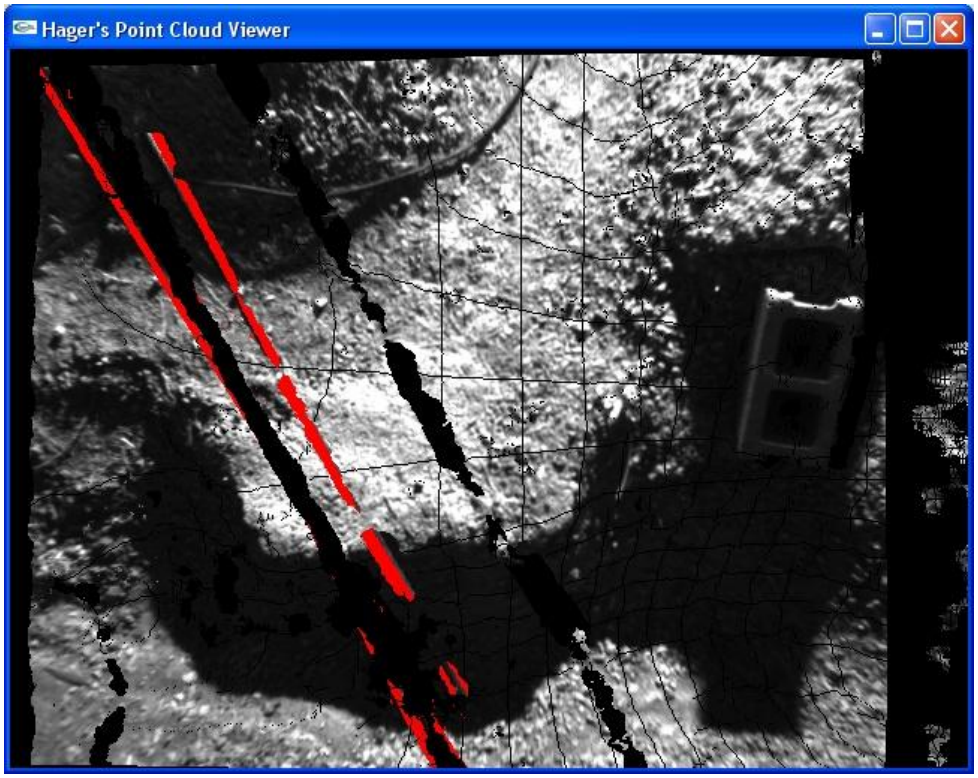
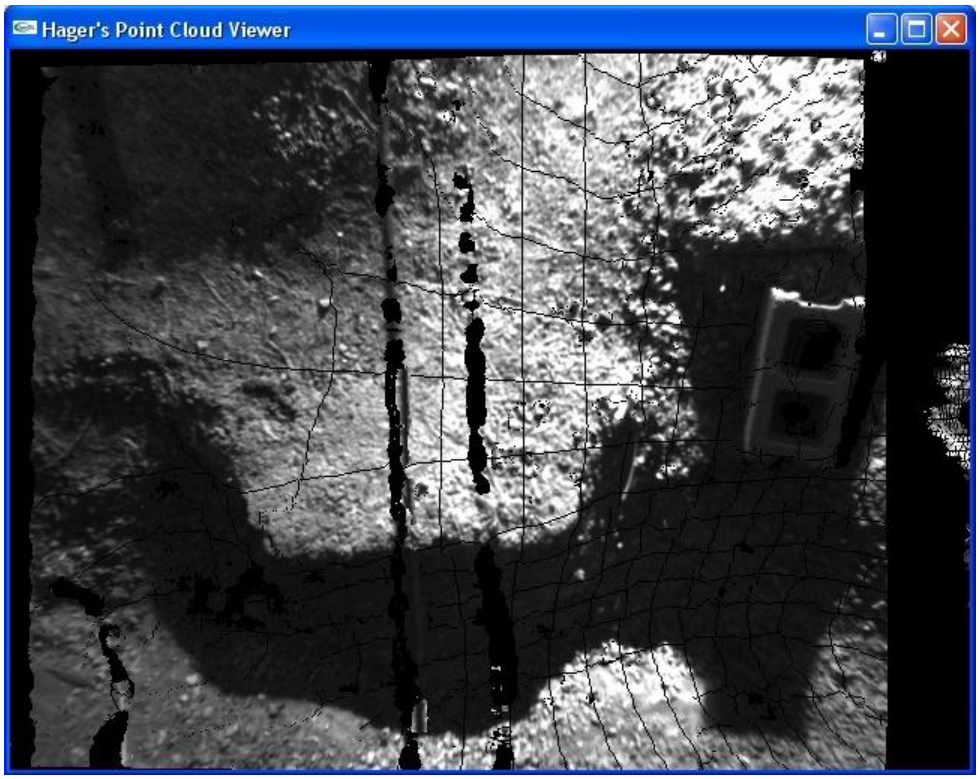


Figure B.8 Scene 60_H2. (a) Input range image. (b) Utility cable detection output.

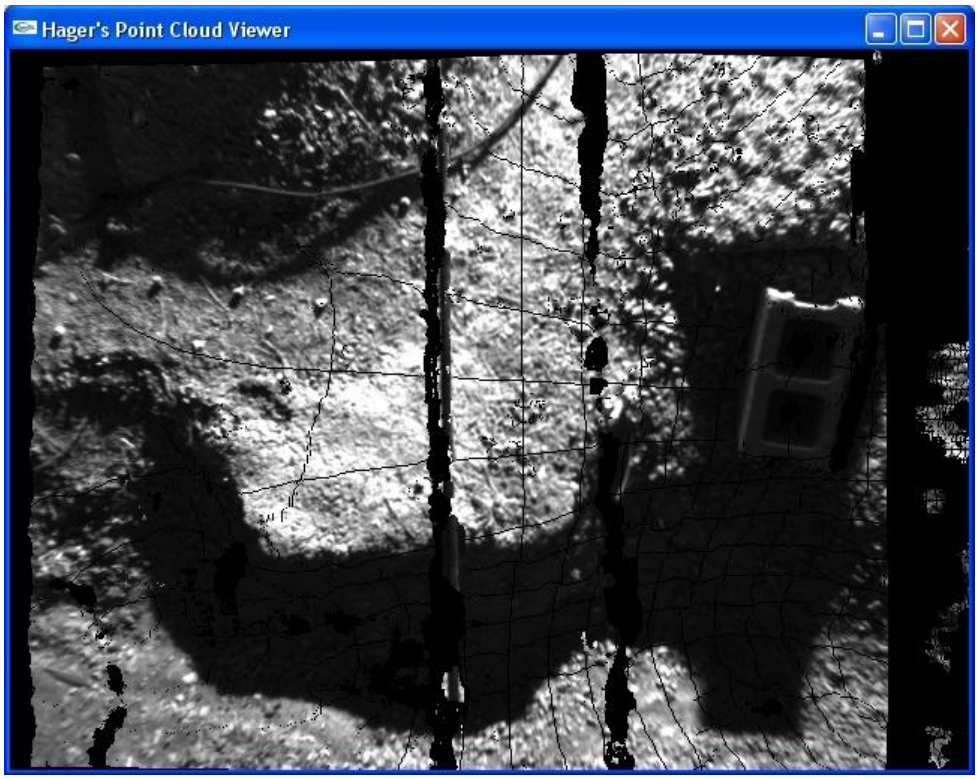
(a)



(b)

Figure B.9 Scene 90_H1. (a) Input range image. (b) Utility cable detection output.

(a)



(b)

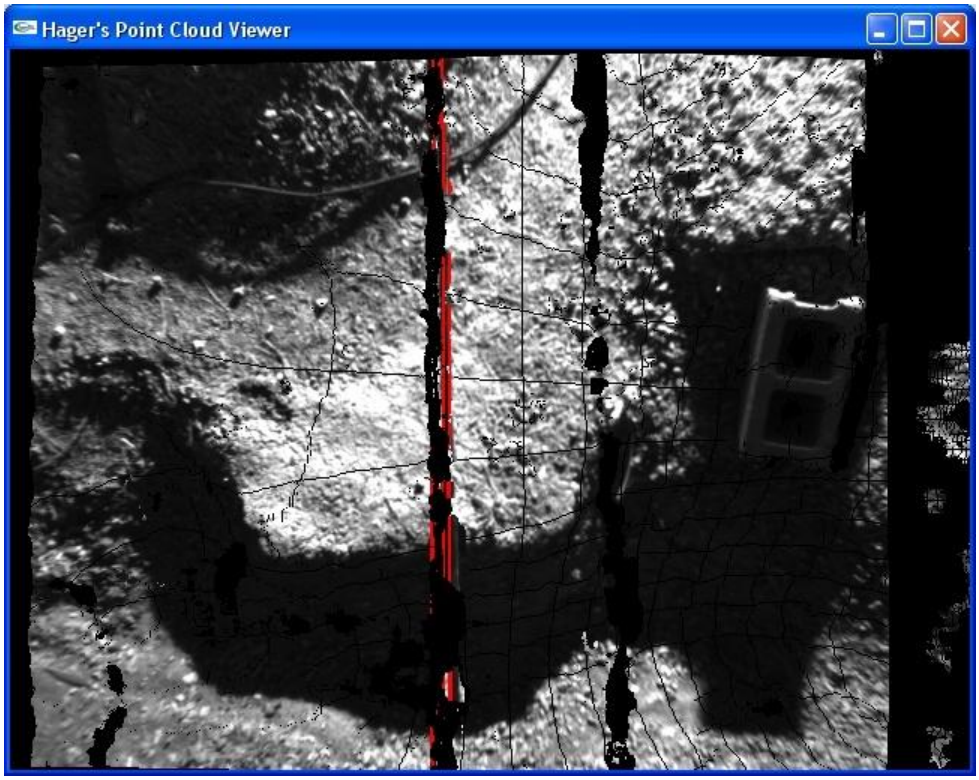
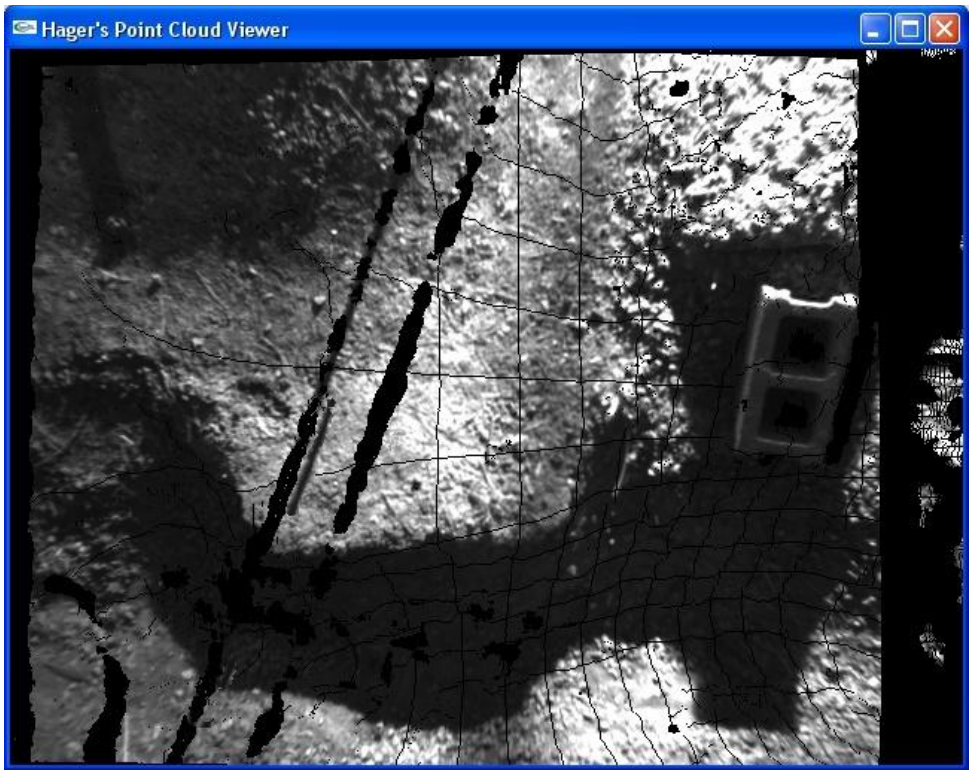


Figure B.10 Scene 90_H2. (a) Input range image. (b) Utility cable detection output.

(a)



(b)

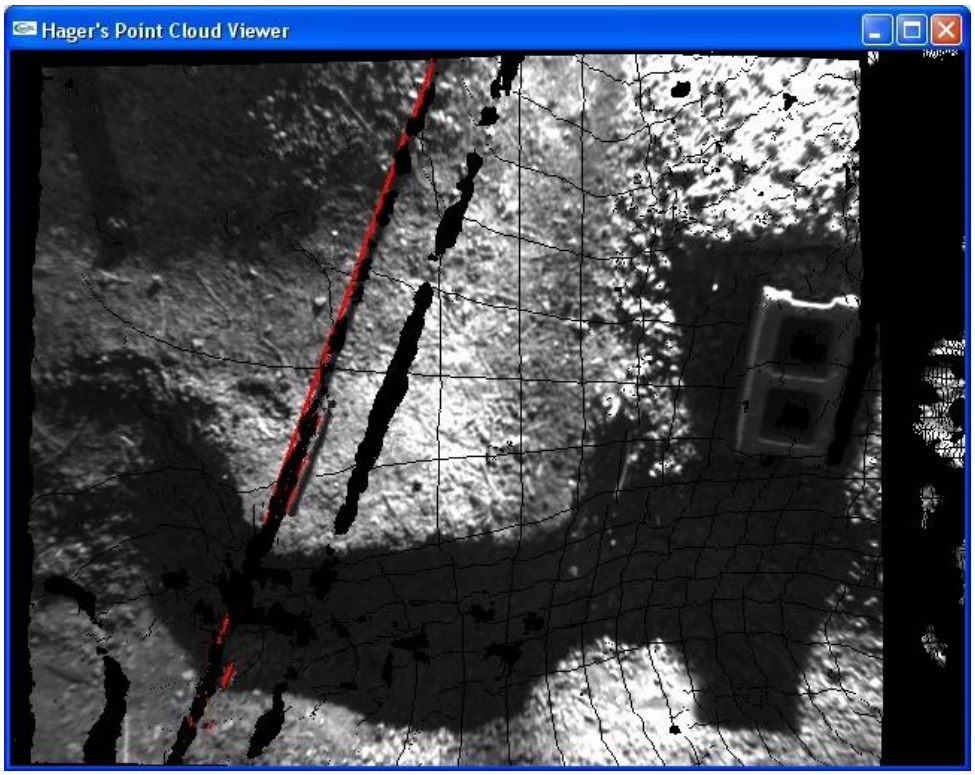
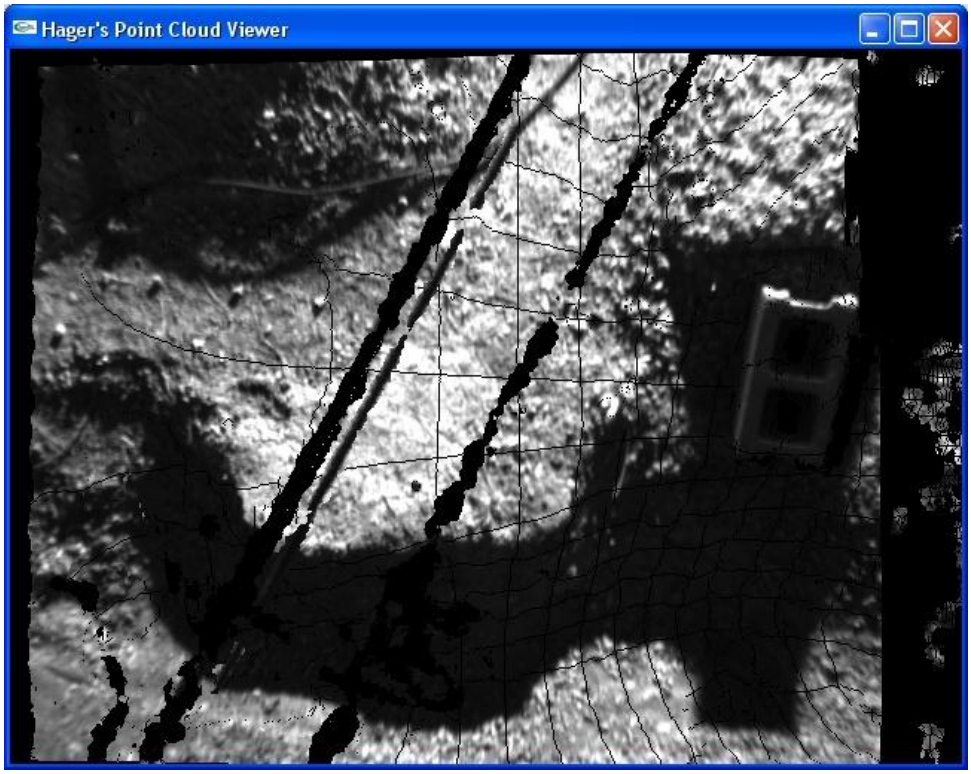


Figure B.11 Scene 120_H1. (a) Input range image. (b) Utility cable detection output.

(a)



(b)

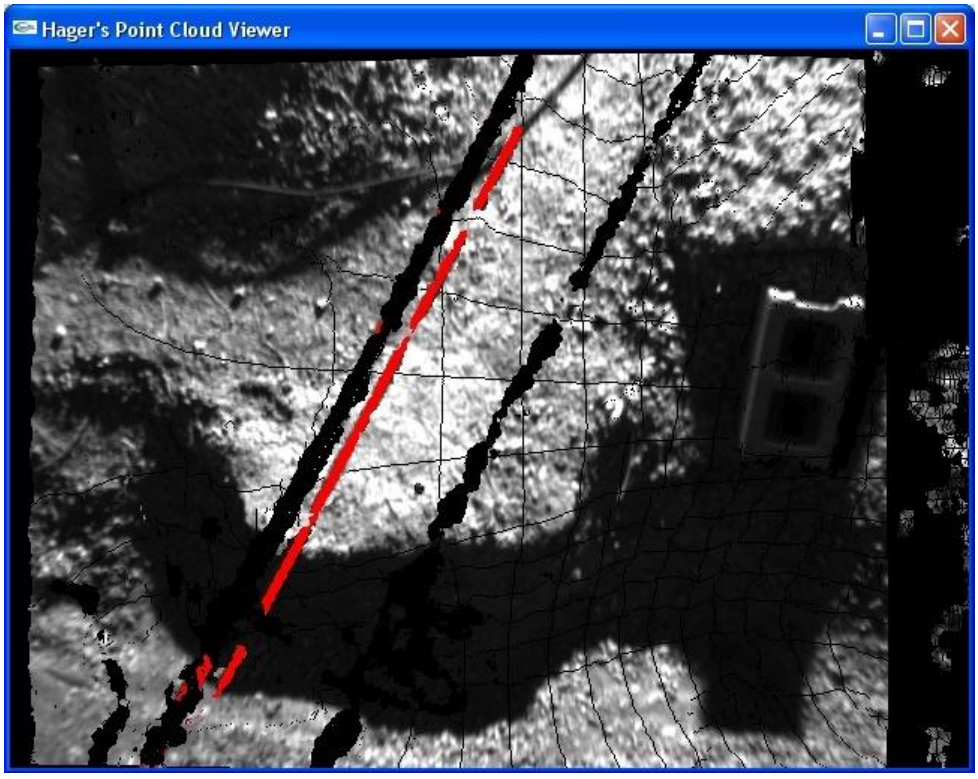
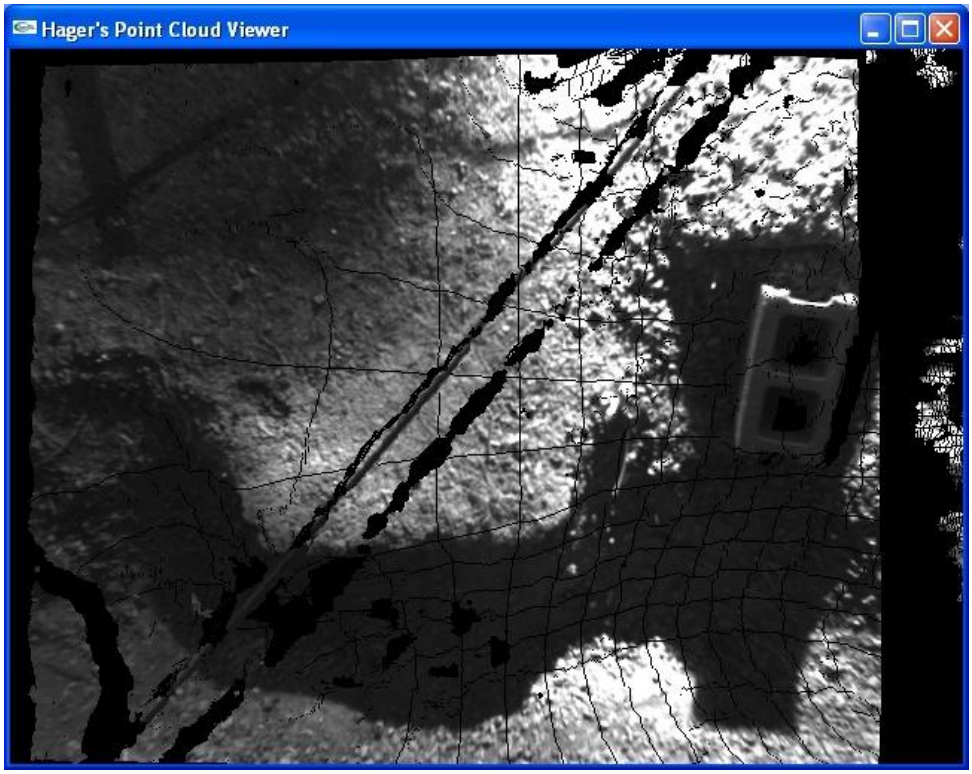


Figure B.12 Scene 120_H2. (a) Input range image. (b) Utility cable detection output.

(a)



(b)

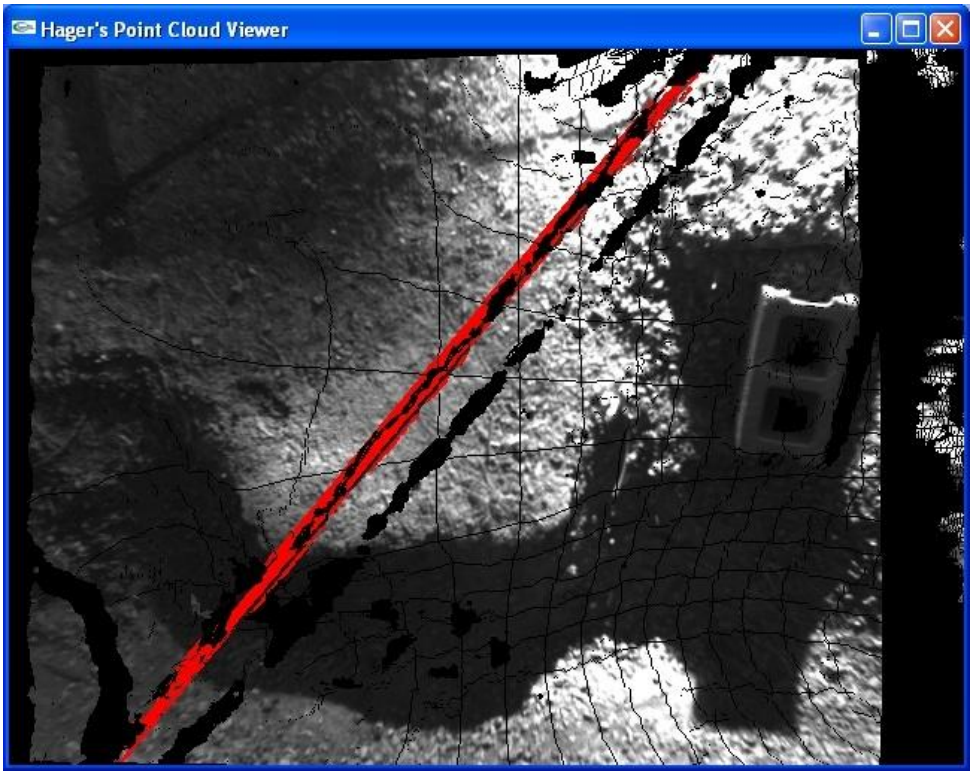


Figure B.13 Scene 135_H1. (a) Input range image. (b) Utility cable detection output.

(a)

(b)

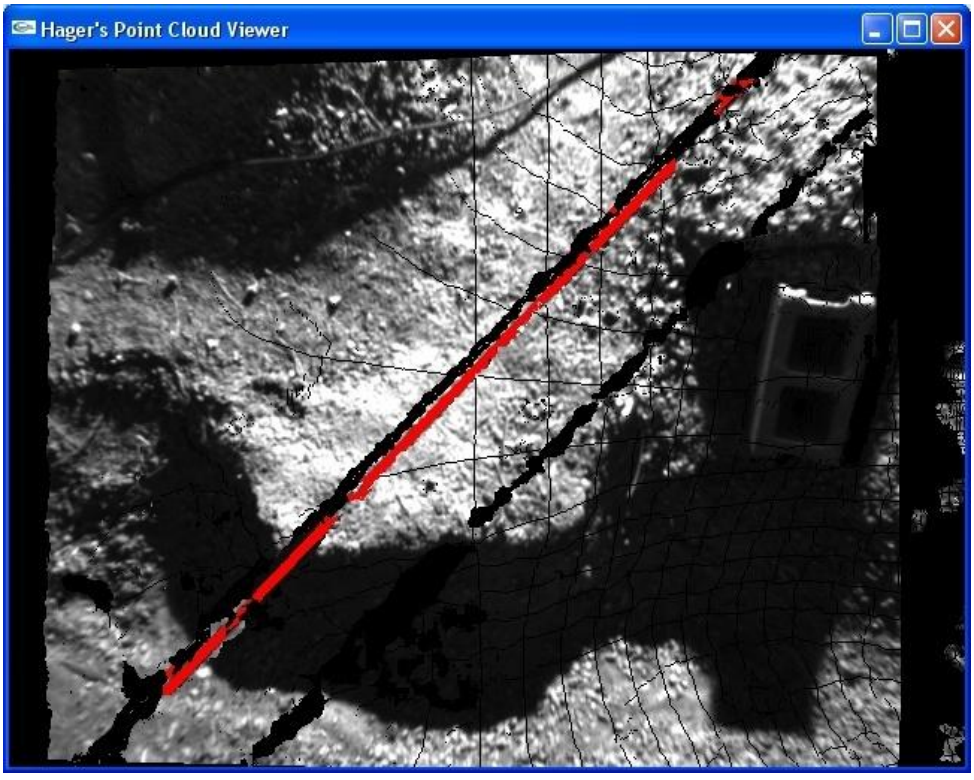


Figure B.14 Scene 135_H2. (a) Input range image. (b) Utility cable detection output.

(a)

(b)

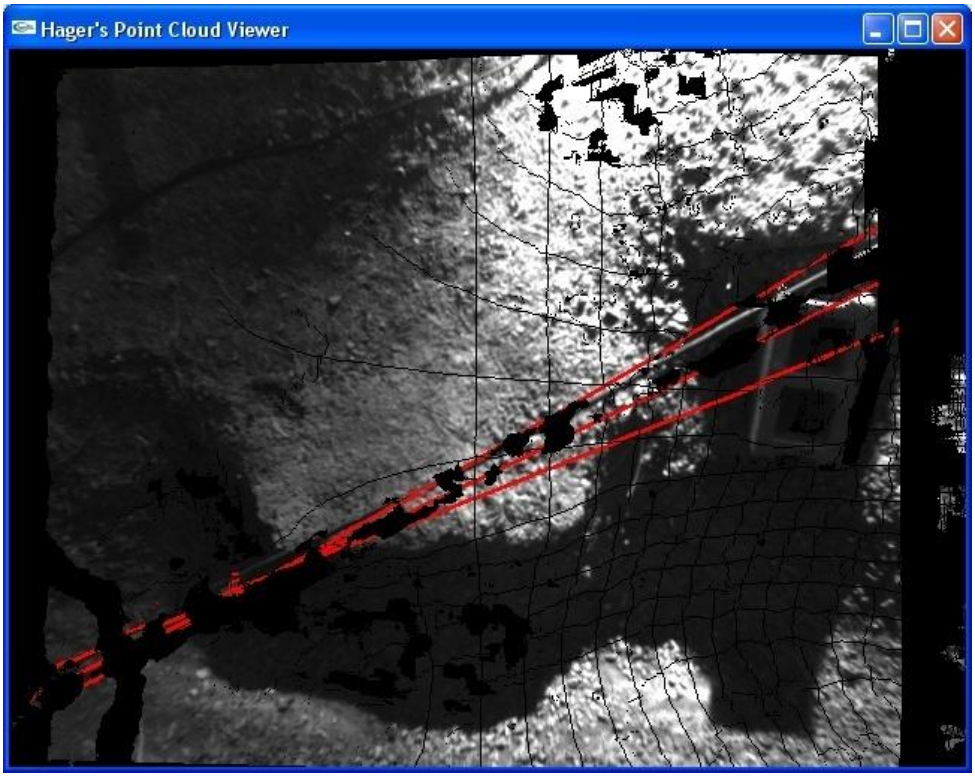


Figure B.15 Scene 150_H1. (a) Input range image. (b) Utility cable detection output.

(a)

(b)

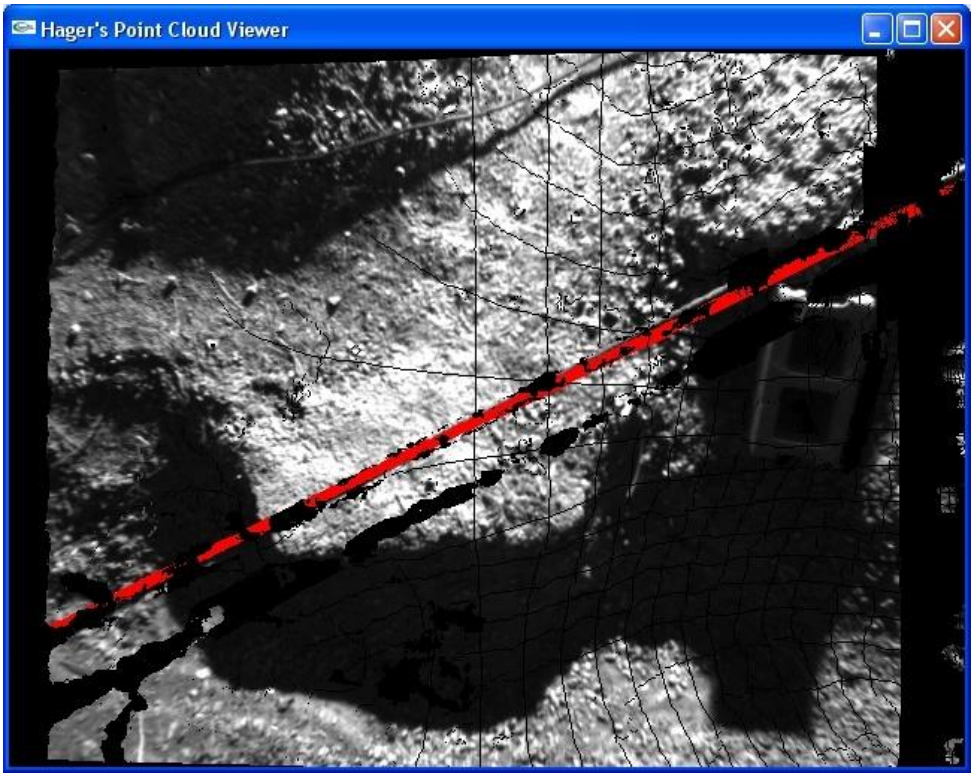
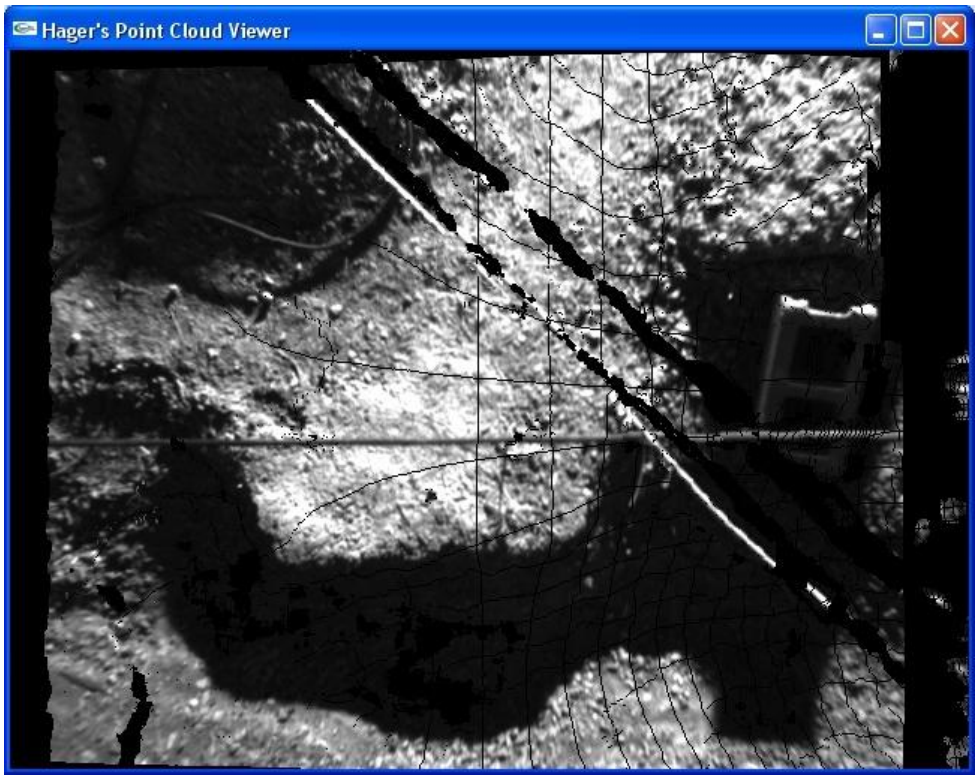


Figure B.16 Scene 150_H2. (a) Input range image. (b) Utility cable detection output.

(a)



(b)

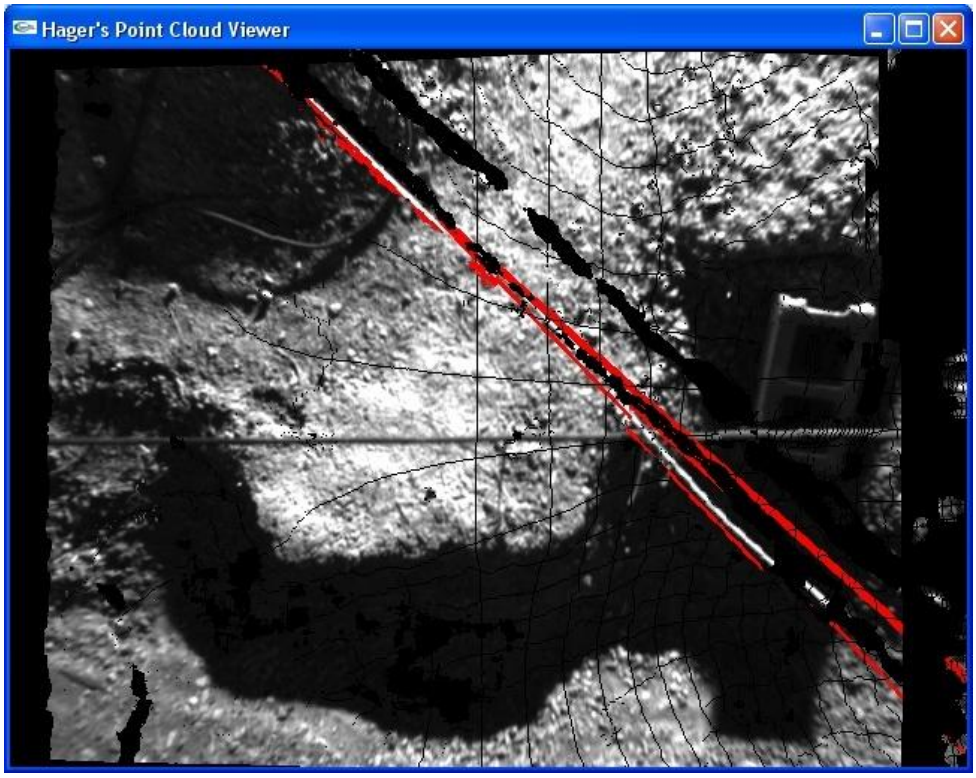


Figure B.17 Scene Cross_45_1. (a) Input range image. (b) Utility cable detection output.

(a)

(b)

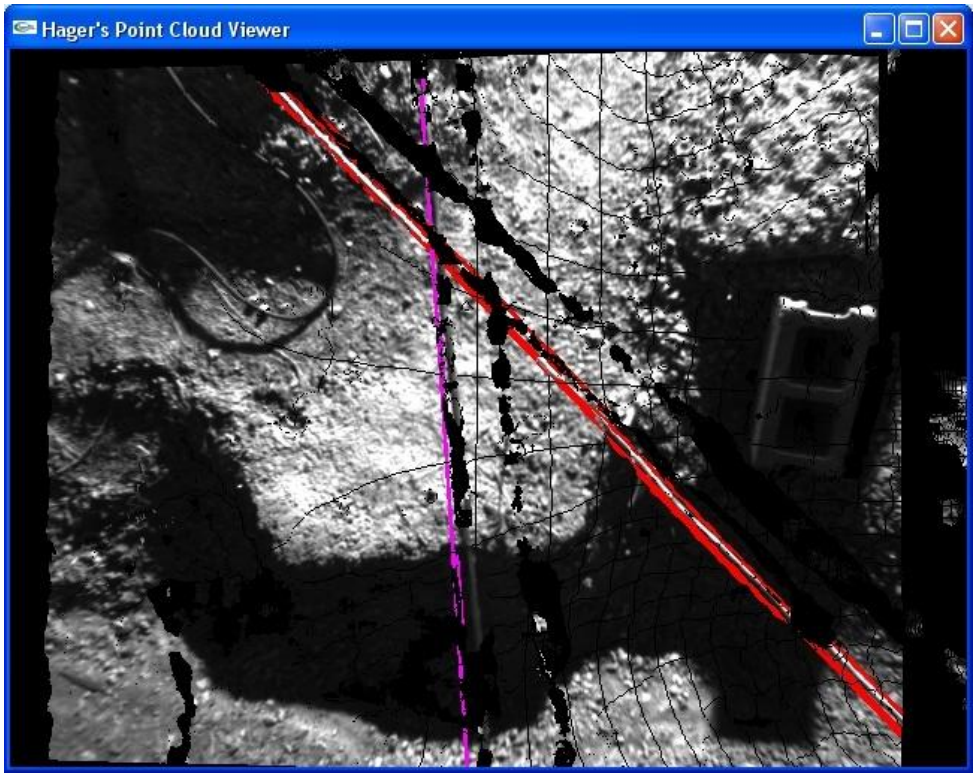


Figure B.18 Scene Cross_45_2. (a) Input range image. (b) Utility cable detection output.

(a)

(b)

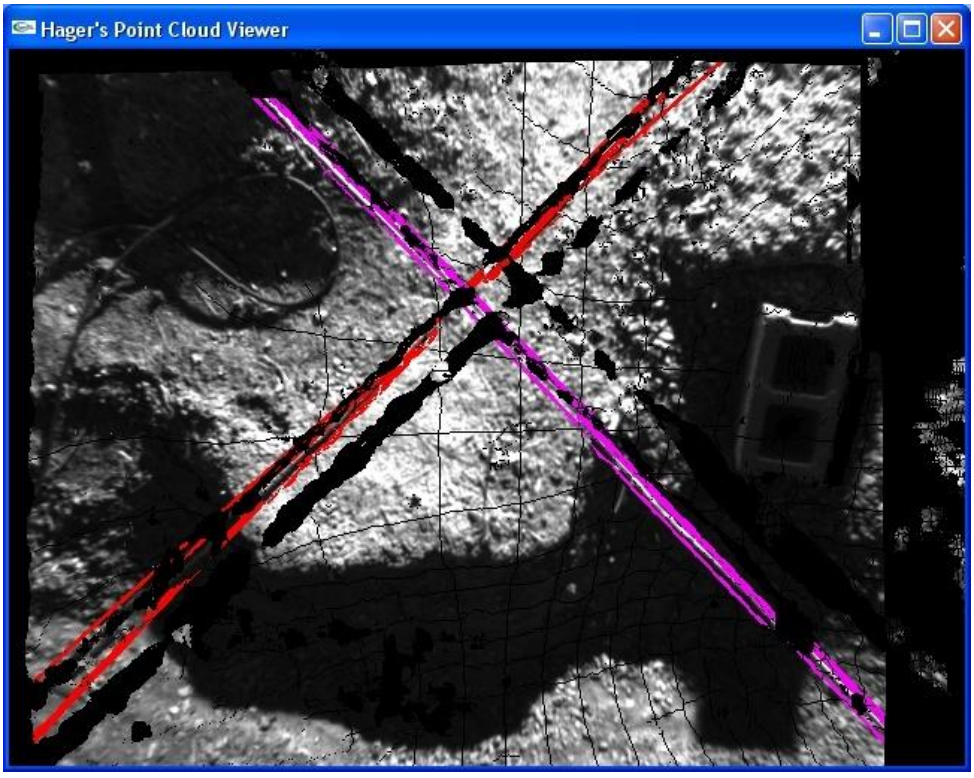
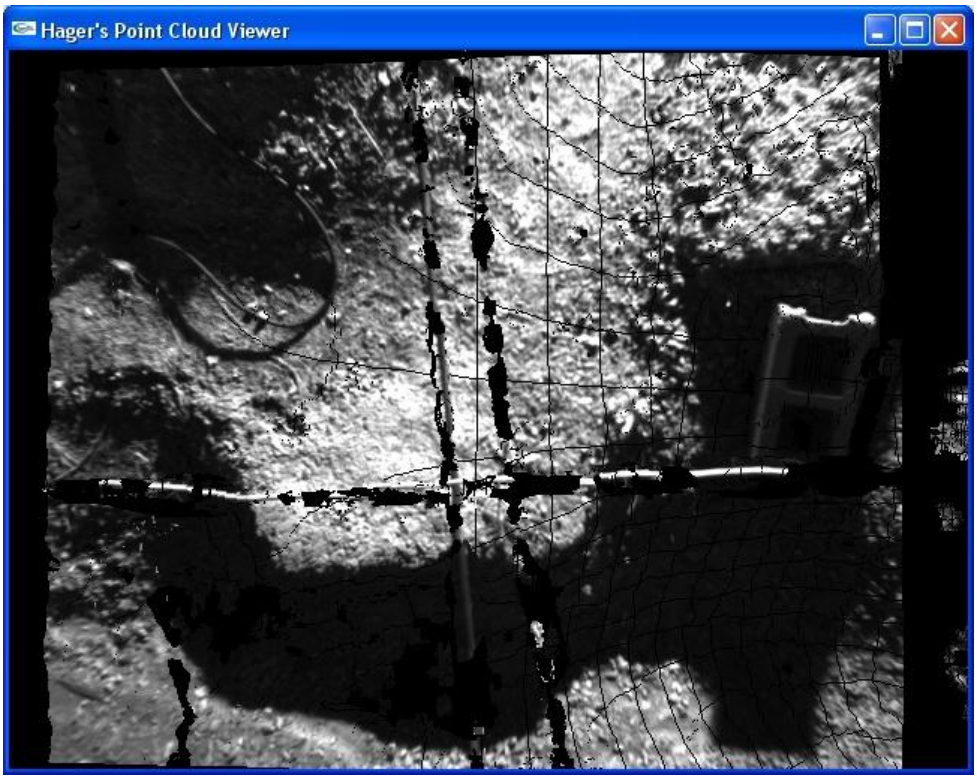


Figure B.19 Scene Cross_90_1. (a) Input range image. (b) Utility cable detection output.

(a)



(b)

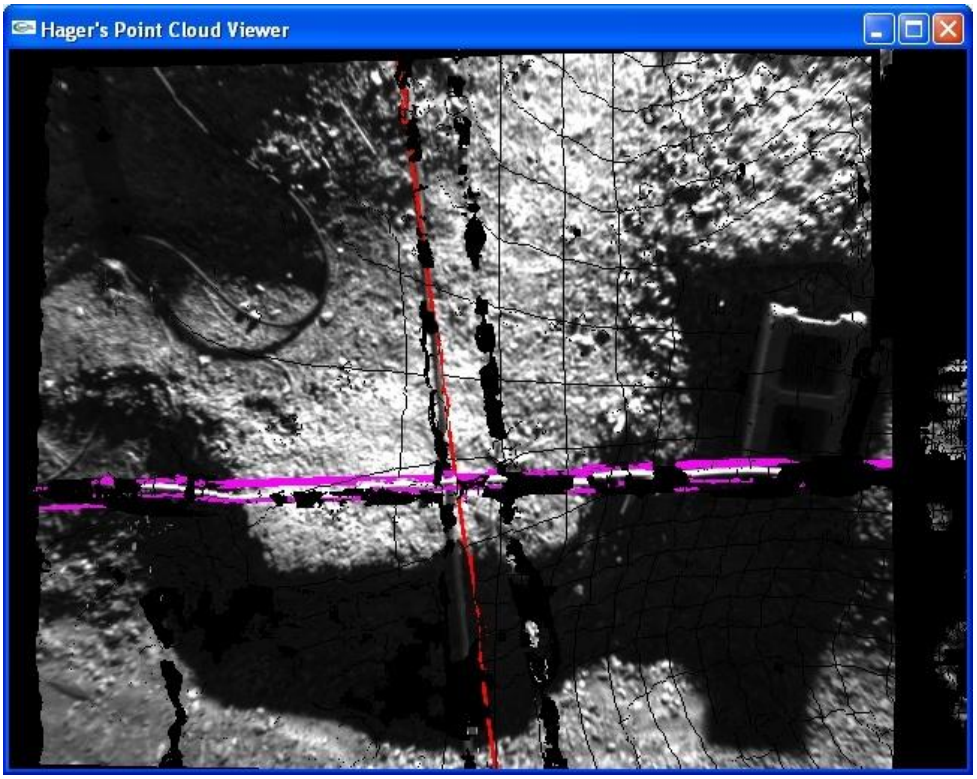


Figure B.20 Scene Cross_90_2. (a) Input range image. (b) Utility cable detection output.