AUTOMATED LANDING SITE EVALUATION FOR SEMI-AUTONOMOUS UNMANNED AERIAL VEHICLES

Dylan Klomparens

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 Kochersberger

Paul Plassmann

Robert Broadwater

August 20, 2008

Blacksburg, VA

Keywords: Stereo Vision, Landing Site Evaluation, UAV, Supervisory Control, Obstacle Detection, User Performance Evaluation

© 2008, Dylan Klomparens

AUTOMATED LANDING SITE EVALUATION FOR SEMI-AUTONOMOUS UNMANNED AERIAL VEHICLES

Dylan Klomparens

ABSTRACT

A system is described for identifying obstacle-free landing sites for a vertical-takeoff-and-landing (VTOL) semi-autonomous unmanned aerial vehicle (UAV) from point cloud data obtained from a stereo vision system. A relatively inexpensive, commercially available stereo vision camera system was selected for this study. A "point cloud viewer" computer program was written to analyze 3D point cloud data obtained from 2D images transmitted from the UAV to a remote ground station. The program divides the point cloud data into sections, identifies the best-fit plane through the data for each section, and performs an independent analysis on each section to assess the feasibility of landing in that area. The program also rapidly presents the 3D information and analysis to the remote mission supervisor and facilitates quick, reliable decisions about where to safely land the UAV. The features of the program and the methods used to identify suitable landing sites are presented in this thesis. Also presented are the results of a user study that compares the abilities of humans and computer-supported point cloud analysis in certain aspects of landing site assessment. The study demonstrates that the computer-supported evaluation of potential landing sites provides an immense benefit to the UAV supervisor.

TABLE OF CONTENTS

Abstract ii

Table of Contents iii

List of Figures vi

List of Tables ix

Grant Information x

Dedication xi

Acknowledgements xii

I. Introduction 1

A. Background 1

B. Thesis Organization 3

II. Stereopsis Concepts and Applications 5

A. Stereopsis 5

1) Camera Model and Calibration 5

2) Multiple View Geometry and Rectification 10

3) Pixel Correspondence 12

B. Description of Stereo Vision System Options 13

C. List of Contributions 14

III. Description of Bumblebee Parameters, Capabilities, and Image Quality 16

IV. Methods Developed for Identification of Suitable Landing Sites 21

A. Division of the Point Cloud 21

B. Best-fit Plane Generation 21

C. Determining Suitability of a Landing Site 23

D. Example Scene 25
V. Point Cloud Viewer 29
A. Features 29
1) 2D and 3D Representations of the Scene 29
2) Multi-dimensional Representation of the Scene 30
3) Input and Output of a Data Set 31
4) Selection 32
5) Navigation 33
6) Mouse and Keyboard Input Commands 33
B. Code Structure and Code Interface 34
1) The Cloud Structure 35
2) The BumblebeeCamera Class 41
3) Secondary “Container” Structures 41
4) Selection 42
5) Texturing 42
C. Memory Management Considerations 43
D. Informal Speed Test 43
VI. User study 45
A. Experiment Design 45
B. Results 48
VII. Conclusions and Future Work 53
VIII. Works Cited 54
Appendix A: Point Cloud Viewer Code 55
A. Driver.cpp 55
B. Bumblebee.h 71

C. Bumblebee.cpp 72
D. Cloud.h 76
E. Cloud.cpp 79
F. Render.h 94
G. Render.cpp 96
H. Config.h 106
Appendix B: User Study Approval and Questionnaire 107

LIST OF FIGURES

Figure 1: A pinhole camera. 6
Figure 2: Axis setup for a pinhole camera and relevant planes. 7
Figure 3: Camera frame offset. The basis for the design of this image came from [10]. 9
Figure 4: Epipolar geometry. The basis for the design of this image came from [10]. 11
Figure 5: The final geometric setup of rectified images. The basis for the design of this image came from [10]. 11
Figure 6: Unmodified left image from a stereoscopic camera system. 12
Figure 7: Unmodified right image from a stereoscopic camera system. 12
Figure 8: Left rectified image from a stereoscopic camera system. 12
Figure 9: Right rectified image from a stereoscopic camera system. 12
Figure 10: The Bumblebee stereo vision camera system from Point Grey Research, Inc. 14
Figure 11: An image that would perform very well with many stereo matching techniques. 16
Figure 12: An image that would perform very poorly with many stereo matching techniques. 16
Figure 13: A scene constructed to demonstrate the effects of the point cloud optimization techniques provided by the Bumblebee software. 17
Figure 14: The example scene with no point cloud optimization enabled. 18
Figure 15: The example scene with point cloud optimization enabled. 18
Figure 16: The example scene with no point cloud optimization enabled. The noisy points that were mostly removed by the image enhancement techniques of the Bumblebee software are highlighted in green. 19
Figure 17: A sample best-fit plane and its corresponding data points. 22
Figure 18: Distance *a* depicts the vertical distance from a selected point to its corresponding best-fit plane, while distance *b* depicts the shortest distance from the point to the plane. 22
Figure 19: To help determine the plausibility of a landing site, the normal vector of a given plane can be compared to the vertical direction. 24
Figure 20: To help determine the plausibility of a landing site, the normal vector of a given plane can be compared to the normal vector of neighboring planes. 24
Figure 21: There may be up to eight adjacent planes considered in a proximity analysis. 24

Figure 22: An example scene obtained using the Bumblebee stereo vision camera system. This image is from the Bumblebee's right camera. 26
Figure 23: The point cloud of the example scene, obtained using images from the Bumblebee stereo vision camera system... 26
Figure 24: The example scene's point cloud with overlaid best-fit planes................................. 27
Figure 25: The example scene's 2D representation with acceptable landing sites indicated by blue boxes. 27
Figure 26: A simulated descent, at a height of 2.44 meters. 28
Figure 27: A simulated descent, at a height of 2.44 meters with acceptable landing sites overlaid. 28
Figure 28: A simulated descent, at a height of 2.11 meters. 28
Figure 29: A simulated descent, at a height of 2.11 meters with acceptable landing sites overlaid. 28
Figure 30: A simulated descent, at a height of 1.42 meters. 28
Figure 31: A simulated descent, at a height of 1.42 meters with acceptable landing sites overlaid. 28
Figure 32: An abstract example that shows the mapping between a 2D pixel and a 3D point..... 30
Figure 33: An example scene that depicts the mapping between the 2D and 3D representation of the data. 31
Figure 34: The "open file" dialog from the point cloud viewer. 32
Figure 35: An example of selection and detailed plane information in the point cloud viewer... 33
Figure 36: Class diagram of the point cloud viewer program. 36
Figure 37: The flow of execution for the point cloud viewer program. 38
Figure 38: The flow of execution for the analysis of a point cloud. 39
Figure 39: A point cloud with best-fit planes overlaid. The planes are colored according to their slope (in comparison to the vertical direction). Flat planes appear lighter, while steep planes appear darker. 40
Figure 40: A point cloud with best-fit planes overlaid. The planes are colored according to their slope (in comparison to the vertical direction). In this particular case, a steepness threshold of 25 degrees was chosen. All planes that have a slope of less than or equal to 25 degrees are colored green, while all other (steeper) planes are colored red. 40

Figure 41: Wood pallet tilted at 00.00 degrees. Denoted as image B throughout the experiment. 47
Figure 42: Wood pallet tilted at 04.10 degrees. Denoted as image E throughout the experiment. 47
Figure 43: Wood pallet tilted at 13.07 degrees. Denoted as image F throughout the experiment.47
Figure 44: Wood pallet tilted at 15.89 degrees. Denoted as image A throughout the experiment. 47
Figure 45: Wood pallet tilted at 21.29 degrees. Denoted as image D throughout the experiment. 47
Figure 46: Wood pallet tilted at 26.52 degrees. Denoted as image C throughout the experiment. 47
Figure 47: A boxplot which shows the divergence of angle estimations from the true angle. The box encloses the 25th to 75th percentile and is known as the inner quartile range. The whiskers enclose the 5th to 95th percentile. The “x” marks represent the minimum and maximum deviations, and the dot inside the inner quartile range represents the median............................. 51

LIST OF TABLES

Table 1: Summary of input commands for the point cloud viewer. 33
Table 2: Results of an informal speed test of the point cloud viewer. 44
Table 3: Results of an informal speed test of the point cloud viewer. 44
Table 4: The label of each image used in the user study and the corresponding orientation of the woodn pallet. 48
Table 5: The results of the user study. All angles are in degrees. 49
Table 6: The divergence of user responses from the true angle. All angles are in degrees. 50
Table 7: The point cloud viewer's estimations of the angles presented in the user study. All angles are in degrees. 52

GRANT INFORMATION

The authors wish to acknowledge the support of Pacific Northwest National Laboratories in funding this research, under the guidance of the Unmanned Systems and NBC Instrumentation Electronics and Measurement Systems Group.

DEDICATION

To Claire, Steve, Sean, and Robin for

their continual love and support.

ACKNOWLEDGEMENTS

I would like to thank my advisors, Lynn Abbott and Kevin Kochersberger, for their guidance in what has been an extremely interesting project. They've both provided me with incredible guidance and inspiration.

I'd also like to thank my colleagues at the Unmanned Systems Lab: Adam Sharkasi, James May, and John Bird. Even with their own busy schedules, they would always find time to lend a helping hand – and the research progressed as a result of their kindness. And thank you to Andrew Culhane for his thoughts and background knowledge about running a user study.

I'm also extremely thankful to my roommate, David Class, for our back-and-forth of ideas about each other's thesis topics. David often provided a unique and useful perspective on problems that (at the time) seemed insurmountable.

My good friend Samantha Stolarz also played a pivotal role in this project by initially directing me to the Unmanned Systems Lab.

And last, but certainly not least, I would like to thank my family for their continual love and support.

I. INTRODUCTION

This thesis examines the problem of semi-automated landing of vertical-takeoff-and-landing (VTOL) unmanned aerial vehicles (UAV). The research was originally intended for use in nuclear forensics, but has other potential applications. In the case of a nuclear incident, the site needs to be examined for information. A typical first step in this mission is constructing a new map of the terrain using a large scale imaging system. The implementation of such a large scale system is beyond the scope of this thesis. The next step is selecting one or more regions of interest and sending a UAV to collect detailed forensics information. The UAV may carry a tethered sub-vehicle to perform the forensics work. In this scenario, it requires knowledge of the ground terrain so that a successful ground sampling operation can occur. Additionally, it may be necessary to land the VTOL which also requires an accurate assessment of the landing site. The proposed design requires the UAV to transmit its navigation input to a remote base of operations. The navigation input would then be analyzed to identify potential landing sites and allows management by consent/exception for autonomous landing capability [1].

A. Background

The main focus of this thesis is UAV navigation using stereo vision. The specific challenge was to develop a mathematical approach for identifying acceptable landing sites for a VTOL UAV and tethered sub-vehicle. This was accomplished by creating a computer program to analyze visual data collected at the scene and rapidly identify potential landing sites in a quantitative manner.

The research problem originated in the Unmanned Systems Lab of Virginia Tech's Mechanical Engineering Department. The general research objective was to improve navigation of the UAV using the Bumblebee stereo vision system from Point Grey Research, in Vancouver, British Columbia, Canada. The technical requirement to solve the problem was a proficiency in computer engineering. This led to a partnership between the Mechanical Engineering and Electrical & Computer Engineering departments at Virginia Tech. The Bumblebee system produces a low quality video signal at night and in low-visibility conditions, so improving the video signal or the analysis performed on the signal was a novel research venture. Preliminary work explored several avenues including colored structured lighting techniques [2], image

segmentation [3], and artificial illumination with visible spectrum light. The general research objective was eventually narrowed to a consideration of landing site selection for high-visibility conditions.

Other researchers have considered the use of a VTOL UAV to search for a predetermined target [4]. In this previous study, the UAV searched for and found a target that was known *a priori* to the navigation software. The study was not concerned with obstacle detection when landing the UAV, or with the suitability of a landing site. The research for this thesis addresses those problems. A major consideration with obstacle detection and landing site evaluation is the accuracy of the visual information. Fusing **Li**ght **D**etection **a**nd **R**anging (LIDAR) information and stereo range data was investigated as a potential solution [5], but was not pursued here because of limited feasibility. The LIDAR system used in the preliminary research required between 20 to 40 seconds to capture information about a single scene, and this delay was unreasonable given the goals and constraints of the objective.

The Bumblebee stereo vision system's software outputs a set of 3D points when it captures visual information [6]. This set of points is called a point cloud. One particularly interesting avenue of research which applied to landing site selection was the creation of a mesh from the point cloud to reconstruct barriers in a scene. Several techniques for constructing a mesh were researched, such as [7], but few of the techniques seemed readily applicable to the project. It appeared that the mesh approaches would have trouble with noisy (i.e. erroneous) points in the point cloud.

The technique of determining a best-fit plane to sections of the point cloud was chosen over the mesh approaches. A best-fit plane is inherently resistant to noise because the noise will not drastically affect the properties of the plane. Another benefit of the best-fit plane approach was the ease of implementation. Calculating a mesh for a point cloud is a major research field in computer engineering and there are many complex solutions to the problem. Calculating a best-fit plane is considered a solved problem and simple to implement.

It was clear that a visualization tool would be needed to view the point cloud and the results of the analysis. After a review of the available point cloud viewers on the Internet, it was decided that creating a point cloud viewer specifically for this project would be beneficial. C++ and

OpenGL were chosen to implement this point cloud viewer because they are versatile, freely available, and extremely powerful. A preliminary version of the point cloud viewer was constructed. It was able to display the points and the best-fit planes, but had no other capabilities.

The remainder of the research primarily focused on creating additional capabilities of the point cloud viewer. These capabilities are described in detail in chapter V. The primary reference for constructing the point cloud viewer was [8].

After a preliminary version of the point cloud viewer was created, criteria for evaluating the suitability of a landing site were developed. These criteria are applied to the best-fit planes and used to evaluate the suitability as landing sites. The analysis methods are discussed in chapter IV. Techniques for visualization the analysis results were also considered and added to the point cloud viewer as the research progressed. These visualization methods are discussed in detail in chapter V.

The question of how well the point cloud viewer cloud helps a mission supervisor became relevant in the later stages of the research. It was decided that the best way to answer this question was to compose a user study similar to the one developed by Culhane [9].

B. Thesis Organization

This thesis describes the theory behind the stereo vision techniques used in this project, and then delves into the application of stereo vision to UAV navigation. Chapter I provides a history of how the project evolved and a summary of approaches and technologies that were considered. Chapter II describes the theory behind stereopsis, and details the commercially available image capture options considered for use in this project. This chapter also distinguishes the methods designed specifically to accomplish this project from preexisting methods. Chapter III describes the configuration and use of the Bumblebee stereo vision system from Point Grey Research. Chapter IV describes the methods used to identify suitable landing sites. This chapter is quite extensive as it details how the landing site is broken down into components to be analyzed, and the mathematics behind determining the suitability of a landing site. Chapter V describes the highlight of this thesis: the point cloud viewer program written specifically to visualize the stereo vision information and analysis. The features of the program and the design of the internal code structure are described. Chapter VI describes the user study performed in order to quantify the

benefit of the point cloud viewer to the mission. Finally, Chapter VII concludes the thesis and describes future work.

II. STEREOPSIS CONCEPTS AND APPLICATIONS

Isaac Newton once wrote that "if I have seen further it is by standing on the shoulders of giants." The works of this thesis echo that sentiment. This chapter describes the concept of stereopsis and some of the existing applications that made the accomplishments of this project possible, along with a description of new contributions.

A. Stereopsis

Stereo vision is a well known subject in the scientific community. Stereopsis, or stereo range estimation, was first described by Charles Wheatstone over one and a half centuries ago. The geometry behind stereopsis is considered solved, but the problem of pixel correspondence is not and research on the topic is still being done.

Stereopsis is the process of deducing 3D information from two or more 2D images of the same scene. The images must be obtained from different viewpoints, and the relative location of each camera capturing the images must be known to obtain quantitative measurements. 3D information can be obtained by finding corresponding locations between points in multiple images. This section describes the mathematics and techniques behind stereopsis. Sharkasi provides a very complete overview of stereopsis in [10] and much of his information, equations, and figure designs are reiterated here. This chapter also draws from [11].

1) Camera Model and Calibration

A mathematical model of the cameras is necessary for stereo range estimation. The most basic model is the pinhole camera, which is created by poking a small hole in a thin opaque material. This hole is the optical center of the camera and is the origin of the camera's coordinate frame. Light passes through the hole onto an image plane located a distance f behind the focal plane. The distance f is known as the focal length. The image that is projected onto the image plane is an inverted version of how a human would perceive the scene, so it is convenient to place a virtual image plane a distance f in front of the focal plane. The projection of the scene on this plane is not inverted. The image plane, focal plane, and virtual image plane are all parallel. The Z axis is perpendicular to all three planes and passes through the focal point. Figure 1 depicts the setup of the pinhole camera and Figure 2 depicts the relevant coordinate frames that apply.

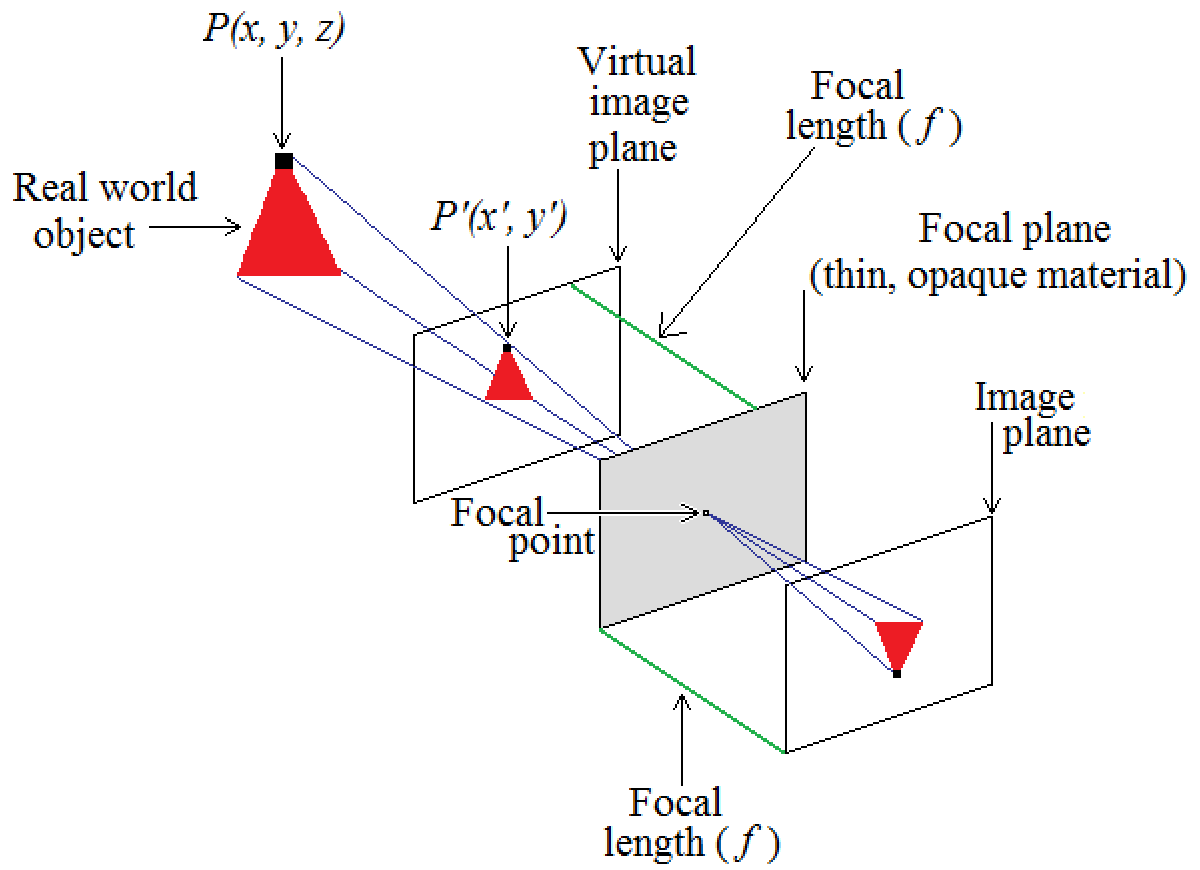


Figure 1: A pinhole camera.

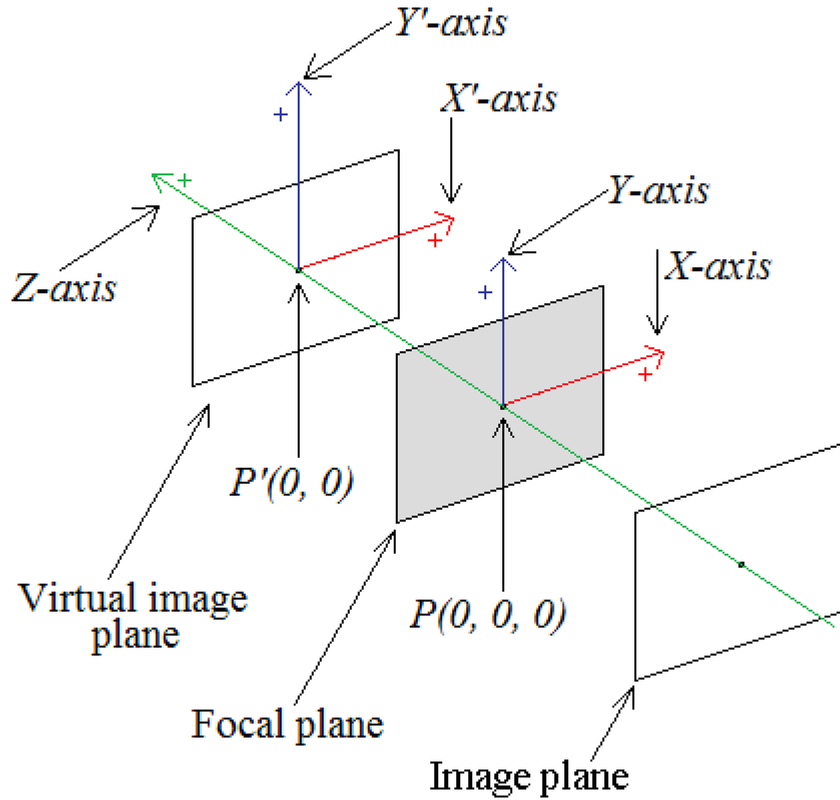


Figure 2: Axis setup for a pinhole camera and relevant planes.

The X, Y, and Z axes compose the left-handed camera coordinate frame. The X' and Y' axes compose the image plane coordinate frame. A mapping exists between all locations in the real world and locations in the image plane. The relationship between an object's location in the real world – denoted $P(x, y, z)$ – and the object's location on the virtual image plane – denoted $P'(x', y')$ – is expressed by the following equations:

$$x' = \frac{f \cdot x}{z} \tag{1}$$

$$y' = \frac{f \cdot y}{z} \tag{2}$$

These equations can be condensed into a single linear transformation when represented in projection space:

$$\begin{pmatrix} r \cdot x' \\ r \cdot y' \\ r \end{pmatrix} = \begin{pmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} \tag{3}$$

The left hand side of the previous equation is the homogenous coordinate of a point on the image plane, with respect to the image plane's coordinate frame. The vector $(x, y, z)^T$ is the corresponding homogeneous coordinate in the real world, with respect to the camera's coordinate frame. It is convenient to rewrite this single linear transformation into more components for use in further derivations:

$$x_{image} = K \cdot x_{camera} \tag{4}$$

where x_{image} is the image plane coordinate and x_{camera} is the real world coordinate. K is a 3×3 matrix known as the camera calibration matrix. Through several simple modifications to the calibration matrix, a suitable realistic model of a camera can be constructed from this simple pinhole camera model.

There are two main differences between a pinhole camera and a digital camera that must be considered when forming the mathematical model. First, a digital camera contains a light sensor to record the picture. This light sensor is called a charged-coupling device (CCD). The second difference is that a digital camera uses a lens to bend light to accommodate the focal point of the camera.

The modifications that are made to the calibration matrix account for the CCD properties and camera lens geometry. The CCD is an array of photo-sensing elements that lie on the image plane. The camera lens bends the incoming light and distorts the image, so a mathematical operation must be applied to the image to undo this distortion.

The calibration matrix modifications that account for the CCD properties are simple. The skew of the CCD sensing elements, denoted as s, must be input to the matrix. If the sensing elements are not exactly parallel to the focal plane then this will correct it. In many cases, the sensing elements are exactly parallel to the focal plane, and $s = 0$. Additionally, each sensing element in the CCD accounts for a single pixel of the image. Thus, the number of pixels per unit length

must be input to the matrix. This is known as the magnification value, and can be different for height and width, necessitating the composition of two separate variables:

$$a_u = -f \cdot k_u \tag{5}$$

$$a_v = -f \cdot k_v \tag{6}$$

In general, $a_u = a_v$. Inserting s, a_u, and a_v into the calibration matrix produces part of a realistic model. One last modification is necessary for the calibration matrix. For purposes of computation it is convenient to place the origin at one of the corners of the image. The points of the image can be shifted by inserting the row and column distance of the image plane's coordinate frame to the desired origin. These distances are denoted by p_x and p_y as seen in Figure 3.

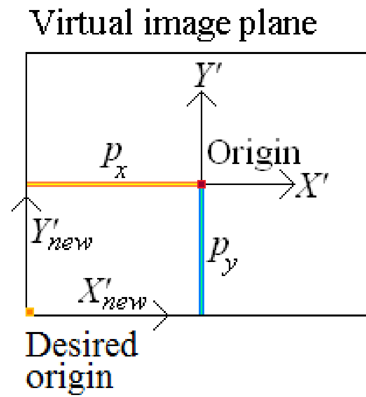


Figure 3: Camera frame offset. The basis for the design of this image came from [10].

With these modifications the calibration matrix becomes:

$$K = \begin{pmatrix} a_u & s & p_x \\ 0 & a_v & p_y \\ 0 & 0 & 1 \end{pmatrix} \tag{7}$$

After the CCD properties have been accounted for, the distortion caused by the lens geometry must also be corrected. The following operations can be applied to the image to correct for radial lens distortion:

$$x_{corrected} = p_x + L(r)(x - p_x) \tag{8}$$

$$y_{corrected} = p_y + L(r)(y - p_y) \quad (9)$$

where…

$$L(r) = 1 + k_1 \cdot r + k_2 \cdot r + k_3 \cdot r \ldots \quad (10)$$

$$r = \sqrt{(x - p_x)^2 + (y - p_y)^2} \quad (11)$$

$L(r)$ is a Taylor series expansion and the coefficients k_1, k_2, k_3, … are properties of the camera. These coefficients are generally documented by the manufacturer of the camera.

The mathematical model of the camera so far accounts for what are known as intrinsic parameters. The extrinsic parameters are added to the model by multiplying the calibration matrix by the measured rotation matrix (R) and translation vector (t) of the camera. The fully composed model is known as the camera matrix, W:

$$W = K[R|t] \quad (12)$$

The camera matrix W replaces the original calibration matrix K in equation (4) to produce the proper image coordinates of each pixel:

$$x_{rectified} = W \cdot x_{camera} \quad (13)$$

2) Multiple View Geometry and Rectification

Epipolar geometry describes the geometric setup of stereo vision. Figure 4 depicts this setup, with O_L and O_R as the optical centers of two cameras. The virtual image planes can be seen in front of the camera focal points. The plane that intersects an arbitrary 3D point P and the optical centers of the cameras is called an epipolar plane, and the line that passes through the optical centers of both cameras is known as the baseline.

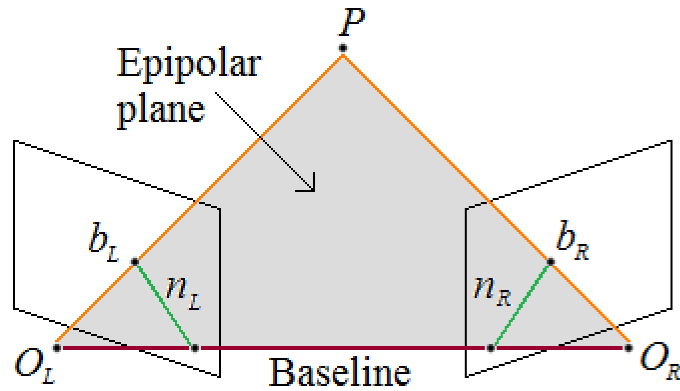


Figure 4: Epipolar geometry. The basis for the design of this image came from [10].

For the given point P, it is known that the projection point of the left image is b_L. This implies that the same real world location (as seen from the right camera) must lie along the epipolar line (from n_R to b_R) in the right image. Thus, by identifying the corresponding point in the right image, the 3D real world location of the point can be deduced.

This procedure of finding corresponding points can be simplified even further by moving the virtual image planes so they are coplanar. Once this is accomplished, the epipolar line (from n_R to b_R) becomes horizontal. That is, the image can be examined a single row at a time, as shown in Figure 5:

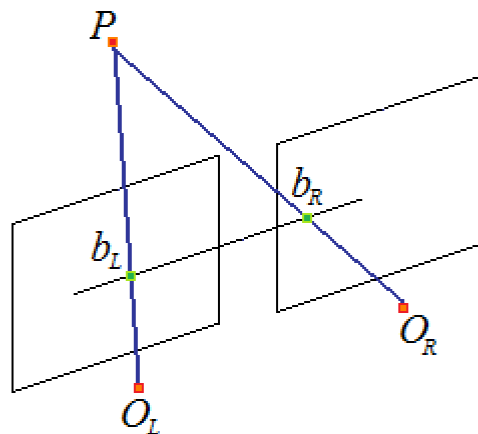


Figure 5: The final geometric setup of rectified images. The basis for the design of this image came from [10].

Figure 6 and Figure 7 show an example scene from the left and right viewpoints of a stereo vision camera system that have not been rectified. Figure 8 and Figure 9 show the rectified versions of the images, exactly like the setup in Figure 5.

Figure 6: Unmodified left image from a stereoscopic camera system.

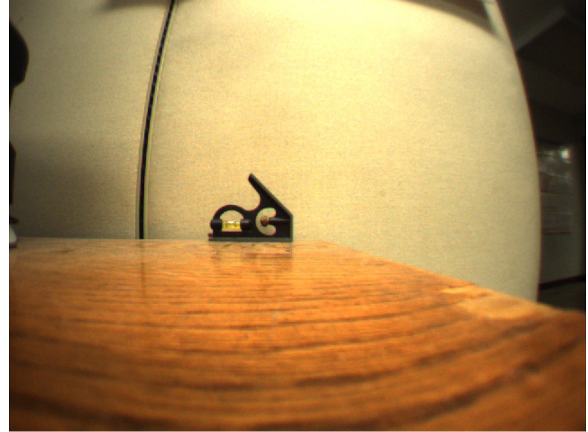


Figure 7: Unmodified right image from a stereoscopic camera system.

Figure 8: Left rectified image from a stereoscopic camera system.

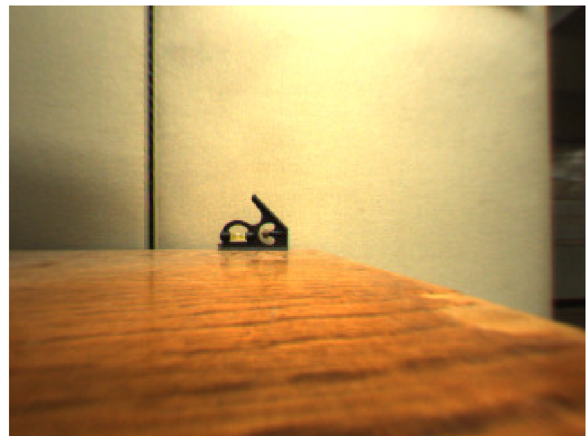


Figure 9: Right rectified image from a stereoscopic camera system.

3) Pixel Correspondence

The main difficulty in stereopsis is identifying correspondences between points in both images. As previously mentioned, there are several ways to search for these correspondences. One way to do this is to detect image features and then attempt to match features from one image with features in the other image. Example features are: intensity, edges (discovered via an edge detection algorithm), texture elements, et cetera. Feature-based methods of finding correspondences often use high-level abstraction of the scene data, and can lack point data where there are few or no features to analyze.

Correlation-based methods of finding pixel correspondence also exist. These methods generally make several comparisons and then choose the best match. The camera system used in this thesis employs the Sum of Absolute Differences (SAD) metric to correlate pixels, which is a fast and parallelizable correlation-based method of finding pixel correspondences.

The Bumblebee documentation [6] describes the SAD metric with this expression:

$$\min_{d=d_{min}}^{d_{max}} \sum_{i=-\frac{m}{2}}^{\frac{m}{2}} \sum_{i=-\frac{m}{2}}^{\frac{m}{2}} \left| I_{right}[x+i][y+j] - I_{left}[x+i+d][y+j] \right| \quad (14)$$

where d_{min} and d_{max} are the maximum disparities, m is the mask size, and I_{left} and I_{right} are the right and left image arrays. Note that the brackets after I_{left} and I_{right} are indexes of arrays, and not quantities to be multiplied. This notation is used in computer programming. The mask size is the square neighborhood around a given pixel that the algorithm uses to compute a correlation value. The distance between the two corresponding pixels for the left and right image is known as the disparity.

The end goal of the correspondence algorithm is to generate a set of 3D points from the given 2D images. This set of 3D points is called a point cloud.

B. Description of Stereo Vision System Options

Several image capture options were considered for improving the design of the navigation system. The factors that were considered include cost, size, and capabilities of the system. Hulin and Schüßler [12] present an extensive overview of these considerations, and their description was the main source of information for many of the design decisions.

The first design decision that was made was whether the system should be allowed to emit light to enable or improve image capture. Systems that do not emit radiation are typically preferred for military applications because they are not easily detectable by the enemy [12]. This was not a primary consideration for our design since there was little concern for stealth during the intended application of nuclear forensic analysis. The system currently does not emit light, but future work could be pursued to add a light source to enable nighttime operation.

The next design decision that was made was which spectrum of light to capture. This choice was mainly influenced by the cost of the camera system. Systems that capture the visible spectrum of light are typically the cheapest [12]. An additional consideration was time-dependence of image capture. Images that look different during daytime and nighttime (due to illumination) must be processed differently [12]. Some spectra of light can avoid or minimize this problem, such as ultraviolet-C. This type of radiation is emitted by the sun, but is completely filtered out by the earth's atmosphere. Thus, images captured using ultraviolet-C cameras look the same regardless of lighting conditions but require artificial illumination. Constant lighting conditions would be a huge simplification (and advantage) to the image processing techniques, but would incur a steep cost. Therefore, visible spectrum image capture was chosen for the design. Visible light image capture varies with solar illumination, and as a result the system was confined to daytime operation only.

The Bumblebee stereo vision camera system from Point Grey Research is sufficiently small to fit on a UAV (it has a mass of 342 grams and a baseline length of 156 millimeters [13]), and is relatively low in cost. Thus, it adheres to the constraints of the design. This visible-spectrum camera system, shown in Figure 10, was subsequently chosen for the implementation of the system.

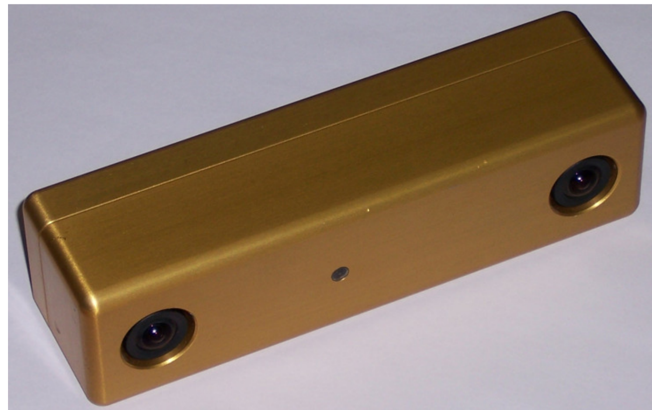


Figure 10: The Bumblebee stereo vision camera system from Point Grey Research, Inc.

C. List of Contributions

The description of Bumblebee parameters, capabilities, and image quality in chapter III is preexisting work that was done by Point Grey Research. They have provided a versatile and

robust commercial stereo vision system, complete with a software interface. The Bumblebee hardware is their design, as is the software used to generate the point cloud.

The analysis methods applied to the point cloud and the point cloud viewer were created specifically for use in this project. The analysis methods determine landing site suitability from the point cloud information. The point cloud viewer is a computer program written for the purpose of visualizing the results of the analysis performed on the 3D points. Both the point cloud analysis and the point cloud viewer were the main contributions of the thesis.

The point cloud viewer uses OpenGL, which is a modern (and surprisingly easy to use) graphics library. This research did not contribute anything to the OpenGL community, but simply used the graphics library to construct the program. In addition to that, the design for a basic OpenGL program was referenced from [8], and the point cloud viewer contains code from the tutorial section of that website. Obviously, the code has been heavily modified to fit the needs of the project.

The user study presented in chapter VI is also a major contribution of this thesis. It was designed with the specific intent of showing the improvement in performance provided by the work in this thesis.

III. DESCRIPTION OF BUMBLEBEE PARAMETERS, CAPABILITIES, AND IMAGE QUALITY

The Bumblebee software uses the Sum of Absolute Differences (SAD) metric to process image input and generate a point cloud. Several pre- and post-processing techniques are also utilized to enhance image quality and improve the accuracy of the point cloud [6].

One of the weaknesses of two-camera stereo systems such as the Bumblebee is that the rows of coupled images are examined for pixel similarity. This can lead to poor point cloud generation if each of the pixels in the row are similar to each other. Figure 11 shows an image in which the Bumblebee would typically produce a high-quality point cloud, while Figure 12 shows the same image (rotated 90 degrees) in which the Bumblebee would produce a low-quality point cloud.

Figure 11: An image that would perform very well with many stereo matching techniques.

Figure 12: An image that would perform very poorly with many stereo matching techniques.

All image enhancement techniques provided by the Bumblebee software were enabled for the point cloud viewer, including edge detection, validation, and subpixel interpolation. Figure 13 shows an example scene that is used to demonstrate the difference between having these enhancement techniques enabled and disabled. Figure 14 shows the point cloud that was generated with all image enhancement techniques disabled, while Figure 15 shows the point cloud of the same scene with image enhancement techniques enabled. There is much less noise with these techniques enabled. Figure 16 highlights which portions of the point cloud were most affected by the image enhancement techniques. The reddish hue in Figure 13 through Figure 16 is caused by indoor lighting. The Bumblebee was optimized for outdoor lighting conditions at the time this data was captured.

Figure 13: A scene constructed to demonstrate the effects of the point cloud optimization techniques provided by the Bumblebee software.

Figure 14: The example scene with no point cloud optimization enabled.

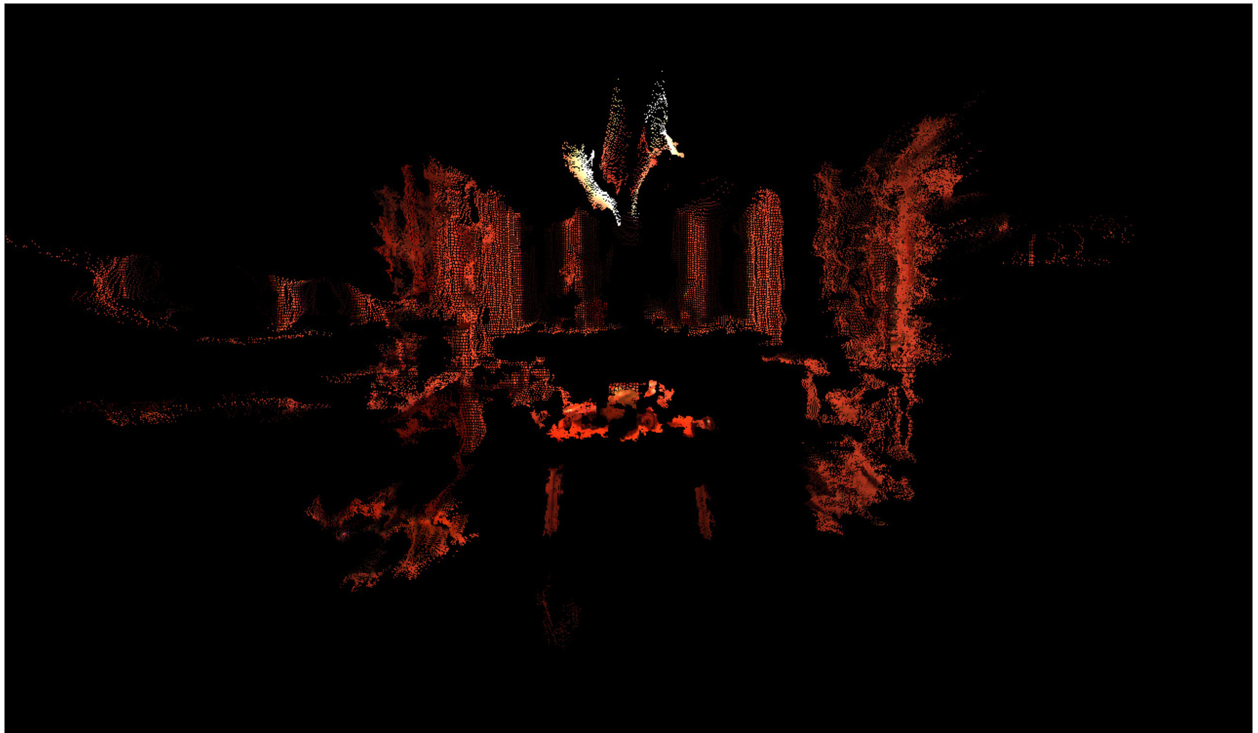


Figure 15: The example scene with point cloud optimization enabled.

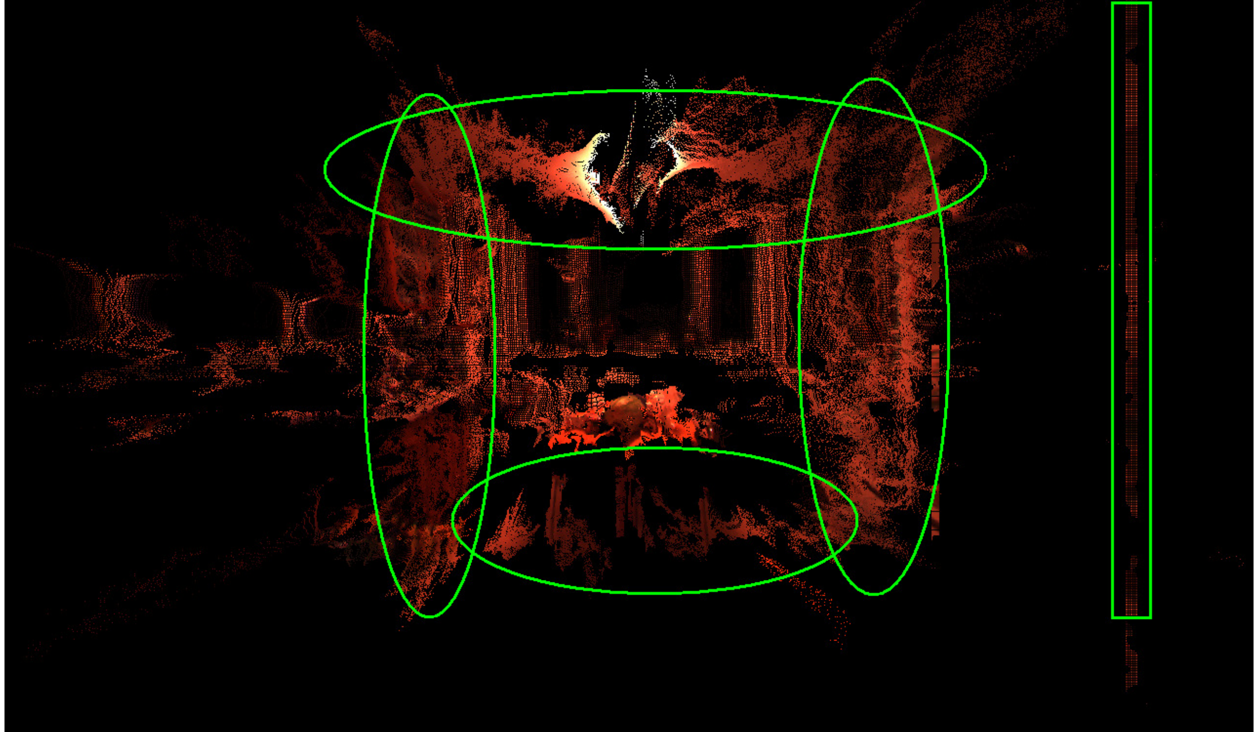


Figure 16: The example scene with no point cloud optimization enabled. The noisy points that were mostly removed by the image enhancement techniques of the Bumblebee software are highlighted in green.

The Bumblebee camera system is capable of capturing images at 640×480 or 320×240 resolution. Lower resolution captures require less processing time. The 640×480 resolution was used in this study.

As recommended by the Point Grey documentation [6], a depth limit is applied when the point cloud is generated. Any points that are perceived to be greater than five meters away from the Bumblebee are filtered out. This limit of five meters was selected specifically for this project and is hard-coded into the image capture software.

The Bumblebee can capture images in grayscale or color. Grayscale images take less time to process since less data is recorded during the capture operation. Color images, however, improve accuracy. Color images were captured for use in this study.

One avenue of research that was not explored in this thesis was the "region of interest" capability of the Bumblebee. Specifying a region of interest allows the observer to generate point clouds for selected sections in the scene, thereby shortening the processing time. This capability was not

considered a high priority when performing the research for this thesis, and thus was not explored in depth.

IV. METHODS DEVELOPED FOR IDENTIFICATION OF SUITABLE LANDING SITES

After stereo techniques have been used to generate a 3D point cloud, the resulting range data can be analyzed to identify potential landing sites. Using multiple linear regressions on sub-divided sections of the visual image, planar sections are identified that may or may not satisfy the criteria needed for landing. Intuitively, a suitable landing site must be relatively flat, approximately horizontal, and sufficiently large in extent to accommodate the vehicle. Additionally, the site should not lie in close proximity to tall obstructions that represent collision hazards.

This chapter describes the analysis that is applied to the point cloud to evaluate the suitability of a location as a landing site. Chapter V describes how this analysis is displayed using a point cloud viewer program that was developed specifically for the purposes of this thesis.

A. Division of the Point Cloud

The point cloud is divided into sections so that detailed information can be deduced about specific areas of the scene. Each 3D point in the cloud is associated with a single 2D point in the reference image. The 3D points are partitioned by dividing the reference image into rectangular sections. The programmer is able to choose how many sections the reference image is divided into. For the work presented in this thesis, the reference image was always divided into thirty rows and thirty columns. The numbers of divisions were chosen for their granularity. If the divisions are too small then there won't be enough points per division to compose a meaningful best-fit plane. If the divisions are too large then the information about a specific area may not be detailed enough to produce reliable conclusions. Each division of the 2D image is examined, and a best-fit plane is fit to the corresponding subset of 3D points that are generated from within that division.

B. Best-fit Plane Generation

After the point cloud is divided into sections, a multiple linear regression is performed on the 3D points that were generated from each section. A standard multiple linear regression technique is used to generate the planes. A sample section is shown in Figure 17. The x and y coordinates (horizontal directions) are the independent variables and the z coordinate (vertical direction) is the dependent variable.

Figure 17: A sample best-fit plane and its corresponding data points.

Note that the vertical distance from a given point to the best-fit plane is taken to be the error. This is not necessarily the same as the minimum distance to the plane, as shown in Figure 18.

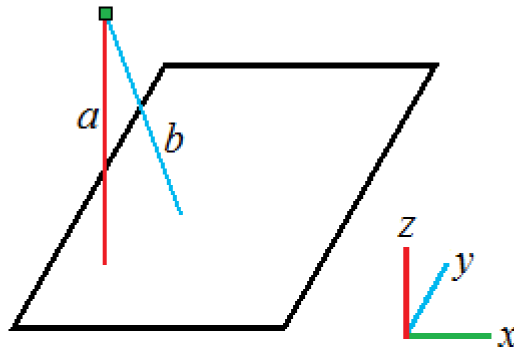


Figure 18: Distance *a* depicts the vertical distance from a selected point to its corresponding best-fit plane, while distance *b* depicts the shortest distance from the point to the plane.

Given a set of m points $\{(x_i, y_i, z_i)\}_{i=1}^{m}$ the best-fit plane is found by minimizing the sum of the squared errors for the equation $z = Ax + By + C$ for all points. As stated previously, the error is measured by the vertical distance between the plane and each point, squared:

$$Error(A,B,C) = \sum_{i=1}^{m} [(Ax_i + Bx_i + C) - z_i]^2 \qquad (15)$$

Thus, the problem can be set up using a system of linear equations. Solving for $[A, B, C]^T$ yields the coefficients for the best-fit plane:

$$\begin{bmatrix} \sum_{i=1}^{m} x_i^2 & \sum_{i=1}^{m} x_i y_i & \sum_{i=1}^{m} x_i \\ \sum_{i=1}^{m} x_i y_i & \sum_{i=1}^{m} y_i^2 & \sum_{i=1}^{m} y_i \\ \sum_{i=1}^{m} x_i & \sum_{i=1}^{m} y_i & \sum_{i=1}^{m} 1 \end{bmatrix} \begin{bmatrix} A \\ B \\ C \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^{m} x_i z_i \\ \sum_{i=1}^{m} y_i z_i \\ \sum_{i=1}^{m} z_i \end{bmatrix} \qquad (16)$$

Once the equation for the best-fit plane is produced, the normal vector $\hat{n}$ of the plane is given by the coefficients of the equation for the plane:

$$\hat{n} = \langle A, B, C \rangle \qquad (17)$$

Information for this best-fit technique was referenced from [14].

C. Determining Suitability of a Landing Site

To evaluate the quality of the landing surface, the gradient of each best-fit plane and the variance of gradient between adjacent best-fit planes were considered. If the angular difference between the normal vector and the vertical direction in world coordinates is less than a predetermined threshold, then that planar section is classified as acceptable for landing (as shown in Figure 19). Adjacent planes that have dramatically different orientations indicate uneven terrain or debris, and therefore represent areas that should be avoided. In the current implementation, the system compares normal directions for each plane with its neighboring planes (as shown in Figure 20), and if the average deviation of all the surrounding planes exceeds a threshold, the center plane is rejected as unacceptable. Up to eight neighboring planes may be present in the calculation (one per each horizontal, vertical, and diagonal neighbor of the current plane, as shown in Figure 21).

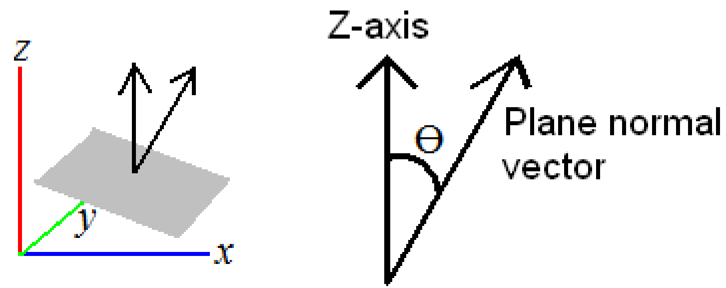


Figure 19: To help determine the plausibility of a landing site, the normal vector of a given plane can be compared to the vertical direction.

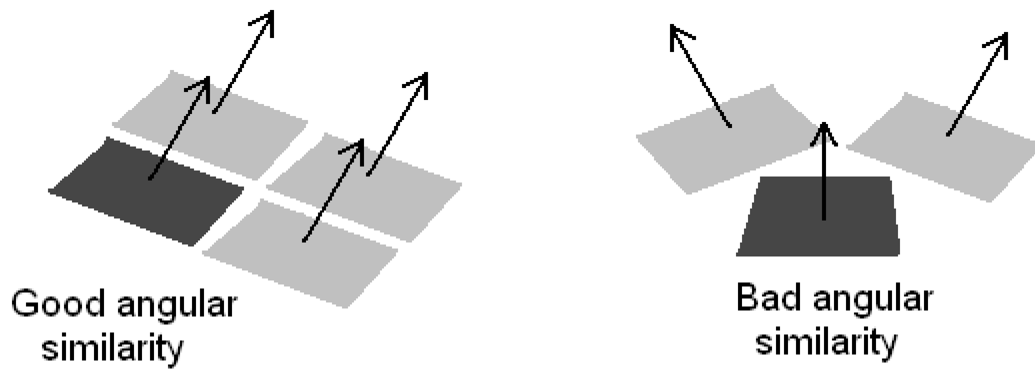


Figure 20: To help determine the plausibility of a landing site, the normal vector of a given plane can be compared to the normal vector of neighboring planes.

Figure 21: There may be up to eight adjacent planes considered in a proximity analysis.

To evaluate the concept, the following criterion has been selected:

$$\left|\cos^{-1}(\hat{n}_i \cdot \hat{Z})\right| \leq 25° \quad (18)$$

where $\hat{n}_i$ is the unit normal vector of the ith best-fit plane, and $\hat{Z}$ is the global vertical orientation unit vector.

Additionally, the normal vector of the plane in question is compared to the normal vectors of adjacent planes. These comparisons are averaged together, and must be less than or equal to $25°$ or the plane is rejected as unacceptable.

This threshold method is only one of several that can be implemented and seems to give conservative results on the quality of the landing site.

Statistical measures such as variance and kurtosis can also be used to determine the plausibility of a landing site. Kurtosis measures the "peakedness" of the data points relative to the best-fit plane. If a plane is fit to data points that have a sharp spike in any direction, it is very likely that an obstacle lies within the bounds of the plane. In these cases, the kurtosis calculation is designed to mitigate the chances that a landing site will be falsely marked as acceptable for landing. The measure of kurtosis for each best-fit plane is calculated and stored when an analysis is run. The value for kurtosis is given by the expression:

$$\frac{\mu_4}{\sigma^4} - 3 \quad (19)$$

where μ_4 is the fourth moment about the mean and σ is the standard deviation.

D. Example Scene

A model scene was created to test the landing criteria, shown in Figure 22. The full-size model was designed to reasonably represent an actual mission site, including two halves of a metal barrel, a wooden pallet (near the center), and other scattered metal and wood debris. A ground control operator might quickly identify the pallet in the center or the large grassy area in the top right of the image as reasonable landing sites, and so the algorithm was tested against this supposition. Figure 23 also shows the point cloud data for the scene.

Figure 22: An example scene obtained using the Bumblebee stereo vision camera system. This image is from the Bumblebee's right camera.

Figure 23: The point cloud of the example scene, obtained using images from the Bumblebee stereo vision camera system.

In Figure 24, the best-fit planes are overlaid onto the point cloud to evaluate the quality of the landing site. Planes that represent acceptable landing regions are colored in green, while unacceptable regions are colored in red. In this example, the pallet and grassy area have been identified as suitable landing sites, but a false-positive landing site is also indicated at the bottom right of the image. Figure 25 shows blue boxes drawn on the 2D image, indicating acceptable landing sites, and it is up to the operator to assess which areas are truly acceptable given that false positive indications may be present. Future work could focus on the use of statistical (data trend) analyses to further eliminate false positive regions. The green/red coloring in Figure 24 and the blue rectangles in Figure 25 are all automatically generated by the point cloud viewer. The green rectangles in Figure 24 are arranged identically to the blue rectangles in Figure 25 but may not appear that way due to occlusion and perspective inherent to the 3D view. Best-fit planes occasionally overlap with each other, creating the illusion that the best-fit plane in question is not rectangular or is arranged differently than its blue counterpart.

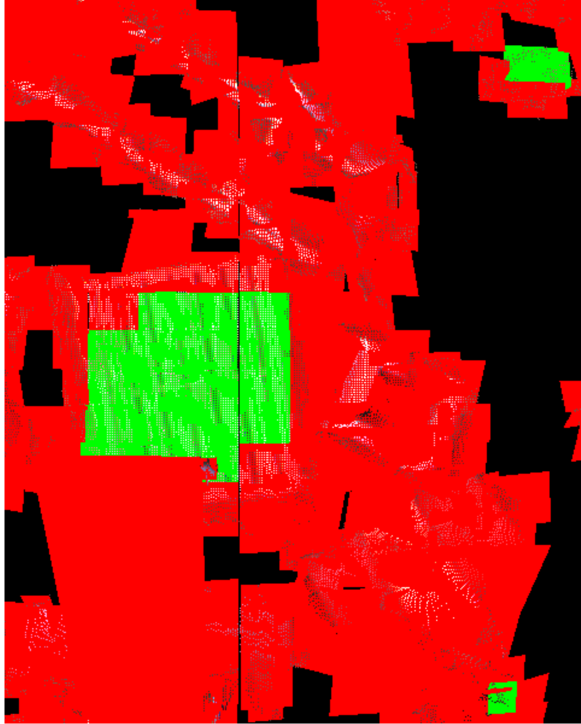


Figure 24: The example scene’s point cloud with overlaid best-fit planes.

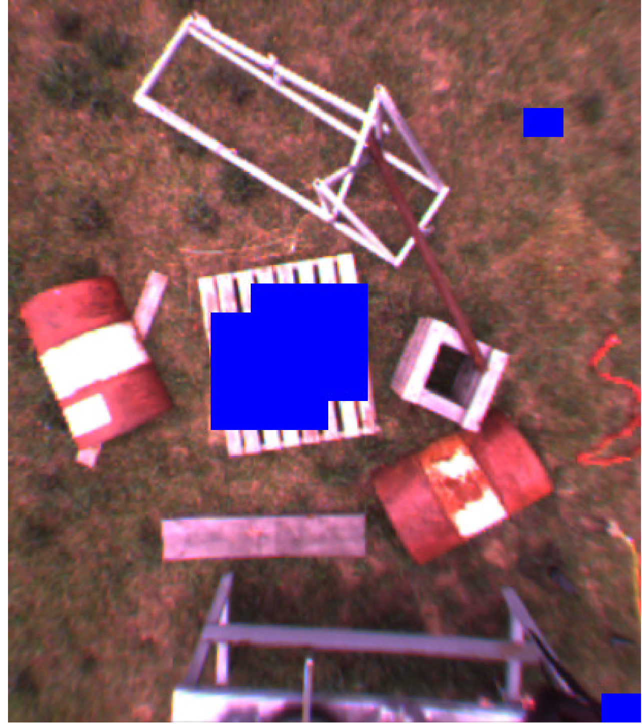


Figure 25: The example scene's 2D representation with acceptable landing sites indicated by blue boxes.

A simulated helicopter descent profile was created by moving the stereo camera system downward while capturing images at three elevations. Images from the reference camera are shown in Figure 26 through Figure 31 along with the overlay of acceptable landing sites as automatically computed by the system. Note that the suggested sites change somewhat during the descent, due to lighting variability and uncertainty in the stereo data. One way of accommodating these changes is to keep a memory of previously identified acceptable landing sites and use them in a weighted estimate.

Figure 26: A simulated descent, at a height of 2.44 meters.

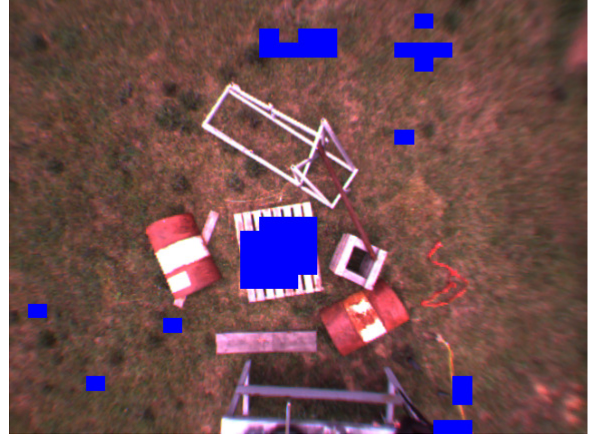


Figure 27: A simulated descent, at a height of 2.44 meters with acceptable landing sites overlaid.

Figure 28: A simulated descent, at a height of 2.11 meters.

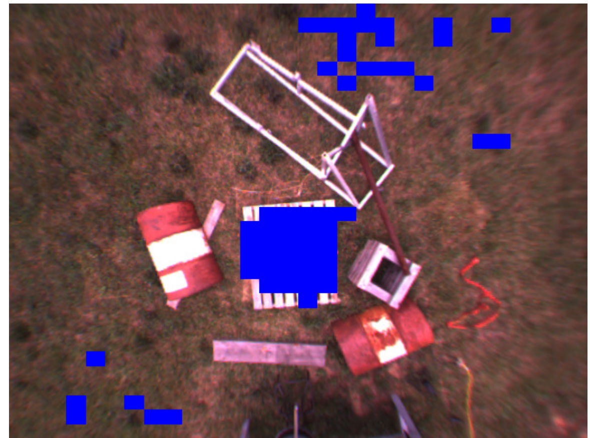


Figure 29: A simulated descent, at a height of 2.11 meters with acceptable landing sites overlaid.

Figure 30: A simulated descent, at a height of 1.42 meters.

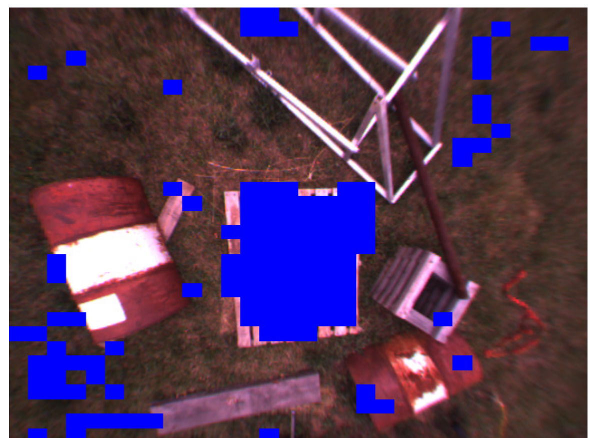


Figure 31: A simulated descent, at a height of 1.42 meters with acceptable landing sites overlaid.

V. POINT CLOUD VIEWER

The point cloud viewer's primary purpose is to enable visualization of the results of analysis performed on the 3D points collected from a scene. It is able to view any 3D point cloud and not just point clouds that are generated using the Bumblebee software. The program's code is modular, allowing for components to be used only when needed. It is also written to be easily portable to an operating system other than Windows. OpenGL and Windows API are used to create the graphical user interface.

This chapter details the extensive list of features of the point cloud viewer and how to use those features. A concise summary of the features is presented for quick reference in Table 1. This chapter also examines the internal code structure of the point cloud viewer in detail. Several design considerations (such as memory management) are also discussed. The code for the point cloud viewer is in Appendix A.

A. Features

The point cloud viewer has many features that allow the user to extract extensive amounts of information from a scene. It is capable of viewing both 2D and 3D representations of the scene (or a combination of the two). It also uses color to convey large amounts of information to the user in a graphical (and easily understandable) manner. The user can select portions of the scene to acquire more detailed information about a location. Navigation is also available to move about the scene so it can be rendered from different viewpoints.

1) 2D and 3D Representations of the Scene

The point cloud viewer can render a scene in 2D or 3D, depending on the user's choice. The 2D representation of the scene is simply the rectified image of the Bumblebee's right camera displayed as a flat plane. As the description implies, 2D representation does not express depth information but is useful for conveying a summary of the scene.

With 3D rendering, the point cloud is displayed and the user can optionally display the best-fit planes. This rendering mode is useful for debugging and obtaining detailed information about a scene. It is also the rendering mode capable of displaying the most diverse amount of information because the user can navigate through the 3D point cloud and examine the analysis that has been applied to the cloud. Additionally, the user can view the slope and color of the best-

fit plane to assess the quality of a section of the scene. The importance of a best-fit plane's color is discussed later in this chapter.

2D rendering is the default when the program is started. The user can switch to 3D rendering by pressing '3', and switch back to 2D rendering by pressing '2'. In 3D rendering mode, the best-fit planes are displayed by default. The planes can be hidden/shown by pressing 'B'.

2) Multi-dimensional Representation of the Scene

When the Bumblebee captures a scene, the point cloud viewer receives a 2D image of the scene and the 3D point cloud of the scene. The data is structured such that a mapping between the pixels of the 2D image and individual 3D points is preserved. Thus, given any point $P(x, y, z)$ in the 3D data set, the corresponding 2D pixel $P'(x', y')$ can be found, as depicted in Figure 32.

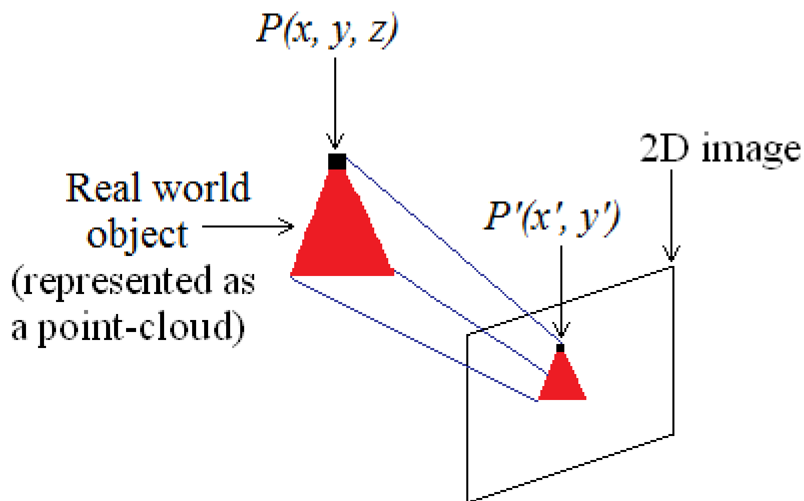


Figure 32: An abstract example that shows the mapping between a 2D pixel and a 3D point.

The section on stereopsis in the introduction of this thesis explains how the 2D locations on an image correspond to real world locations. The Bumblebee's right camera is the reference image when deriving the 3D point information, which is why it is displayed in the 2D rendering mode (instead of the left camera's image).

Another example that depicts the mapping between the 2D and 3D representation of the scene is shown in Figure 33. The blue rectangle near the center of the image is the bounding box for a section of the 2D representation of the scene. All 3D points that were produced from within that

bounding box are displayed near the top of the image (which are the same color as the wood pallet in the image), with their corresponding best-fit planes (shown in red).

This “2D/3D overlay mode” can be enabled/disabled by pressing ‘5’. When the program starts (in 2D rendering mode) this feature is enabled by default.

Figure 33: An example scene that depicts the mapping between the 2D and 3D representation of the data.

3) Input and Output of a Data Set

The 2D and 3D representation of the scene is bundled in a single data set. The program can open a data set in two ways. Using the first method, the data set can be opened from a file. The user must press ‘I’ and choose the data set to load using the standard “open file” dialog from Windows (shown in Figure 34). Using the second method, the user can capture a data set directly from the Bumblebee by pressing ‘C’. The camera system must be powered on and connected when the program starts for this option to work correctly.

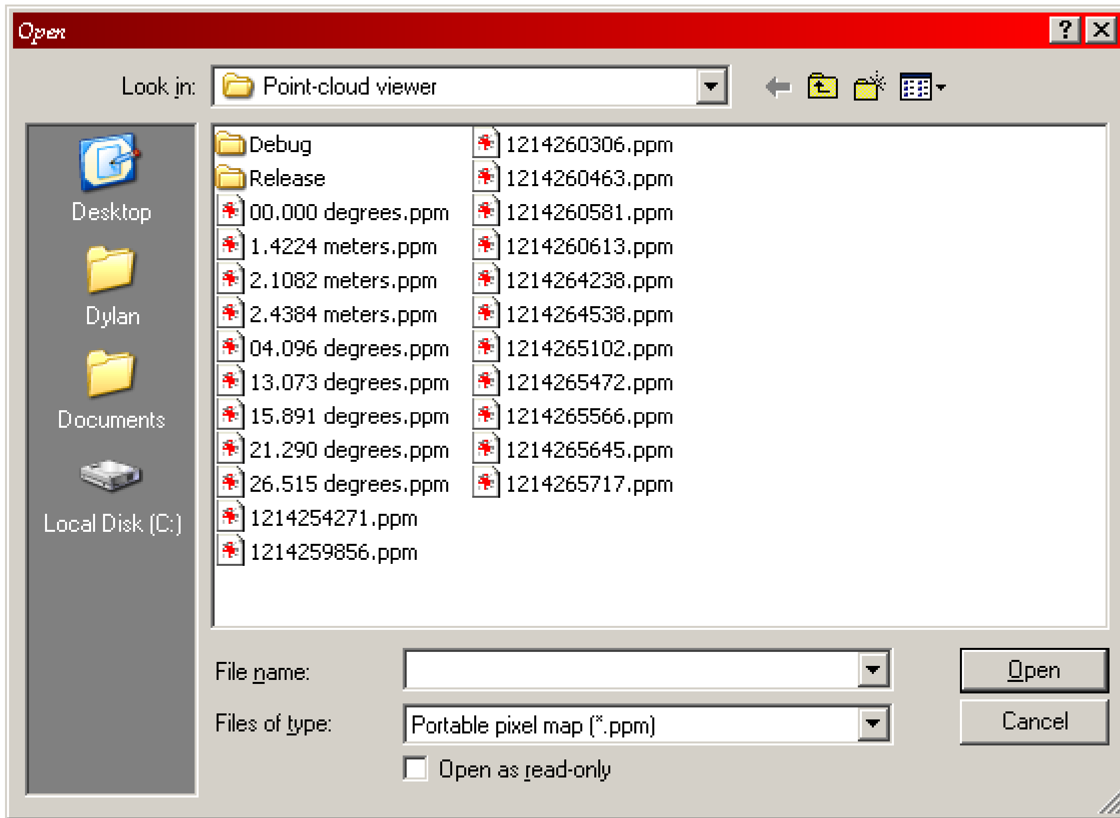


Figure 34: The "open file" dialog from the point cloud viewer.

The user can save a data set by pressing ‘O’. The scene is saved as a portable pixel map (PPM). The file name is determined by the timestamp of when the data were captured.

4) Selection

The user is able to click on a plane to acquire detailed information about a rectangular section of the scene. This works in both 2D and 3D rendering modes. When the user clicks on the plane, it will turn blue to indicate that it has been selected. Detailed information about the most recently selected plane is displayed in the title bar of the program. By default, “sticky” selection is enabled – so that each plane that is selected will be colored blue. The user can disable sticky selection by pressing ‘T’, thereby only allowing one plane, at most, to be selected at a time. Figure 35 depicts a scene with two planes selected and detailed information about the most recently selected plane in the title bar. While in 2D rendering mode, the user can press ‘M’ to select all planes that qualify as landing sites (thereby coloring those planes blue).

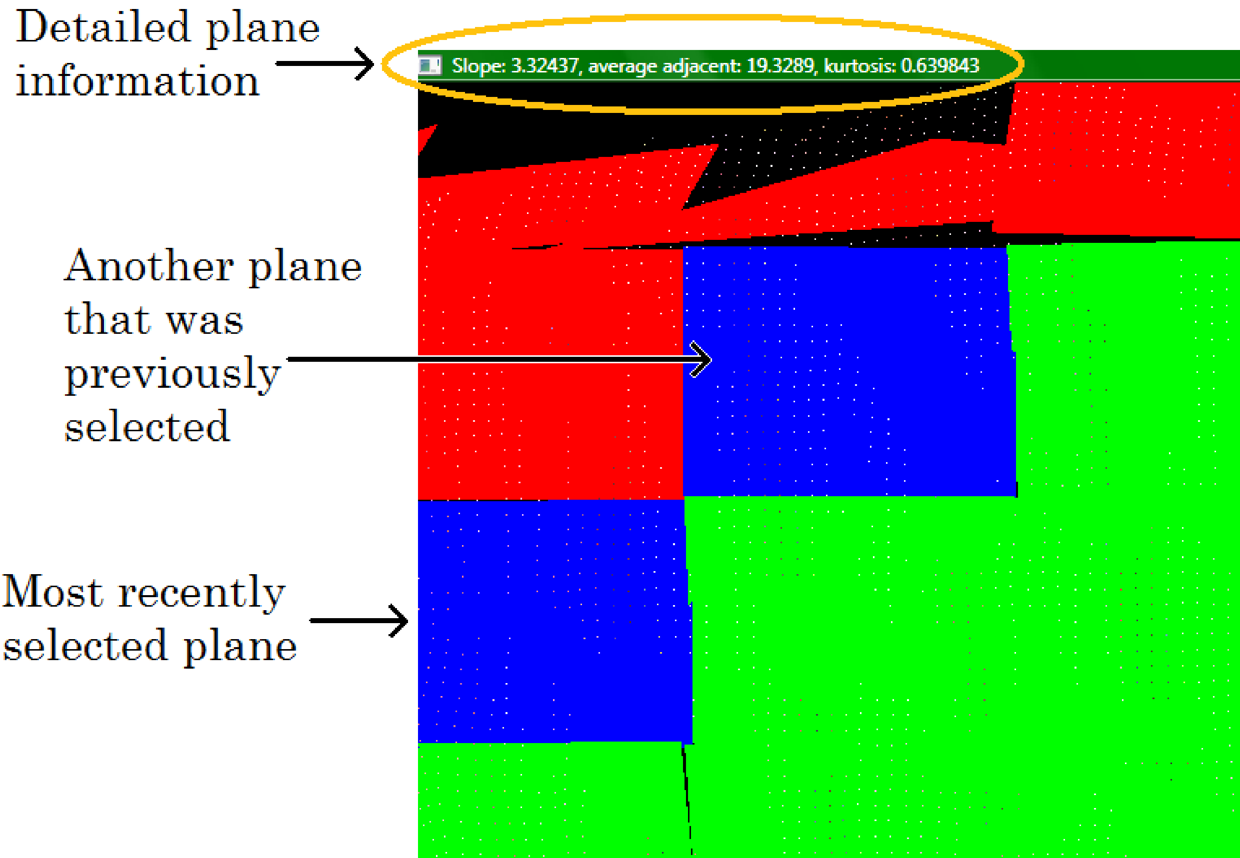


Figure 35: An example of selection and detailed plane information in the point cloud viewer.

5) Navigation

The user is able to examine the point cloud from any viewpoint. Moving the camera about the point cloud to obtain a different viewpoint is known as navigation. Navigation is accomplished by either using the mouse or the keyboard. The keyboard and mouse input that affect the view are described in Table 1. To reset the view to its default position, the user can press '0' (zero).

6) Mouse and Keyboard Input Commands

Table 1 presents a summary of all the input commands to the point cloud viewer.

Table 1: Summary of input commands for the point cloud viewer.

Input Sequence	Effect
2	Renders the scene as two-dimensional.
3	Renders the scene as three-dimensional.
5	Toggles the "3D overlay" feature for two-dimensional rendering mode. With this option enabled, planes that are selected in the 2D view will have their corresponding 3D points overlaid on the scene. This feature has no effect in 3D rendering mode.

B	Hides/shows the best-fit planes. Only applicable in 3D rendering mode.
I	Causes the program to display an "open file" dialog box, allowing the user to open a scene from a file.
C	Captures a scene from the Bumblebee. The scene is automatically displayed to the viewer.
O	Outputs the scene to a PPM file. The file name is the same as the timestamp of when the image was taken.
M	Selects all planes that are qualified as acceptable landing sites. Applicable to 2D view only.
T	Toggles sticky selection. Applicable in both 2D and 3D rendering modes.
Left/right arrow key	Translates the points in the x direction.
Up/down arrow key	Translates the points in the y direction.
Z	Translates the points in the positive z direction.
X	Translates the points in the negative z direction.
D	Affects the yaw of the scene in the negative direction.
A	Affects the yaw of the scene in the positive direction.
S	Affects the pitch of the scene in the negative direction.
W	Affects the pitch of the scene in the positive direction.
E	Affects the roll of the scene in the negative direction.
Q	Affects the roll of the scene in the positive direction.
0 (zero)	Resets the view to the default (that is, the origin). X, Y, Z, roll, pitch, and yaw of the scene are all set to 0.
F1	Sets the best-fit planes to be colored according to slope.
F2	Sets the best-fit planes to be colored according to the average angular difference of the surrounding planes.
F3	Sets the best-fit planes to be colored according to binary threshold values. (That is, planes that are designated as viable landing sites are colored as green, and all other planes are colored red.)
F4	Sets the best-fit planes to be colored according to kurtosis.
Left mouse button	Affects pitch and yaw of the scene.
Left mouse button while holding down shift	Affects pitch and roll of the scene.
Right mouse button	Affects x and y translation of the scene.
Right mouse button while holding down shift	Affects x and z translation of the scene.
Mouse wheel (middle mouse button)	Affects the z translation of the scene.

B. Code Structure and Code Interface

The point cloud viewer was written in C++, with responsibilities divided among several classes. Figure 36 depicts a class diagram of the point cloud viewer. The following is a breakdown of the responsibilities of each class/structure:

1) The Cloud Structure

The `Cloud` structure contains the most essential data for the point cloud viewer and performs all of the analysis that is applied to the cloud. Some of the more notable contents of the `Cloud` structure are:

- An array of points (containing the 2D and 3D coordinates of each point, and color information)
- The rectified right-camera image of the scene
- A two-dimensional array, containing pointers to the best-fit planes

The programmer can load data into a `Cloud` structure by opening a file, or by directly capturing the data from the Bumblebee. In the first case, if the programmer decides to load the cloud data from a file, the programmer must call `Cloud::Open`. The `Open` function attempts to open a portable pixel map (PPM) image with embedded 3D point information. Essentially, the user opens a 2D and 3D representation of the scene from the same file. In the second case, if the programmer decides to capture data directly, the `Cloud` structure must be passed to the global `Bumblebee` object. The `Bumblebee` object grabs data from the camera system and fills the `Cloud` structure.

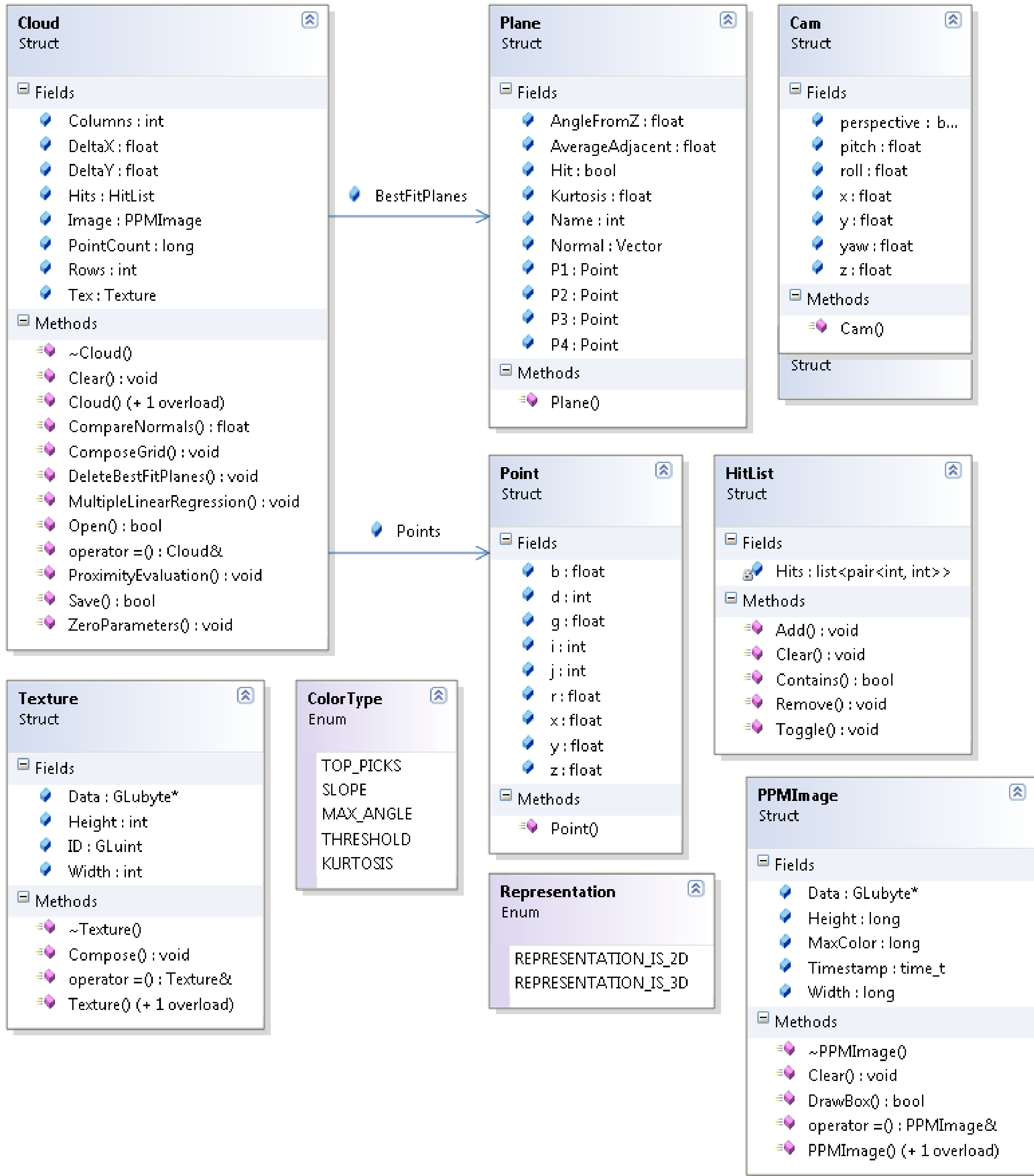


Figure 36: Class diagram of the point cloud viewer program.

Once data is loaded into the `Cloud` structure, the programmer can perform an analysis on the `Cloud` by calling `Cloud::ComposeGrid`. The function divides the point cloud into sections and performs a multiple linear regression on each section. The programmer must provide the number of desired rows and columns. The function then compares the resulting best-fit plane's normal vector with the vertical direction and the surrounding planes' normal vectors. These comparisons are performed within the `Cloud::MultipleLinearRegression` and `Cloud::ProximityEvaluation` functions. Figure 37 depicts the entire program's function call sequence. More specifically, Figure 38 depicts the function call sequence of the cloud analysis function, `Cloud::ComposeGrid`. The results of the analysis are stored within the `Cloud::Planes` array.

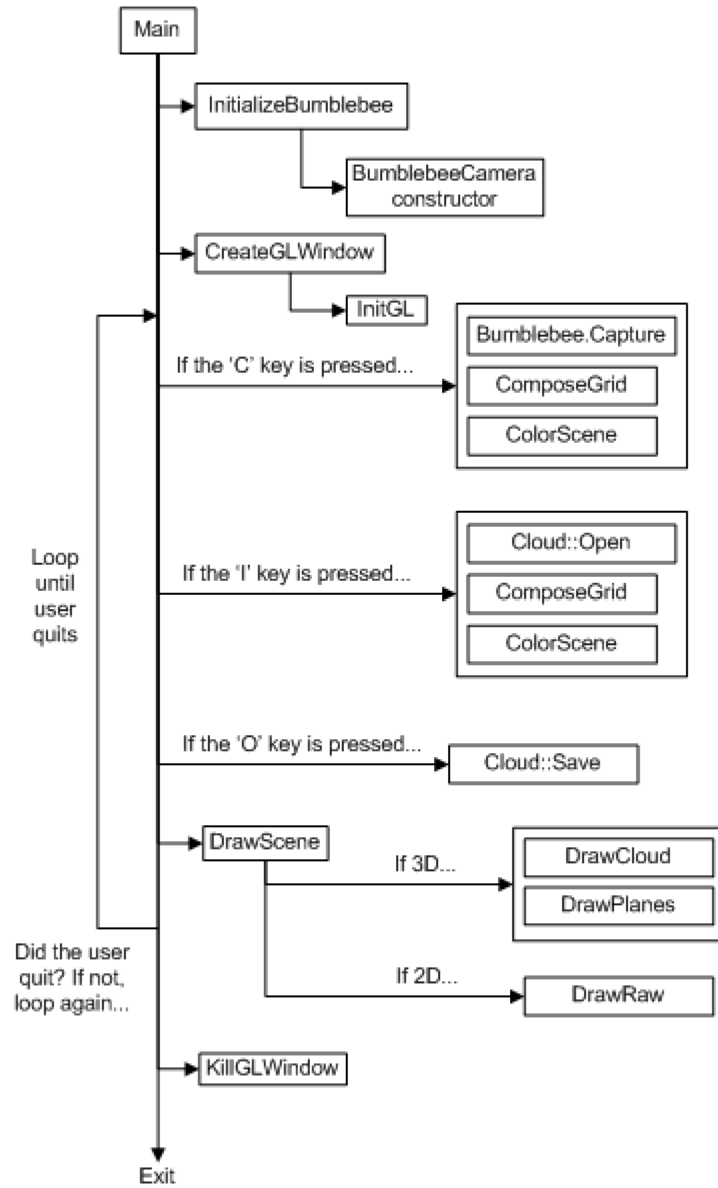


Figure 37: The flow of execution for the point cloud viewer program.

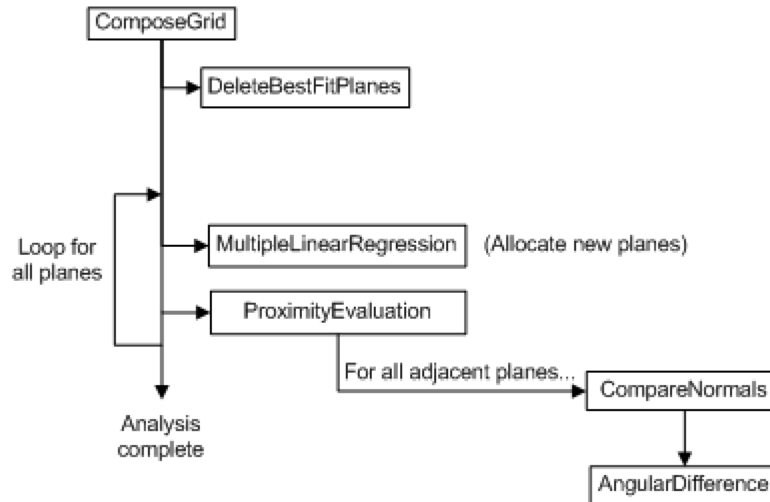


Figure 38: The flow of execution for the analysis of a point cloud.

It should be noted that the analysis of the cloud is highly parallelizable, if calculation speed becomes more of a concern for future work. The program could be easily modified to use OpenMP or MPI to greatly decrease overall calculation time.

After the analysis is complete, the point cloud viewer displays the results to the screen using OpenGL. The programmer is able to choose several visualization options when displaying the data. Resulting analyses can be displayed individually or in combination. For example, the programmer can choose to display the slope of the planes' normal vectors relative to the vertical direction, as shown in Figure 39. Alternatively, as seen in Figure 40, the programmer can set a threshold for certain attributes of a plane and display all the planes that meet the criteria as (for instance) green and all other planes as red. The captions of Figure 39 and Figure 40 explain the visualizations in detail.

Figure 39: A point cloud with best-fit planes overlaid. The planes are colored according to their slope (in comparison to the vertical direction). Flat planes appear lighter, while steep planes appear darker.

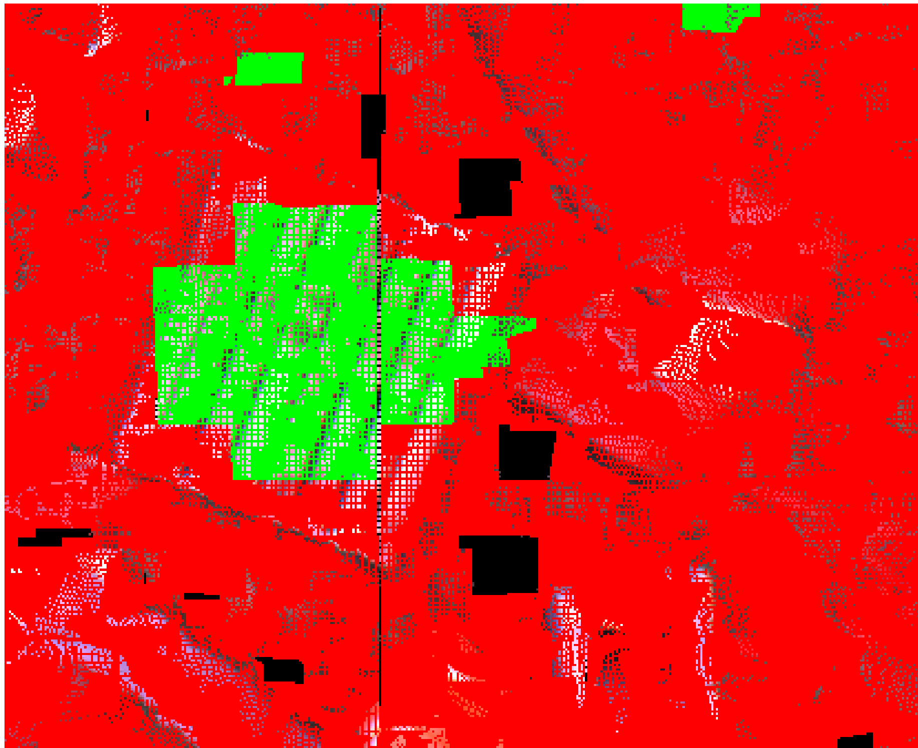


Figure 40: A point cloud with best-fit planes overlaid. The planes are colored according to their slope (in comparison to the vertical direction). In this particular case, a steepness threshold of 25 degrees was chosen. All planes that have a slope of less than or equal to 25 degrees are colored green, while all other (steeper) planes are colored red.

2) The BumblebeeCamera Class

The `BumblebeeCamera` class is designed to oversee the responsibilities of configuring and operating the Bumblebee. These responsibilities include configuring the camera system when the point cloud viewer is first started, capturing data with the camera system, and passing captured data back to a `Cloud` structure.

Only one instance of this class is allowed to exist in the entire program. This single instance has global scope, and is guaranteed to be unique because the class' constructor, copy constructor, and assignment operator have private scope. The camera system is initialized in the `BumblebeeCamera`'s constructor. The programmer is able to access the class through the global `Bumblebee` pointer, which must be initialized with the `InitializeBumblebee` function. Once initialized, the programmer can capture a scene using the `Bumblebee::Capture` function.

3) Secondary "Container" Structures

The `PPMImage` structure is responsible for encapsulating a two-dimensional image and all its properties. The class is able to read and write a portable pixel-map (PPM) file to the file system. Care was taken to ensure the image format is cross-platform.

The `Point` structure is used to contain point information. The `x`, `y`, and `z` members of the structure denote the point's location in 3-space. The `i`, `j`, and `d` members of the structure denote the row, column, and disparity (respectively) of the corresponding pixel in the right rectified image of scene. Because the same structure contains both the 2D and 3D location of the point, an inherent mapping is preserved between all points. If the location of a valid point is known in 2-space, the corresponding 3-space point can be found. The converse is also true.

The `Plane` structure's job is to encapsulate all the analysis results for the scene, such as how steep the plane is and the measure of kurtosis of the plane's fit to the points. The bounds of the plane are defined using points P1 through P4.

The `Texture` structure is meant to contain all the information necessary for OpenGL to texture an object. Its contents are essentially a stripped down version of the `PPMImage` class (and in fact, the texture class is constructed using a `PPMImage` object by calling `Texture::Compose`). It contains a texture identifier, the texturing data, and the height and width of the data.

The `Cam` structure contains the camera system's location and orientation when viewing the data in the point cloud viewer. The coordinate system is relative to OpenGL world coordinates.

The `Vector` structure is used for internal vector calculations, such as finding the dot product between two vectors.

The `HitList` structure is responsible for recording which plane was clicked on (when the user left clicks). The user can click on additional planes, which are subsequently added to the hit list, depending current on the mode of operation of the point cloud viewer.

4) Selection

OpenGL provides facilities for polygon selection. These facilities were utilized in the point cloud viewer's `Selection` function. First, selection mode is enabled with a call to `glRenderMode`. Next, the portion of the scene that the user clicked on is drawn to a separate buffer called the "selection view". OpenGL subsequently identifies which polygon was clicked on when `glRenderMode` is called again and the render mode is set back to its original state.

After the polygon that was clicked is identified, it is added to a "hit list". The hit list is specifically designed to keep track of which polygons have been selected. If sticky selection is disabled then there is a maximum of one polygon on the list. The program checks the hit list and colors those polygons blue when the scene is rendered.

5) Texturing

Texturing is the capability to overlay a 2D image onto a 3D polygon. The point cloud viewer employs texturing when drawing the 2D representation of the scene. In this case, the point cloud viewer draws several flat planes and overlays the Bumblebee's right camera image onto the planes. This presents the 2D representation of the scene in the same way it would be viewed with a conventional picture viewer. When the 2D representation of the scene is drawn it looks like a single plane textured with the Bumblebee's right camera image, but this is not the case in reality. In fact, multiple planes are drawn so that the user can select a specific section of the 2D representation of the scene.

C. Memory Management Considerations

Calculation speed was emphasized in the design of the point cloud viewer. The system is designed to do all analysis on a standard desktop computer, so trade-offs for speed are made in favor of memory size.

The `Cloud` and `PPMImage` structures both allocate dynamic memory. As a result, their constructors, destructors, assignment operators, and copy constructors have all been overloaded so that both classes avoid the deep-copy problem and are easily manageable. When designing the code, Standard Template Library (STL) classes were considered (for containing the point data) but ultimately rejected because most of the STL functionality was not needed and would inhibit speed performance. The assignment operators and copy constructors of both classes were thoroughly tested and contain no known problems.

D. Informal Speed Test

An informal speed test was conducted to obtain a quantifiable estimate of how much processing time the point cloud viewer requires to analyze a data set. Several data sets of varying size were used in the test. The selected data sets constitute typical data set sizes for the system and were opened from the hard drive (as opposed to direct capture with the Bumblebee).

The test was conducted using a "release" build of the application (as opposed to a "debug" build) to avoid performance decrease caused by debug-symbols in the executable. The test was performed on a standard desktop computer running Windows Vista with a 2.66 GHz processor and 2 GB of RAM. The point cloud viewer was compiled using Microsoft Visual Studio 2008.

The results are shown in Table 2 and Table 3. The explanation of the headers for Table 2 and Table 3 is as follows:

- "Number of points" is the number of 3D points contained in the point cloud.
- "File size" is the collective size of the portable pixel map and the associated 3D point cloud.
- "Open time" is the amount of time required to read the file from the hard drive.
- "Analysis time" is the amount of time required to perform the analysis on the data set.

Table 2: Results of an informal speed test of the point cloud viewer.

Number of divisions: **fifteen rows** and **fifteen columns**			
Number of points	*File size (bytes)*	*Open time (milliseconds)*	*Analysis time (milliseconds)*
68110	4,390,337	1109	63
95994	5,715,149	1515	125
120612	7,150,345	1875	297
214971	11,581,554	3203	828

Table 3: Results of an informal speed test of the point cloud viewer.

Number of divisions: **thirty rows** and **thirty columns**			
Number of points	*File size (bytes)*	*Open time (milliseconds)*	*Analysis time (milliseconds)*
68110	4,390,337	1093	235
95994	5,715,149	1500	438
120612	7,150,345	1875	985
214971	11,581,554	3219	3265

The results indicate that calculation duration generally increases with more row and column divisions. Calculation duration also increases with the number of points contained within the data set.

VI. USER STUDY

One of the notable strengths of the point cloud viewer is its ability to quantify the slope of a surface. An experiment was performed to compare surface angles estimated by humans with those determined by the point cloud viewer. The comparison quantifies the increase in accuracy that can be attributed to the use of the point cloud viewer. The hypothesis was that humans make relatively poor angle estimations. The questionnaire that was used to conduct the study is in Appendix B.

A. Experiment Design

The experiment consisted of 25 human subjects performing two tasks. The first task required the participant to sequentially view six 2D images (Figure 41 through Figure 46) and estimate the angle from the horizontal plane of a slanted wood pallet in each image. The scene is always presented from a bird's-eye view, and the pallet is slanted at a different angle in each image. The view that was presented to the participants was monocular, and not stereoscopic. No specific amount of precision was requested from the participants for the angle estimations. The camera system was perfectly level when each scene was captured, but this information was not explicitly conveyed to the participants.

The second task simultaneously presented the participant with three images of the wood pallet and required the subject to select the image with the pallet at the flattest angle. There was no time limit for providing answers in either task.

It is true that the human participants were at a disadvantage because they were presented with a monocular view of the scene while the computer was able to work with stereoscopic information. However, a remote operator would only have monocular view available so the disadvantage is realistic. (3D display technology such as a stereoscopic headset would remove this disadvantage, but such hardware is expensive and adds unnecessary complexity to the project.)

The true angle of the wood pallet was measured to compare with human estimations and point cloud viewer estimations. Trigonometry was used to obtain the true angle, by measuring two sides of the triangle formed by the wooden pallet's slant.

The blue tarp beneath the pallet was used to reduce glare created by the overhead lights in the room. (Some of the participants wondered why it was there.) The tarp was never explicitly mentioned in the experiment.

This experiment may appear to be trivial; however, the necessity to make rapid judgments on visual data during a descent is paramount to the project.

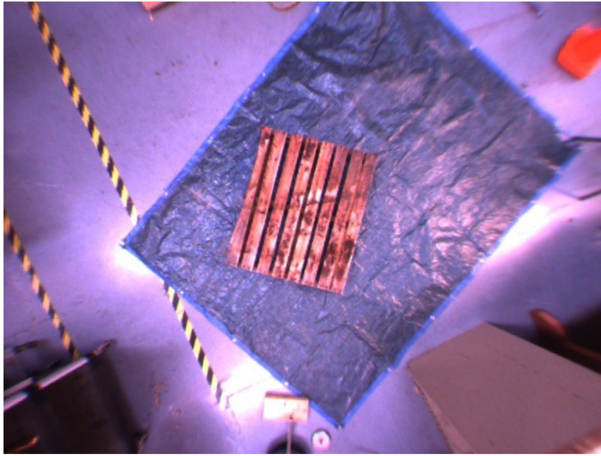


Figure 41: Wood pallet tilted at 00.00 degrees. Denoted as image B throughout the experiment.

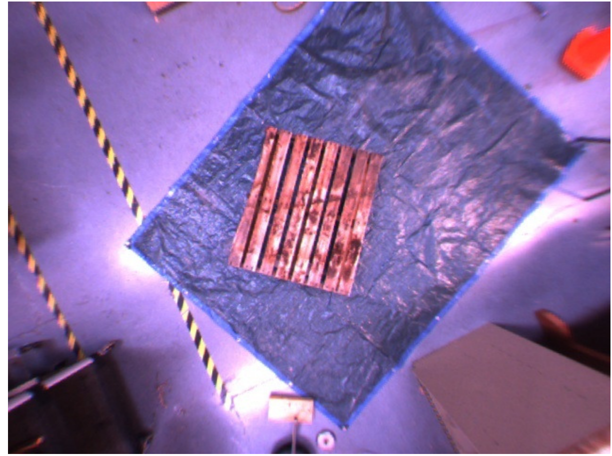


Figure 42: Wood pallet tilted at 04.10 degrees. Denoted as image E throughout the experiment.

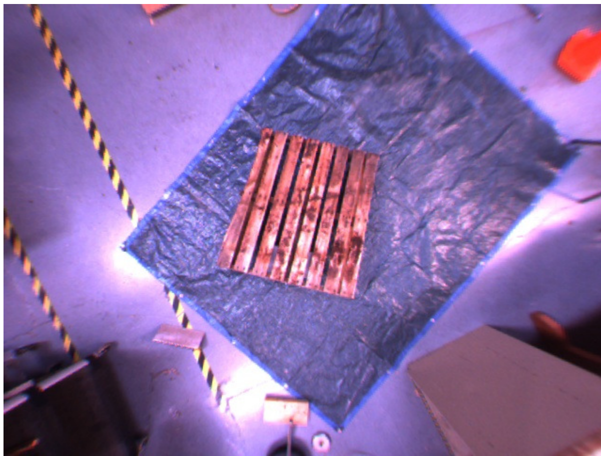


Figure 43: Wood pallet tilted at 13.07 degrees. Denoted as image F throughout the experiment.

Figure 44: Wood pallet tilted at 15.89 degrees. Denoted as image A throughout the experiment.

Figure 45: Wood pallet tilted at 21.29 degrees. Denoted as image D throughout the experiment.

Figure 46: Wood pallet tilted at 26.52 degrees. Denoted as image C throughout the experiment.

B. Results

The hypothesis was: human estimates of the slope of a surface pictured in a 2D image and viewed from above will be less accurate than the stereo based slope estimates for the same image using the point cloud viewer. The data that was collected from the experiment confirmed the original hypothesis. The results also indicated that humans perform better at comparing multiple angles than simply estimating quantitative angles.

Each image was assigned a label that is unrelated to the pallet's slope. The mapping between the label and the angle (which was not presented to participants of the study) is detailed in Table 4. These labels are used for convenience throughout the study.

Table 4: The label of each image used in the user study and the corresponding orientation of the woodn pallet.

Label	Angle
Image A	15.89
Image B	00.00
Image C	26.52
Image D	21.29
Image E	04.10
Image F	13.07

Table 5 shows the results of the user study. The correct answer of each question is included in the first row of each column for quick reference.

Table 5: The results of the user study. All angles are in degrees.

Participant	Image B (00.00°)	Image E (04.10°)	Image F (13.07°)	Image A (15.89°)	Image D (21.29°)	Image C (26.52°)	Comparison 1 (Answer = 3)	Comparison 2 (Answer = 2)
1	0	10	20	30	45	45	3	1
2	0	10	20	10	30	45	3	2
3	0	3	5	7	15	30	3	2
4	30	5	30	45	70	75	3	2
5	10	5	10	20	30	40	3	1
6	20	10	15	30	3	5	All equal	All equal
7	8	10	12	12	3	5	2	2
8	5	15	45	20	50	65	3	2
9	5	0	10	35	20	45	3	2
10	5	0	5	20	25	30	3	2
11	0	0	10	10	20	30	3	2
12	5	0	3	10	18	30	3	2
13	5	5	20	30	25	40	3	2
14	0	0	10	45	10	25	3	2
15	5	5	5	15	15	30	3	2
16	0	0	20	30	45	70	3	2
17	0	10	20	20	30	40	3	2
18	20	12	10	30	5	10	3	2
19	15	10	10	25	5	10	2	1
20	10	10	30	30	45	45	3	2
21	5	0	5	20	25	30	3	2
22	0	0	0	20	20	45	3	2
23	15	15	20	30	20	45	3	1
24	4	0	10	15	20	20	3	2
25	20	15	15	25	15	20	Not 2	2

The seventh and eighth questions of the user study asked the user to compare multiple images of the pallet (while each image had the pallet tilted at a different angle) and choose the flattest one. For question seven, 84% of the participants answered correctly. For question eight, 80% of the participants answered correctly. This presents a compelling argument that most people can make fairly good comparisons when asked to identify the minimum angle.

The divergence of the user's guesses from the true answer is presented in a tabular format in Table 6, and in a graphical format in Figure 47. A boxplot was chosen to represent the results of the study because it does not make an assumption about the underlying statistical distribution of the data. The box encloses the 25th to 75th percentile and is known as the inner quartile range. The whiskers enclose the 5th to 95th percentile. The "x" marks represent the minimum and maximum deviations, and the dot inside the inner quartile range represents the median.

Table 6: The divergence of user responses from the true angle. All angles are in degrees.

Participant	Image B (00.00°)	Image E (04.10°)	Image F (13.07°)	Image A (15.89°)	Image D (21.29°)	Image C (26.52°)
1	0.00	5.90	6.93	14.11	23.71	18.49
2	0.00	5.90	6.93	-5.89	8.71	18.49
3	0.00	-1.10	-8.07	-8.89	-6.29	3.49
4	30.00	0.90	16.93	29.11	48.71	48.49
5	10.00	0.90	-3.07	4.11	8.71	13.49
6	20.00	5.90	1.93	14.11	-18.29	-21.52
7	8.00	5.90	-1.07	-3.89	-18.29	-21.52
8	5.00	10.90	31.93	4.11	28.71	38.49
9	5.00	-4.10	-3.07	19.11	-1.29	18.49
10	5.00	-4.10	-8.07	4.11	3.71	3.49
11	0.00	-4.10	-3.07	-5.89	-1.29	3.49
12	5.00	-4.10	-10.07	-5.89	-3.29	3.49
13	5.00	0.90	6.93	14.11	3.71	13.49
14	0.00	-4.10	-3.07	29.11	-11.29	-1.52
15	5.00	0.90	-8.07	-0.89	-6.29	3.49
16	0.00	-4.10	6.93	14.11	23.71	43.49
17	0.00	5.90	6.93	4.11	8.71	13.49
18	20.00	7.90	-3.07	14.11	-16.29	-16.52
19	15.00	5.90	-3.07	9.11	-16.29	-16.52
20	10.00	5.90	16.93	14.11	23.71	18.49
21	5.00	-4.10	-8.07	4.11	3.71	3.49
22	0.00	-4.10	-13.07	4.11	-1.29	18.49
23	15.00	10.90	6.93	14.11	-1.29	18.49
24	4.00	-4.10	-3.07	-0.89	-1.29	-6.52
25	20.00	10.90	1.93	9.11	-6.29	-6.52

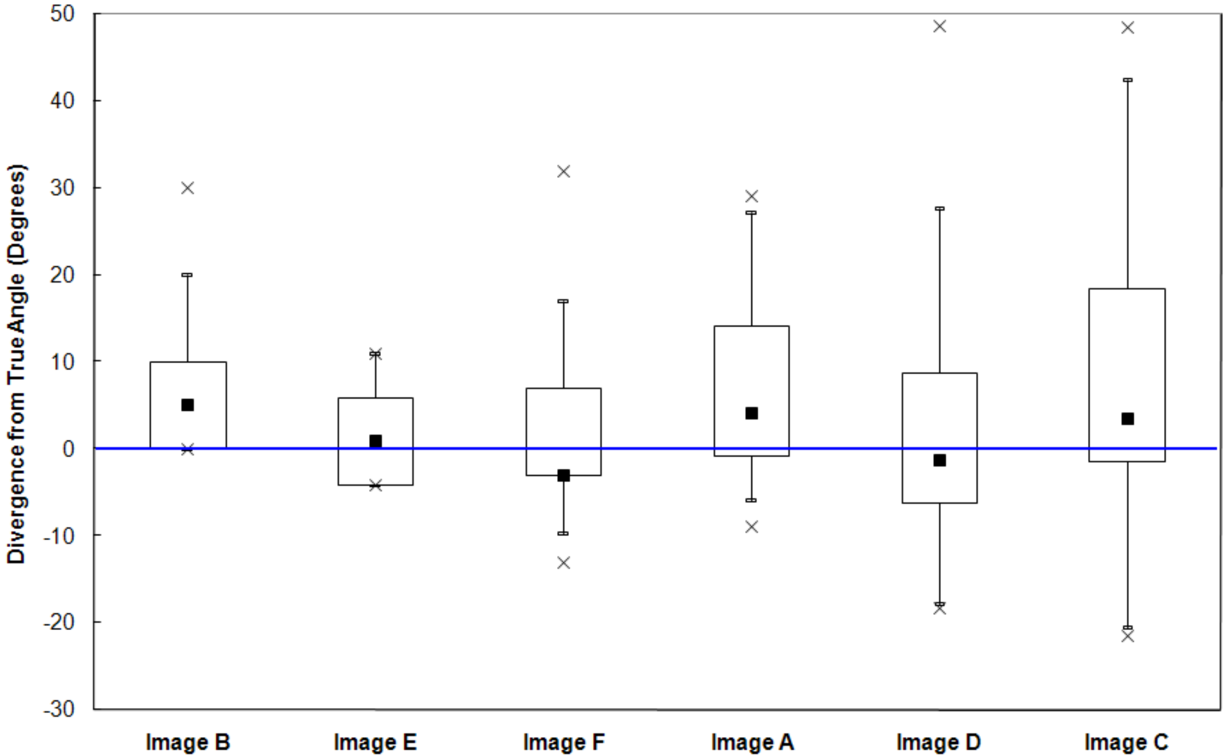


Figure 47: A boxplot which shows the divergence of angle estimations from the true angle. The box encloses the 25th to 75th percentile and is known as the inner quartile range. The whiskers enclose the 5th to 95th percentile. The "x" marks represent the minimum and maximum deviations, and the dot inside the inner quartile range represents the median.

Image C exemplifies the conclusion of the study quite well. It was the image with the highest angle (26.52°) and it has the largest inner quartile range. It is safe to assume that 50% of the time, a human cloud makes angle estimations to within ±20° of accuracy (because 20° is the inner quartile range for image C).

It is also interesting to note that 139 of the 150 total responses for angle estimation were a multiple of 5. That equates to 92.67% of the responses that were a multiple of 5, which suggests that humans have an implicit level of granularity for estimations. It should be noted that the participants were not asked to provide a specific level of precision for their answers.

The tethered payload of the UAV requires a (fairly) level landing site, so it is clear that the lack of accuracy in human angle estimations presents a difficult problem. Fortunately, the point cloud

viewer is much better at making these estimations. The angles were estimated using the point cloud viewer and the results are presented in Table 7.

Table 7: The point cloud viewer's estimations of the angles presented in the user study. All angles are in degrees.

	Truth	Point cloud viewer estimation	Divergence
Image B	00.00	01.51	-1.51
Image E	04.10	04.13	-0.03
Image F	13.07	12.89	0.18
Image A	15.81	15.26	0.55
Image D	21.29	21.51	-0.22
Image C	26.52	26.53	-0.01

The point cloud viewer's estimations were made by loading the scene into the viewer and selecting a section of the scene that lay on the pallet. The angle estimation was then recorded from the title bar of the application. The point cloud viewer correctly estimated the angle to within ±2°. This constitutes an immense increase in accuracy over monocular human estimation.

VII. CONCLUSIONS AND FUTURE WORK

The system described in this thesis is capable of using two 2D images to deduce 3D information for identifying potential landing sites for a semi-autonomous VTOL UAVs. The thesis also outlines the operation of a point cloud viewer program, which is used to visualize and rapidly identify potential landing sites in a quantitative manner. The program's ability to identify the slope of an object was tested against human qualitative assessment of the same task. Significant improvement in accuracy of angle estimation was shown when using the program. Overall, the work provides a valuable contribution towards improving UAV navigation.

Possible avenues of research have become evident as a result of this project. As discussed in chapter II, capturing images in the visual spectrum requires different image processing techniques for different times of day. Images captured during the day must be processed differently from those captured at night. The Bumblebee software can partially compensate for this difficulty because the camera system's parameters can be optimized for various lighting conditions [6]. Even so, nighttime operation is impossible with the system's current configuration. Future work could focus on ensuring the system can be used under any lighting conditions. There are several possible ways to accomplish this. One example would be to use artificial illumination (perhaps with high-power light emitting diodes) to ensure that the scene is visible. These solutions will inevitably raise other concerns such as power consumption.

The point cloud viewer was designed with the intention of being used as an endgame application. It is a medium that would provide complete control to the mission supervisor. It is clear that the program has not yet reached a level of development that is worthy of deployment. It is a beta version of sorts – almost all of the functionality is accessed through keyboard commands. Future work could focus on honing the design of the user interface to make the application ready for deployment.

Currently, the system will occasionally generate false positive and false negative landing sites. Future work could focus on using mathematical methods to better identify landing sites.

Solutions to all of these issues are feasible. Hopefully there will be some interesting future additions to the project.

VIII. WORKS CITED

[1] R. Parasuraman, T. B. Sheridan, and C. D. Wickens, "A model for types and levels of human interaction with automation," *Systems, Man and Cybernetics, Part A, IEEE Transactions on*, vol. 30, no. 3, pp. 286-297, 2000.

[2] S. Lee, et al., "A 3D IR camera with variable structured light for home service robots," in *Robotics and Automation. Proceedings of the 2005 IEEE International Conference on*, Barcelona, Spain, April 2005, pp. 1859-1864.

[3] G. Vosselman, B. G. H. Gorte, G. Sithole, and T. Rabbani, "Recognising structure in laser scanner point clouds," in *Proceedings of Conference on Laser Scanners for Forest and Landscape Assessment and Instruments*, Freiburg, Germany, 2004.

[4] C. S. Sharp, O. Shakernia, and S. S. Sastry, "A vision system for landing an unmanned aerial vehicle," in *Robotics and Automation, IEEE International Conference on*, 2001, pp. 1720-1727.

[5] T. Schenk and B. Csatho, "Fusion of LIDAR data and aerial imagery for a more complete surface description," *International Archives of Photogrammetry, Remote Sensing, and Spatial Information Sciences*, vol. 34, pp. 310-317, 2002.

[6] Point Grey Research Incorporated, "Triclops Stereo Vision System Manual," User guide and reference manual, 2003.

[7] F. Remondino, "From point cloud to surface: the modeling and visualization problem," in *Proceedings of International Workshop on Visualization and Animation of Reality-Based 3D Models*, 2003.

[8] GameDev.net, LLC. (2008, Jul.) NeHe Productions. [Online]. http://nehe.gamedev.net/

[9] A. Culhane, "Development of an obstacle detection system for human supervisory control of a UAV in urban environments," M.S. thesis, Department of Mechanical Engineering, Virginia Tech, Blacksburg, VA, USA, 2007.

[10] A. Sharkasi, "Stereo vision based aerial mapping using GPS and internal sensors," M.S. thesis, Department of Mechanical Engineering, Virginia Tech, Blacksburg, VA, USA, 2008.

[11] N. Ayache and C. Hansen, "Rectification of images for binocular and trinocular stereo vision," in *Pattern Recognition, Proceedings of 9th International Conference on*, Rome, Italy, Nov. 1988, pp. 11-16.

[12] B. Hulin and S. Schüßler, "Concepts for day-night stereo obstacle detection in the pantograph gauge," in *Industrial Informatics, Proceedings of 5th IEEE International Conference on*, vol. 1, Munich, Germany, 2007, pp. 449-454.

[13] Point Grey Research, Inc. (2008, Jun.) Point Grey Research. [Online]. http://www.ptgrey.com/products/bumblebee2/bumblebee2_xb3_datasheet.pdf

[14] D. Eberly. (2008, Jun.) Geometric Tools. [Online]. http://www.geometrictools.com/

APPENDIX A: POINT CLOUD VIEWER CODE

A. Driver.cpp

```
/*
 *          This Code Was Created By Jeff Molofee 2000
 *          A HUGE Thanks To Fredric Echols For Cleaning Up
 *          And Optimizing The Base Code, Making It More Flexible!
 *          If You've Found This Code Useful, Please Let Me Know.
 *          Visit My Site At nehe.gamedev.net.
 *          This code was heavily modified by Dylan Klomparens
 *          in 2008 for use in a masters thesis.
 */

#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 <ctime>
#include <iostream>
#include <fstream>
#include "cloud.h"
#include "render.h"
#include "bumblebee.h"
using namespace std;

HDC hDC = NULL; // Private GDI Device Context
HGLRC hRC = NULL; // Permanent Rendering Context
HWND hWnd = NULL; // Holds Our Window Handle
HINSTANCE hInstance; // Holds The Instance Of The Application
int MouseX;
int MouseY;
```

```
Representation Rep = REPRESENTATION_IS_2D;
bool Overlay = true;
bool ToggleSelection = true;
ColorType CurrentColorType = THRESHOLD;

bool keys[256]; // Array Used For The Keyboard Routine
bool active = TRUE; // Window Active Flag Set To TRUE By Default
bool fullscreen = FALSE; // Fullscreen flag not set to fullscreen mode by default

LRESULT     CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);

void KillGLWindow()                                 // Properly Kill The Window
{
      if(fullscreen)                                      // Are We In Fullscreen Mode?
      {
            ChangeDisplaySettings(NULL, 0);           // If So Switch Back To The Desktop
            ShowCursor(TRUE);                         // Show Mouse Pointer
      }
      if (hRC)                                            // Do We Have A Rendering 
Context?
      {
            wglMakeCurrent(NULL, NULL);               // Are We Able To Release The DC 
And RC Contexts?
            wglDeleteContext(hRC);                    // Are We Able To Delete The RC?
            hRC = NULL;                                     // Set RC To NULL
      }
      if(hDC && !ReleaseDC(hWnd, hDC))              // Are We Able To Release The DC
            hDC = NULL;                                     // Set DC To NULL
      if(hWnd && !DestroyWindow(hWnd))              // Are We Able To Destroy The Window?
            hWnd = NULL;                                    // Set hWnd To NULL
      UnregisterClass("OpenGL", hInstance);         // Are We Able To Unregister Class
      hInstance = NULL;                                 // Set hInstance To NULL
}

/*    This Code Creates Our OpenGL Window.  Parameters Are:                *
 *    title             - Title To Appear At The Top Of The Window               *
 *    width             - Width Of The GL Window Or Fullscreen Mode              *
 *    height                  - Height Of The GL Window Or Fullscreen Mode       *
 *    bits              - Number Of Bits To Use For Color (8/16/24/32)     *
 *    fullscreenflag    - Use Fullscreen Mode (TRUE) Or Windowed Mode (FALSE) */
```

```
BOOL CreateGLWindow(LPCSTR title, int width, int height, int bits, bool fullscreenflag)
{
	GLuint		PixelFormat;			// Holds The Results After Searching For A Match
	WNDCLASS	wc;				// Windows Class Structure
	DWORD		dwExStyle;			// Window Extended Style
	DWORD		dwStyle;			// Window Style
	RECT		WindowRect;			// Grabs Rectangle Upper Left / Lower Right Values
	WindowRect.left=(long)0;			// Set Left Value To 0
	WindowRect.right=(long)width;		// Set Right Value To Requested Width
	WindowRect.top=(long)0;				// Set Top Value To 0
	WindowRect.bottom=(long)height;			// Set Bottom Value To Requested Height

	fullscreen=fullscreenflag;			// Set The Global Fullscreen Flag

	hInstance		= GetModuleHandle(NULL);			// Grab An Instance For Our
Window
	wc.style		= CS_HREDRAW | CS_VREDRAW | CS_OWNDC;	// Redraw On Size, And Own DC For
Window.
	wc.lpfnWndProc		= (WNDPROC) WndProc;			// WndProc Handles Messages
	wc.cbClsExtra		= 0;					// No Extra Window Data
	wc.cbWndExtra		= 0;					// No Extra Window Data
	wc.hInstance		= hInstance;				// Set The Instance
	wc.hIcon		= LoadIcon(NULL, IDI_WINLOGO);		// Load The Default Icon
	wc.hCursor		= LoadCursor(NULL, IDC_ARROW);		// Load The Arrow Pointer
	wc.hbrBackground	= NULL;					// No Background
Required For GL
	wc.lpszMenuName		= NULL;						// We Don't Want
A Menu
	wc.lpszClassName	= "OpenGL";				// Set The Class Name

	if (!RegisterClass(&wc))					// Attempt To Register
The Window Class
	{
		MessageBox(NULL, "Failed To Register The Window Class.", "ERROR", MB_OK|MB_ICONEXCLAMATION);
		return FALSE;						// Return FALSE
	}

	if (fullscreen)							// Attempt
Fullscreen Mode?
	{
		DEVMODE dmScreenSettings;				// Device Mode
```

```
            memset(&dmScreenSettings,0,sizeof(dmScreenSettings)); // Makes Sure Memory's Cleared
            dmScreenSettings.dmSize=sizeof(dmScreenSettings);           // Size Of The Devmode Structure
            dmScreenSettings.dmPelsWidth  = width;                      // Selected Screen Width
            dmScreenSettings.dmPelsHeight = height;                     // Selected Screen Height
            dmScreenSettings.dmBitsPerPel = bits;                             // Selected Bits Per Pixel
            dmScreenSettings.dmFields=DM_BITSPERPEL|DM_PELSWIDTH|DM_PELSHEIGHT;

            // Try To Set Selected Mode And Get Results.  NOTE: CDS_FULLSCREEN Gets Rid Of Start Bar.
            if (ChangeDisplaySettings(&dmScreenSettings,CDS_FULLSCREEN)!=DISP_CHANGE_SUCCESSFUL)
            {
                  // If The Mode Fails, Offer Two Options.  Quit Or Use Windowed Mode.
                  if (MessageBox(NULL, "The requested fullscreen mode is not supported by your video card.
Would you like to use windowed mode instead?", "Point-cloud viewer", MB_YESNO|MB_ICONEXCLAMATION) == IDYES)
                  {
                        fullscreen = FALSE;           // Windowed Mode Selected.  Fullscreen = FALSE
                  }
                  else
                  {
                        // Pop Up A Message Box Letting User Know The Program Is Closing.
                        MessageBox(NULL, "Program will now close.", "Point-cloud viewer error",
MB_OK|MB_ICONSTOP);
                        return FALSE;                                                      // Return FALSE
                  }
            }
      }

      if (fullscreen)                                                                      // Are We Still
In Fullscreen Mode?
      {
            dwExStyle=WS_EX_APPWINDOW;                                                // Window Extended
Style
            dwStyle=WS_POPUP;                                                         // Windows Style
            ShowCursor(FALSE);                                                              // Hide Mouse
Pointer
      }
      else
      {
            dwExStyle=WS_EX_APPWINDOW | WS_EX_WINDOWEDGE;               // Window Extended Style
            dwStyle=WS_OVERLAPPEDWINDOW;                                      // Windows Style
      }
```

```
	AdjustWindowRectEx(&WindowRect, dwStyle, FALSE, dwExStyle);		// Adjust Window To True Requested
Size

	// Create The Window
	if (!(hWnd=CreateWindowEx(	dwExStyle,							// Extended Style For
The Window
								"OpenGL",							// Class
Name
								title,								//
Window Title
								dwStyle |							// Defined
Window Style
								WS_CLIPSIBLINGS |					// Required
Window Style
								WS_CLIPCHILDREN,					// Required
Window Style
								0, 0,								// Window
Position
								WindowRect.right-WindowRect.left,	// Calculate Window
Width
								WindowRect.bottom-WindowRect.top,	// Calculate Window
Height
								NULL,								// No
Parent Window
								NULL,								// No Menu
								hInstance,							// Instance
								NULL)))								//
Dont Pass Anything To WM_CREATE
	{
		KillGLWindow();								// Reset The Display
		MessageBox(NULL, "Window creation error.", "Point-cloud viewer error",
MB_OK|MB_ICONEXCLAMATION);
		return FALSE;								// Return FALSE
	}

	static	PIXELFORMATDESCRIPTOR pfd=				// pfd Tells Windows How We Want Things
To Be
	{
		sizeof(PIXELFORMATDESCRIPTOR),				// Size Of This Pixel Format Descriptor
		1,											// Version Number
		PFD_DRAW_TO_WINDOW |						// Format Must Support Window
```

```
		PFD_SUPPORT_OPENGL |					// Format Must Support OpenGL
		PFD_DOUBLEBUFFER,					// Must Support Double Buffering
		PFD_TYPE_RGBA,						// Request An RGBA Format
		bits,							// Select Our Color Depth
		0, 0, 0, 0, 0, 0,					// Color Bits Ignored
		0,							// No Alpha Buffer
		0,							// Shift Bit Ignored
		0,							// No Accumulation Buffer
		0, 0, 0, 0,						// Accumulation Bits Ignored
		16,							// 16Bit Z-Buffer (Depth
Buffer)
		0,							// No Stencil Buffer
		0,							// No Auxiliary Buffer
		PFD_MAIN_PLANE,						// Main Drawing Layer
		0,							// Reserved
		0, 0, 0							// Layer Masks Ignored
	};

	if(!(hDC=GetDC(hWnd)))						// Did We Get A Device Context?
	{
		KillGLWindow();						// Reset The Display
		MessageBox(NULL, "Can't Create A GL Device Context.", "Point-cloud viewer error",
MB_OK|MB_ICONEXCLAMATION);
		return FALSE;						// Return FALSE
	}

	if (!(PixelFormat=ChoosePixelFormat(hDC,&pfd)))	// Did Windows Find A Matching Pixel Format?
	{
		KillGLWindow();						// Reset The Display
		MessageBox(NULL, "Can't Find A Suitable PixelFormat.", "Point-cloud viewer error",
MB_OK|MB_ICONEXCLAMATION);
		return FALSE;						// Return FALSE
	}

	if(!SetPixelFormat(hDC,PixelFormat,&pfd))		// Are We Able To Set The Pixel Format?
	{
		KillGLWindow();						// Reset The Display
		MessageBox(NULL, "Can't Set The PixelFormat.", "Point-cloud viewer error",
MB_OK|MB_ICONEXCLAMATION);
		return FALSE;						// Return FALSE
	}
```

```
	if (!(hRC=wglCreateContext(hDC)))				// Are We Able To Get A Rendering Context?
	{
		KillGLWindow();						// Reset The Display
		MessageBox(NULL, "Can't Create A GL Rendering Context.", "Point-cloud viewer error",
MB_OK|MB_ICONEXCLAMATION);
		return FALSE;						// Return FALSE
	}

	if(!wglMakeCurrent(hDC,hRC))					// Try To Activate The Rendering Context
	{
		KillGLWindow();						// Reset The Display
		MessageBox(NULL, "Can't Activate The GL Rendering Context.", "Point-cloud viewer error",
MB_OK|MB_ICONEXCLAMATION);
		return FALSE;						// Return FALSE
	}

	InitGL();
	ShowWindow(hWnd,SW_SHOW);					// Show The Window
	SetForegroundWindow(hWnd);					// Slightly Higher Priority
	SetFocus(hWnd);							// Sets Keyboard Focus To The
Window
	ResizeScene(width, height);					// Set Up Our Perspective GL Screen
	return TRUE;							// Success
}

LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
	switch (uMsg) // Check For Windows Messages
	{
		case WM_ACTIVATE: // Watch For Window Activate Message
		{
			// LoWord Can Be WA_INACTIVE, WA_ACTIVE, WA_CLICKACTIVE,
			// The High-Order Word Specifies The Minimized State Of The Window Being Activated Or
Deactivated.
			// A NonZero Value Indicates The Window Is Minimized.
			if ((LOWORD(wParam) != WA_INACTIVE) && !((BOOL)HIWORD(wParam)))
				active=TRUE;				// Program Is Active
			else
				active=FALSE;				// Program Is No Longer Active
```

```
		return 0;								// Return To The Message Loop
	}

	case WM_MOUSEMOVE:
	{
		const float TranslationScaleFactor = 500;
		const float RotationScaleFactor = 2;
		static POINTS LastPoint;
		POINTS CurrentPoint = MAKEPOINTS(lParam);
		if(wParam & MK_LBUTTON) // Rotation
		{
			if(wParam & MK_SHIFT) // Roll and yaw
			{
				Camera.pitch -= (LastPoint.y - CurrentPoint.y) / RotationScaleFactor;
				Camera.roll -= (LastPoint.x - CurrentPoint.x) / RotationScaleFactor;
			}
			else // Pitch and yaw
			{
				Camera.pitch -= (LastPoint.y - CurrentPoint.y) / RotationScaleFactor;
				Camera.yaw += (LastPoint.x - CurrentPoint.x) / RotationScaleFactor;
			}
		}
		else if(wParam & MK_RBUTTON) // Translation
		{
			if(wParam & MK_SHIFT) // Zoom and left/right
			{
				Camera.x -= (LastPoint.x - CurrentPoint.x) / TranslationScaleFactor;
				Camera.z -= (LastPoint.y - CurrentPoint.y) / TranslationScaleFactor;
			}
			else // Up/down and left/right
			{
				Camera.x -= (LastPoint.x - CurrentPoint.x) / TranslationScaleFactor;
				Camera.y -= (LastPoint.y - CurrentPoint.y) / TranslationScaleFactor;
			}
		}
		LastPoint = CurrentPoint;
		return 0;
	}

	case WM_MOUSEWHEEL:
	{
```

```
		const float ZoomScaleFactor = 150;
		Camera.z -= GET_WHEEL_DELTA_WPARAM(wParam) / ZoomScaleFactor;
		return 0;
	}

	case WM_SYSCOMMAND:						// Intercept System Commands
	{
		switch(wParam)
		{
			case SC_SCREENSAVE:				// Screensaver Trying To Start?
			case SC_MONITORPOWER:			// Monitor Trying To Enter Powersave?
			return 0;						// Prevent From Happening
		}
		break;
	}
	case WM_LBUTTONDOWN:
	{
	MouseX = LOWORD(lParam);
		MouseY = HIWORD(lParam);
		string ItemInfo = Selection(Rep, ToggleSelection);
		SetWindowTextA(hWnd, (LPCSTR)ItemInfo.c_str());
		return 0;
	}

	case WM_CLOSE:							// Did We Receive A Close Message?
	{
		PostQuitMessage(0);					// Send A Quit Message
		return 0;							// Jump Back
	}

	case WM_KEYDOWN:						// Is A Key Being Held Down?
	{
		keys[wParam] = TRUE;				// If So, Mark It As TRUE
		return 0;							// Jump Back
	}

	case WM_KEYUP:							// Has A Key Been Released?
	{
		keys[wParam] = FALSE;				// If So, Mark It As FALSE
		return 0;							// Jump Back
	}
```

```
		case WM_SIZE:							// Resize The OpenGL Window
		{
			ResizeScene(LOWORD(lParam),HIWORD(lParam));  // LoWord=Width, HiWord=Height
			return 0;						// Jump Back
		}
	}

	// Pass All Unhandled Messages To DefWindowProc
	return DefWindowProc(hWnd,uMsg,wParam,lParam);
}

int WINAPI WinMain(	HINSTANCE	hInstance,			// Instance
			HINSTANCE	hPrevInstance,		// Previous Instance
			LPSTR		lpCmdLine,		// Command Line Parameters
			int			nCmdShow)		// Window Show State
{
	#if REDIRECT_COUT
	ofstream Out("cout.txt");
	cout.rdbuf(Out.rdbuf());
	#endif
	#if BUMBLEBEE_INSTALLED
	InitializeBumblebee();
	#endif
	time_t LastKeyPress = time(0);

	MSG msg; // Windows Message Structure
	BOOL done = FALSE; // Bool Variable To Exit Loop

	// Create Our OpenGL Window
	if(!CreateGLWindow("Dylan's Point Cloud Viewer", 640, 480, 16, fullscreen))
		return 0; // Quit If Window Was Not Created

	while(!done)							// Loop That Runs While done=FALSE
	{
		if (PeekMessage(&msg,NULL,0,0,PM_REMOVE)) // Is There A Message Waiting?
		{

			if (msg.message==WM_QUIT)				// Have We Received A Quit Message?
			{
				done=TRUE;					// If So done=TRUE
```

```
			}
			else									// If Not, Deal With Window
Messages
			{
				TranslateMessage(&msg);				// Translate The Message
				DispatchMessage(&msg);				// Dispatch The Message
			}
		}
		else										// If There Are No Messages
		{
			// Draw The Scene.  Watch For ESC Key And Quit Messages From DrawGLScene()
			if((active && !DrawScene(Rep, Overlay)) || keys[VK_ESCAPE]) // Active?  Was There A Quit
Received?
			{
				done=TRUE;							// ESC or DrawGLScene Signalled A
Quit
			}
			else									// Not Time To Quit, Update Screen
			{
				SwapBuffers(hDC);					// Swap Buffers (Double Buffering)
			}

			// X, Y, Z
			if(keys[VK_RIGHT])
				Camera.x -= 0.3f;
			if(keys[VK_LEFT])
				Camera.x += 0.3f;
			if(keys[VK_UP])
				Camera.y -= 0.3f;
			if(keys[VK_DOWN])
				Camera.y += 0.3f;
			if(keys['Z'])
				Camera.z -= 0.3f;
			if(keys['X'])
				Camera.z += 0.3f;

			if(keys['D'])
				Camera.yaw -= 3;
			if(keys['A'])
				Camera.yaw += 3;
			if(keys['S'])
```

```
		Camera.pitch -= 3;
	if(keys['W'])
		Camera.pitch += 3;
	if(keys['E'])
		Camera.roll -= 3;
	if(keys['Q'])
		Camera.roll += 3;

	if(keys['0'])
	{
		Camera.x = 0;
		Camera.y = 0;
		Camera.z = 0;
		Camera.roll = 0;
		Camera.pitch = 0;
		Camera.yaw = 0;
		Camera.perspective = true;
	}
	if(keys['2'])
	{
		Rep = REPRESENTATION_IS_2D;
	}
	if(keys['3'])
	{
		Rep = REPRESENTATION_IS_3D;
	}
	if(keys['5'])
	{
		if(time(0) - LastKeyPress >= 2)
		{
			Overlay = !Overlay;
			LastKeyPress = time(0);
		}
	}
	if(keys['P'])
	{
		if(time(0) - LastKeyPress >= 2)
		{
			ToggleProjection();
			LastKeyPress = time(0);
		}
```

```
}
if(keys['T'])
{
      if(time(0) - LastKeyPress >= 2)
      {
            ToggleSelection = !ToggleSelection;
            LastKeyPress = time(0);
      }
}
if(keys['B'])
{
      if(time(0) - LastKeyPress >= 2)
      {
            ShowBestFit = !ShowBestFit;
            LastKeyPress = time(0);
      }
}
#if BUMBLEBEE_INSTALLED
if(keys['C']) // Capture a cloud
{
      if(time(0) - LastKeyPress >= 2)
      {
            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);
            LastKeyPress = time(0);
      }
}
#endif
if(keys['I']) // Input a cloud
{
      keys['I'] = FALSE;
```

```
			if(time(0) - LastKeyPress >= 2)
			{
				OPENFILENAME ofn; // common dialog box structure
				char szFile[1000];      // buffer for file name
				// Initialize OPENFILENAME
				ZeroMemory(&ofn, sizeof(ofn));
				ofn.lStructSize = sizeof(ofn);
				ofn.hwndOwner = 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();
					C->Open((char*)(ofn.lpstrFile));
					C->ComposeGrid(12, 12);
					// Test the assignment operator...
					Cloud* D = new Cloud();
					*D = *C;
					delete C;
					Clouds.push_back(D);
					ColorScene(CurrentColorType, false);
				}
				LastKeyPress = time(0);
			}
		}
		if(keys['O']) // Output a cloud
		{
```

```
            if(time(0) - LastKeyPress >= 2)
            {
                  if(!Clouds.empty())
                        Clouds[0]->Save();
                  LastKeyPress = time(0);
            }
      }
      if(keys['M']) // Select the acceptable planes
      {
            if(time(0) - LastKeyPress >= 2)
            {
                  ColorScene(CurrentColorType, true);
                  LastKeyPress = time(0);
            }
      }
      if(keys[VK_F1])
      {
            if(time(0) - LastKeyPress >= 2)
            {
                  CurrentColorType = SLOPE;
                  ColorScene(CurrentColorType, false);
                  LastKeyPress = time(0);
            }
      }
      if(keys[VK_F2])
      {
            if(time(0) - LastKeyPress >= 2)
            {
                  CurrentColorType = AVERAGE_ANGLE;
                  ColorScene(CurrentColorType, false);
                  LastKeyPress = time(0);
            }
      }
      if(keys[VK_F3])
      {
            if(time(0) - LastKeyPress >= 2)
            {
                  CurrentColorType = THRESHOLD;
                  ColorScene(CurrentColorType, false);
                  LastKeyPress = time(0);
            }
```

```
                        }
                        if(keys[VK_F4])
                        {
                                if(time(0) - LastKeyPress >= 2)
                                {
                                        CurrentColorType = KURTOSIS;
                                        ColorScene(CurrentColorType, false);
                                        LastKeyPress = time(0);
                                }
                        }
                }
        }

        for(vector<Cloud*>::iterator i = Clouds.begin(); i != Clouds.end(); i++)
                delete *i;
        KillGLWindow(); // Kill The Window
        #if ANALYZE_MEMORY
        _CrtDumpMemoryLeaks();
        #endif
        #if REDIRECT_COUT
        Out.close();
        #endif
        return int(msg.wParam); // Exit The Program
}
```

B. 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
```

C. 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++)
        {
            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
```

D. 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.
};

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

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

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;
};

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 Clear();
	void ZeroParameters();
	void DeleteBestFitPlanes();
	~Cloud();

	long PointCount;
	Point* Points;
	PPMImage Image;
	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;
};

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);
```

E. Cloud.cpp

```
#include "cloud.h"

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

Point::Point() : x(0), y(0), z(0), r(0), g(0), b(0), i(0), j(0), d(0)
{

}

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);
}

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);
		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)
{

}

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;
}

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;
	DeltaX = x.DeltaX;
	DeltaY = x.DeltaY;
	Columns = x.Columns;
	Rows = x.Rows;
```

```
	Image = x.Image;
	Hits = x.Hits;
	Tex = x.Tex;
	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();
	ifstream f(FileName.c_str(), ios_base::binary);
	if(f.fail())
	{
		f.close();
		return false;
	}
```

```
	string Buffer;
	getline(f, Buffer); // Read in "P6"
	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 >> 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;
}

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];
	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;
            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.
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.)
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];
			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;
}

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;
            }
      }
}

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);
}

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;
}

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()
{
      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;
```

```
}
```

F. Render.h

```
#pragma once

#ifdef WIN32
#include <windows.h>
#endif
#include <GL/gl.h>
#include <GL/glu.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")

struct Cam
{
	Cam();
	float x, y, z, roll, pitch, yaw;
	bool perspective;
};

enum ColorType
{
	TOP_PICKS,
	SLOPE,
	AVERAGE_ANGLE,
	THRESHOLD,
	KURTOSIS,
};

enum Representation
{
	REPRESENTATION_IS_2D,
	REPRESENTATION_IS_3D,
};
```

```
void ColorScene(ColorType Type, bool SelectAcceptable);
int DrawScene(Representation R, bool Overlay);
int InitGL();
void ResizeScene(GLsizei width, GLsizei height);
void ToggleProjection();
string Selection(Representation R, bool Toggle);

extern Cam Camera;
extern vector<Cloud*> Clouds;
extern bool ShowBestFit;
extern int MouseX, MouseY;
```

G. Render.cpp

```
#include "render.h"

Cam Camera;
vector<Cloud*> Clouds;
bool ShowBestFit = true;

Cam::Cam() : x(0), y(0), z(0), roll(0), pitch(0), yaw(0), perspective(true)
{

}

void ColorScene(ColorType Type, bool SelectAcceptable)
{
	const float SteepnessThreshold = 25; // degrees
	const float MaxAdjacentThreshold = 25; // degrees
	const int TopPickCount = 3;
	for(vector<Cloud*>::iterator i = Clouds.begin(); i != Clouds.end(); i++)
	{
		Cloud* C = (*i);
		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(Type == TOP_PICKS)
				{
					// TODO: implement this...
					// list<Plane*>
				}
				else if(Type == SLOPE || Type == AVERAGE_ANGLE || Type == KURTOSIS)
				{
					float Scale;
					if(Type == SLOPE)
						Scale = P->AngleFromZ / 90; // Show the slope as a red gradient
					else if(Type == AVERAGE_ANGLE)
						Scale = P->AverageAdjacent / 180; // Show the max angle of an adjacent
planes as a red gradient
					else if(Type == KURTOSIS)
```

```
                                Scale = P->Kurtosis;
                        if(Scale > 1) Scale = 1;
                        if(Scale < 0) Scale = 0;
                        float Color = 1 - 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(Type == THRESHOLD)
                {
                        bool Acceptable = false;
                        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;
                }
            }
        }
//          C->Image.WritePPM("Overlay.ppm");
    }
}

// Note: this function is for debug purposes only.
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;
	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();
				if(Draw_3D_Overlay) DrawBoundedElements(int(y), int(x), LowerX * Width, UpperX *
Width, LowerY * Height, UpperY * Height);
				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, bool HighlightSelection = true)
{
    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(p->Hit && HighlightSelection)
				{
					// Draw the plane as blue to indicate that it has been selected.
					glColor3f(0, 0, 1);
					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);
					glEnd();
					glColor3f(1, 1, 1);
				}
				else
				{
					// Draw the planes with no texture (only color). Do not apply color if we're
drawing for selection.
					glBegin(GL_QUADS);
					if(!Selection) glColor3f(p->P1.r, p->P1.g, p->P1.b); glVertex3f(p->P1.x, p-
>P1.y, p->P1.z);
					if(!Selection) glColor3f(p->P2.r, p->P2.g, p->P2.b); glVertex3f(p->P2.x, p-
>P2.y, p->P2.z);
					if(!Selection) glColor3f(p->P3.r, p->P3.g, p->P3.b); glVertex3f(p->P3.x, p-
>P3.y, p->P3.z);
					if(!Selection) glColor3f(p->P4.r, p->P4.g, p->P4.b); glVertex3f(p->P4.x, p-
>P4.y, p->P4.z);
					glEnd();
				}
			}
		}
	}
}

// Here's where we do all the drawing
int DrawScene(Representation R, bool Overlay)
{
	glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Clear screen and depth buffer
	glLoadIdentity(); // Reset the current modelview 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);
	if(R == REPRESENTATION_IS_2D)
		DrawRaw(false, Overlay);
	else if(R == REPRESENTATION_IS_3D)
	{
		DrawCloud();
		if(ShowBestFit)
			DrawPlanes();
	}
	return TRUE;
}

// All setup for OpenGL goes here
int InitGL()
{
	glShadeModel(GL_SMOOTH); // Enable smooth shading
	glClearColor(0.0f, 0.0f, 0.0f, 0.5f); // Black background
	glClearDepth(1.0f); // Depth buffer setup
	glEnable(GL_DEPTH_TEST); // Enables depth testing
	glDepthFunc(GL_LEQUAL); // The type of depth testing to do
	glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST); // Really nice perspective calculations
	return TRUE;
}

// Resize and initialize the GL window
void ResizeScene(GLsizei width, GLsizei height)
{
	if(height == 0) // Prevent a divide by zero by making height equal 1
		height = 1;

	glViewport(0, 0, width, height); // Reset the current viewport

	glMatrixMode(GL_PROJECTION); // Select the projection matrix
	glLoadIdentity(); // Reset the projection matrix

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

	glMatrixMode(GL_MODELVIEW); // Select the modelview matrix
	glLoadIdentity(); // Reset the modelview matrix
}
```

```
void ToggleProjection()
{
	// Toggle the perspective between orthographic and normal.
	Camera.perspective = !Camera.perspective;

	// Select the projection matrix and clear it out
	glMatrixMode(GL_PROJECTION);

	// Choose the appropriate projection based on the currently toggled mode
	if(Camera.perspective)
	{
		// Set the perspective with the appropriate aspect ratio
		glFrustum(-1.0, 1.0, -1.0, 1.0, .1, 100);
	}
	else
	{
		// Set up an orthographic projection with the same near clip plane
		glOrtho(-1.0, 1.0, -1.0, 1.0, .1, 100);
	}

	// Select modelview matrix and clear it out
	glMatrixMode(GL_MODELVIEW);
}

// Known issues:
// 2D representation selection will only work with first cloud in the Clouds list.
string Selection(Representation R, 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++;
                        }

      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(R == REPRESENTATION_IS_2D)
            DrawRaw(true, false);
      else if(R == REPRESENTATION_IS_3D)
            DrawPlanes(true, false);

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

      if(R == 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 << "Slope: " << p->AngleFromZ << ", max adjacent: " << p->AverageAdjacent
<< ", kurtosis: " << p->Kurtosis;
				return ItemInfo.str();
			}
		}
	}
	else if(R == 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])
						{
							Plane* p = (*i)->BestFitPlanes[x][y];
							p->Hit = !(p->Hit);
							stringstream ItemInfo;
							ItemInfo << "Slope: " << p->AngleFromZ << ", average adjacent: "
<< p->AverageAdjacent << ", kurtosis: " << p->Kurtosis;
							return ItemInfo.str();
						}
	}
	return "";
}
```

H. 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 1 // Has the bumblebee software been installed?
```

APPENDIX B: USER STUDY APPROVAL AND QUESTIONNAIRE

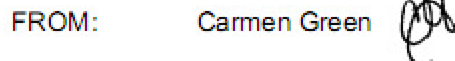
Office of Research Compliance
Carmen T. Green, IRB Administrator
2000 Kraft Drive, Suite 2000 (0497)
Blacksburg, Virginia 24061
540/231-4358 Fax 540/231-0959
e-mail ctgreen@vt.edu
www.irb.vt.edu
FWA00000572(expires 1/20/2010)
IRB # is IRB00000667

DATE: July 11, 2008

MEMORANDUM

TO: Amos L. Abbott
Kevin Kochersberger
Dylan Klomparens

FROM: Carmen Green



SUBJECT: **IRB Exempt Approval:** "Angle Estimation of Objects in 2D Images" , IRB # 08-408

I have reviewed your request to the IRB for exemption for the above referenced project. The research falls within the exempt status. Approval is granted effective as of July 11, 2008.

As an investigator of human subjects, your responsibilities include the following:

1. Report promptly proposed changes in the research protocol. The proposed changes must not be initiated without IRB review and approval, except where necessary to eliminate apparent immediate hazards to the subjects.

2. Report promptly to the IRB any injuries or other unanticipated or adverse events involving risks or harms to human research subjects or others.

Virginia Polytechnic Institute and State University

Informed Consent for Participants

Project title: Angle Estimation of Objects in 2D Images

Investigators: A. Lynn Abbott (faculty advisor), Kevin Kochersberger (faculty advisor), Dylan Klomparens (graduate student)

I. Purpose of Research: The purpose of this research is to identify how well humans can estimate the angle at which an object is tilted, given a two-dimension image. There will be between 15 to 30 subjects for this experiment, with no restrictions on the subject pool.

II. Procedures: You will be asked to estimate the angle at which a wood pallet is tilted in a picture. You will also be asked to compare two or more pictures, and identify the one in which the piece of wood has the least steep angle. You may be asked to repeat this process up to 5 times, requiring no more than 10 minutes to complete the entire experiment.

III. Risks: There are no risks associated with this study.

IV. Benefits: There are no personal benefits to subjects completing this study. No promise or guarantee of benefits has been made to encourage you to participate. The academic benefits of this study include an increased understanding of human interface to semi-automated systems.

V. Extent of Anonymity and Confidentiality: All data collection will be *completely anonymous*. No personally identifiable information about the subjects will be recorded. The data and results of the study may eventually be published.

VI. Compensation: There is no compensation for completing this study.

VII. Freedom to Withdrawal: you may withdrawal from the study at any point. You are free not to answer an experimental question if you choose.

VIII. Subject Responsibilities: I voluntarily agree to participate in this study. I have the following responsibilities: none.

XI. Subject's Permission: I have read the consent form and conditions of this project. I have had all my questions answered. I hereby acknowledge the above and give my voluntary consent. (Consent is implied by returning the completed questionnaire.)

Should I have any questions about this research or its conduct, and the subject's rights, and whom to contact in the event of a research-related injury to the subject, I may contact:

Dylan Klomparens (investigator), by phone 540.630.6229 or by email dklompar@vt.edu

A. Lynn Abbott (faculty advisor), by phone 540.231.4472 or by email abbott@vt.edu

Kevin Kochersberger (faculty advisor), by phone 540.231.5589 or by email kbk@vt.edu

James S. Thorp (department head), by phone 540.231.7494 or by email jsthorp@vt.edu

David M. Moore:

Phone: 540.231.4991

Email: moored@vt.edu

Chair, Virginia Tech Institutional Review

Board for the Protection of Human Subjects

Office of Research Compliance

2000 Kraft Drive, Suite 2000 (0497)

Blacksburg, VA 24060

Please estimate the angle at which the wood pallet in this image is slanted at. (The angle is measured with respect to the floor… in other words, if you think the pallet is perfectly flat your estimation would be 0 degrees). Do not refer to any other pages or images while making your estimate.

Estimate: _____ degrees

Image A

Please estimate the angle at which the wood pallet in this image is slanted at. (The angle is measured with respect to the floor… in other words, if you think the pallet is perfectly flat your estimation would be 0 degrees). Do not refer to any other pages or images while making your estimate.

Estimate: _____ degrees

Image B

Please estimate the angle at which the wood pallet in this image is slanted at. (The angle is measured with respect to the floor… in other words, if you think the pallet is perfectly flat your estimation would be 0 degrees). Do not refer to any other pages or images while making your estimate.

Estimate: _____ degrees

Image C

Please estimate the angle at which the wood pallet in this image is slanted at. (The angle is measured with respect to the floor… in other words, if you think the pallet is perfectly flat your estimation would be 0 degrees). Do not refer to any other pages or images while making your estimate.

Estimate: _____ degrees

Image D

Please estimate the angle at which the wood pallet in this image is slanted at. (The angle is measured with respect to the floor… in other words, if you think the pallet is perfectly flat your estimation would be 0 degrees). Do not refer to any other pages or images while making your estimate.

Estimate: _____ degrees

Image E

Please estimate the angle at which the wood pallet in this image is slanted at. (The angle is measured with respect to the floor… in other words, if you think the pallet is perfectly flat your estimation would be 0 degrees). Do not refer to any other pages or images while making your estimate.

Estimate: _____ degrees

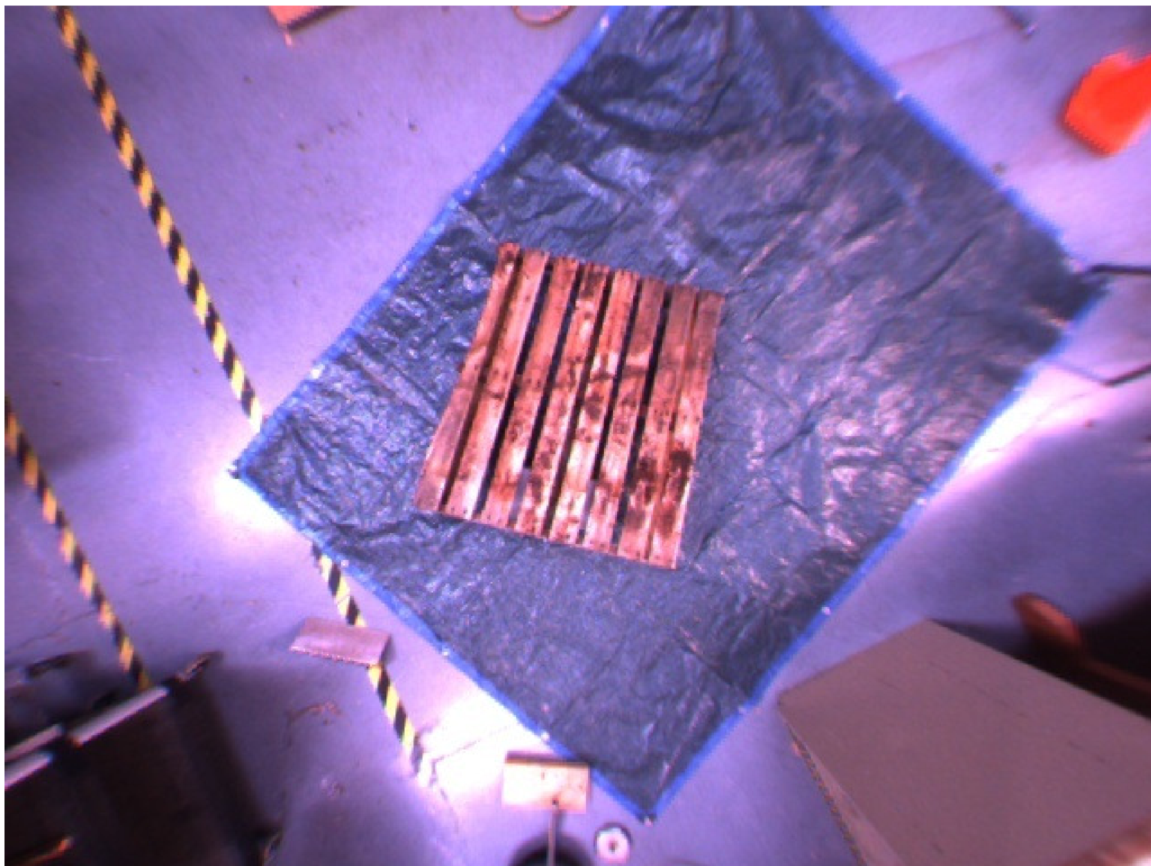


Image F

Please choose the image with the wood pallet slanted at the least steep angle.

Choice: _________

Image 1

Image 2

Image 3

Please choose the image with the wood pallet slanted at the least steep angle.

Choice: _________

Image 1

Image 2

Image 3