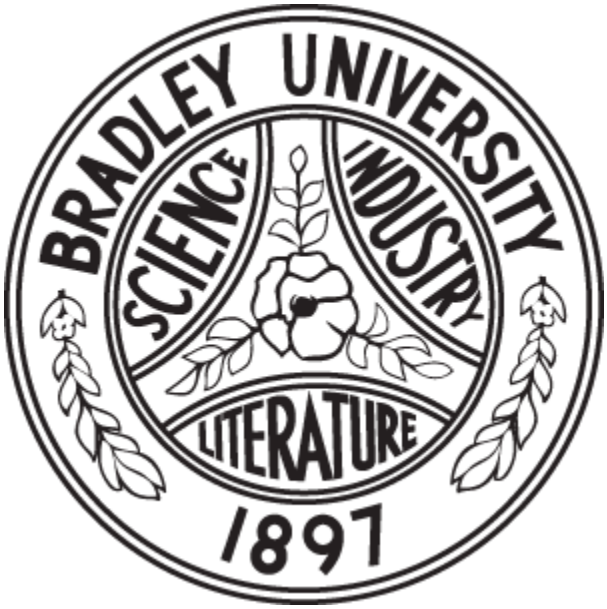

BRADLEY University

3D Environmental Mapping and Imaging for AUVSI RoboBoat

David Bumpus, Dan Kubik, Juan Vazquez

Dr. José Sánchez, Nick Schmidt

Department of Electrical and Computer Engineering

May 3rd, 2016

Abstract

Since 2013, Bradley has twice participated in an autonomous boat competition. Teams are challenged to design a boat that completes challenges without human aid. For successful navigation around buoys and shoreline, accurate distance measurements to objects from the boat must be obtained; this task has proven difficult for Bradley teams. The purpose of this project is to improve the accuracy of these distance measurements through the use of lidar, a laser surveying technique used in the autonomous vehicle industry. This objective is achieved through three stages: data acquisition, data organization, and data processing. Initially, a Velodyne laser scanner and a Logitech webcam are used respectively to record lidar data and photos of the boat's environment. This information is relayed to an Odroid-XU4, a single-board computer selected for its low cost and speed. In the second stage, lidar data is organized based on the azimuthal angle and by regions of interest. In the final stage, the location of the nearest object is determined using feature detection. Additionally, a color-coded image that visually represents distance is produced from the lidar data. This image will help future boat team members assess the navigation system's performance.

Table of Contents

I. Introduction and Overview

A. Problem Background

Every spring, AUVSI (the Association for Unmanned Vehicle Systems International) hosts an International RoboBoat competition in Virginia. Since 2013, Bradley University's Department of Electrical and Computer Engineering has twice participated in this competition. RoboBoat teams are required to design a boat that navigates and performs challenges without human control; all actions must be autonomous. For this reason, it would be beneficial to accurately measure and model the environment around the boat. Obstacles in the water must be detected and noted when making any navigational decisions. This detection has proven difficult for past Bradley teams. Historically, teams have chosen to use a camera to record the environment in front of the boat. While photographic images provide important information about the environment, such as color, these pictures do not provide distance information making it difficult to determine how far an obstacle is relative to the boat. While geometric methods may be used to approximate this distance, these methods have proven to be ineffective. For this reason, the RoboBoat project advisor would like to improve sensing capability of the boat by adding sensors that provide depth information. Increasing the number of sensors on the boat will improve the reliability of the boat's navigational system and increase the chances for success in future RoboBoat competitions. A laser scanner is used to acquire distance information. These scanners utilize a method known as lidar in which lasers are used to measure distance. Lidar is already commonly used in many autonomous vehicle applications.

B. Problem Statement

Dr. José Sánchez advised the development of a three-dimensional (3D) environmental mapping system for use in the RoboBoat competition. Ultimately, the client desired to improve the sensing capabilities of the boat by adding a system that can measure depth. The system would use a laser scanner to complete a full lidar scan of the environment around the boat. This lidar information includes both distance and reflectivity measurements. Measurements are returned to the boat's central processing unit (CPU) and also stored in a hard drive for post-run diagnostics. In addition, the distance measurements are registered with pictures from a camera to create an image with a color overlay to visually demonstrate depth. Finally, the system returns polar coordinates of the location of the object nearest the boat. The system should be lightweight and compact. The lidar system must operate on less than 12 V, draw less than 4 A, and consume less than 50 W. Additionally, the system must fit in a 10 inch (25.3 cm) cube. All of these specifications allowed for the boat to function within competition constraints.

C. Constraints of the Solution

The constraints for the project were agreed upon by the design team, and Nick Schmidt, the RoboBoat advisor. These constraints in Table I are each applied to the system as a whole. Many of these constraints are due to limitations of the boat and rules of the RoboBoat competition. For example, voltage, current, and power constraints are limited by the existing boat's power supply. To prevent capsizing the boat or slowing of the boat, the weight of the system has also been constrained. The rules of the competition and size of the boat also limit how large the system can be, confining the lidar system to a 25.4 x 25.4 x 25.4 cm cube.

The remaining constraints are limited by the preferences of the boat team and how the lidar system will be used. First, the system must measure the environment directly in front of the boat. Assuming the direct front of the boat (bow) is 0°, and the surroundings of the boat encompass a range of 360°, the lidar system must measure objects anywhere in the range of -90° to 90° (270° to 90°) to locate objects that will be important for navigation. Next, two constraints are given that require that the system must measure objects at a minimum distance from the boat. Using testimonials from previous RoboBoat competitions it was determined that objects of interest will be no further than 9.1 m from the boat on the level of the water and will be no higher than 3 m above the level of the lake. Using (1), (2), and (3) found in

Appendix D, the system must measure at least 9.58 m from the boat. Furthermore, to ensure a competitive advantage, a full scan (360°) of the environment must be completed in less than 5 s. Finally, the lidar system must operate for the full duration of a typical competition, 40 min.

TABLE I: LIST OF 3D LIDAR SYSTEM CONSTRAINTS

CONSTRAINTS
3D lidar system must operate on 12 volts or less
3D lidar system must draw less than 4 A
3D lidar system must consume less than 50 W
3D lidar system must weigh less than 3.6 kg
3D lidar system must fit in a 10 inch (25.4 cm) cube
3D lidar system must obtain 180° scan in front of boat
3D lidar system must measure a minimum distance of 9.1 m horizontally
3D lidar system must measure a minimum distance of 3 m vertically
3D lidar system must obtain full scan in no more than 5 s
3D lidar system must operate for 40 min.

II. Statement of Work

The following section contains the statement of work of this project. This section is divided into four subsections for the reader's convenience: *Nonfunctional Requirements, Functional Requirements, Design Overview,* and *Economic Analysis.*

A. Nonfunctional Requirements

TABLE II: LIST OF 3D LIDAR SYSTEM NONFUNCTIONAL REQUIREMENTS

OBJECTIVES
3D lidar system should be weatherproof
3D lidar system should be configurable
3D lidar system should be mechanically stable
3D lidar system should be lightweight
3D lidar system should be executing in a timely manner

Included in Table II is a list of nonfunctional requirements developed for the 3D lidar system. Table XI in Appendix E covers the metrics used to measure how effectively each nonfunctional requirement is met. These metrics are rated on a scale from 1 to 5; with 1 representing the least amount of effectiveness for the nonfunctional requirement and 5 represented the greatest.

"Weatherproof" refers to the ability of the system to survive against the elements such as water, the wind, and the sun. As the boat will be on a lake, it would be beneficial for the system to have safeguards against getting wet. "Configurability" refers to the ease future boat teams can obtain information from the environmental mapping system. This information should be presented in an efficient manner that can be easily accessed by simple commands from the boat's control system. "Mechanically stable" means how the motion of the system, if any, affects the motion of the boat. Less motion is desired for more precise boat maneuvers. "Lightweight" alludes to the preferable lower weight of the system; the boat is less likely

to sink with a lighter load. “Executing in a timely manner” indicates the fact that the system will be used in competition. Faster data acquisition and communication is desired to produce faster navigational decisions, improving the runtime of the boat in competition.

B. Functional Requirements

TABLE III: LIST OF 3D LIDAR SYSTEM FUNCTIONAL REQUIREMENTS

OVERALL FUNCTIONS	SPECIFICATIONS
3D lidar system should pass information to a hard drive	180° of unprocessed scan data, image data, angle and distance to nearest object, and registered image are passed to hard drive.
3D lidar system should pass information to the boat's onboard electronics	Angle and distance to nearest object are passed along with 180° of unprocessed scan data, image data, and registered image.
3D lidar system should accept input angles to define range	Azimuthal range of data can be adjusted without interfering with registration. The dataset returned is reduced accordingly.

Included in Table III are the functional requirements of the overall system and their specifications used to determine if the overall function is met. The first function listed involves the storage of all data in an external hard drive so that it is accessible for troubleshooting after system testing or competitions. Four sets of data must be stored on the hard drive. Similarly, for the second function to be met, the same four sets of data must be returned to the boat’s CPU. Finally, the last function listed is the reduction of the dataset for improved processing speed. This reduction occurs in response to a range defined by two input angles. Regardless of the range defined, the frontal 180° of data must remain intact for use in the registration process.

Tables IV through VI include functional requirements specific to the three subsystems: the laser scanner, camera, and embedded device.

TABLE IV: LIST OF 3D LASER SCANNER SUBSYSTEM FUNCTIONAL REQUIREMENTS

SUBSYSTEM FUNCTIONAL REQUIREMENTS	SPECIFICATIONS
Laser scanner must pass complete scan to embedded device	The entire 360° by 30° dataset of environment must be accessible via the embedded device.
Laser scanner must system must return distance measurements	Distances must range from at least 0 m to 9.58 m, obtained with 95% accuracy.
Laser scanner must return azimuthal angles	Azimuthal angles must range from 0° to 359.99°
Laser scanner must return reflectivity measurements	Reflectivity measurements must range from 0 to 255

Table IV includes the functions and specifications of the laser scanner component of the system. The first function describes the maximum amount of lidar data expected to be transferred from the laser scanner. A full 360° surrounding the scanner as measured by the 16 internal laser channels (30° altitudinal) must potentially be received from the laser scanner and stored in the random access memory (RAM) of the embedded device. The data included in this full scan is included in the following functions describing which specific measurements shall be returned by the laser scanner. The accuracy of the second function is dictated by the accuracy of the scanner and is given as a percentage of the measured

distance. The fourth function alludes to that fact that reflectivity data should fall within the 0 to 255 range as described in [1].

TABLE V: LIST OF CAMERA SUBSYSTEM FUNCTIONAL REQUIREMENTS

SUBSYSTEM FUNCTIONAL REQUIREMENTS	SPECIFICATIONS
Camera must communicate with embedded device	Images must be taken and returned on command
Camera must capture boat environment	Image must be in color and contain at least environment in front of boat from 0° to 30° above water level.

Table V lists the functions and specifications for the camera component of the system. The camera must capture an image on command from the embedded device. These images must be in color to depict a true representation of the environment. Moreover, the field of view of the camera images should be useful for registration purposes and overlap a reasonable amount of the lidar data range.

TABLE VI: LIST OF EMBEDDED DEVICE SUBSYSTEM FUNCTIONAL REQUIREMENTS

SUBSYSTEM FUNCTIONAL REQUIREMENTS	SPECIFICATIONS
Embedded device must register lidar data with image data	Image overlaid with depth information should have RGB colors ranging from 0 to 255
Embedded device must determine nearest object location and distance	Within 9.58m, distance to object must be accurate to within ±0.3 m and angle within ±5°
Embedded device must reduce dataset to desired angle of view	Dataset returned must contain all points within desired range as defined by the two input angles
Embedded device must store information in hard drive	Hard drive must store unprocessed image, unprocessed lidar data, and processed data (distance to nearest object, image overlay with depth information) for one 40 min. boat run.

The functions and specifications for the embedded device are included in Table VI. First, registration must occur in the device, matching lidar data to a camera image. This registration is used to associate areas in the image with measured distances. To represent distance, a color scale must be used that is consistent and should include a spectrum of RGB (red, green, blue) values using the common 0 to 255 notation. This image will be used by future boat teams to verify and visualize the measurements of the mapping system. Next, within the constrained region of 9.58 m (minimum), the distance and azimuthal location of the nearest object must be determined. Finally, the embedded device must transfer all of the data listed above in a hard drive. This hard drive must have enough storage space to store data for a full run of the boat; this may last up to 40 minutes.

C. Design Overview

The following paragraphs will describe the solution and approach the team took to fulfill the functions, objectives, constraints, and requirements discussed above.

First, the laser scanner selected is the Velodyne VLP-16 Puck™. The Puck is a 3D (three-dimensional) laser scanner that is relatively inexpensive compared to other competitors' similar products; even with this lower price, the Puck provides many options that are ideal for this application. First, the Puck is able to operate in the range of 9 V to 32 V. This fits within the constraint of 12 volts or less provided by the boat's power supply. Second, the Puck consumes only 8 W of power. This value is far

below the constraint of 50 W, leaving room for future components that may require additional power. In addition, the dimensions of the Puck fit within a 25.4 cm cube. The scanner measures distances up to 100 m, which is is much further than the required 9.58 m, and greatly improves accuracy at close range. The Puck is able to scan an environment with an azimuthal range of 360° and an altitudinal range of ±15°. The 360° range is sufficient to meet the 180° requirement for the front portion of the boat, and the extra range enables greater flexibility for selectable ranges. To determine the minimum required altitudinal angle, the inverse tangent of the horizontal and vertical measurement requirements, as seen Table I, can be obtained as 18.24°. While the Puck has a maximum altitudinal range of +15°, the scanner can be mounted and tilted so that this 18.24° altitude can be reached. The 3.24° below the horizon will not have a significant impact of the measuring of the environment around the boat as this environment will mostly consist of water. According to [1], the Puck can measure a full 360° field of view in 200 ms, easily meeting the 5 s constraint. To communicate, the Puck uses an Ethernet port; therefore, the embedded device selected must also have an Ethernet port.

For the camera, a 1.3 MP Logitech C500 webcam was selected for the design. The camera uses a Universal Serial Bus (USB) for power, which at most, uses 5 V which is below the 12 V constraint. The current and power draw are also within the constraints for this project. As this camera uses a USB to communicate, the selected embedded device must also have USB capability.

Finally, an Odroid XU-4 was selected as the embedded device for this solution. As mentioned above, it was necessary for the embedded device to have Ethernet and USB capabilities in order to communicate with the laser scanner and camera. The Odroid provides these capabilities as well as two additional USB ports, which improve the reconfigurability factor for future boat teams. To ensure the Odroid will have enough memory space, an embedded MultiMediaCard (eMMC) is also included in the design. In order to meet the storage necessary to hold an entire 40 min. run of the boat, a 1 terabyte (TB) hard drive has been selected for the solution.

Tests can be performed to verify that these components are functioning as desired and to compare the functions of these components to the specifications listed for each system and subsystem. A complete description of these tests can be found in Appendix C.

The following section contains the design overview of this project. This section is divided into seven subsections for the reader's convenience: *System Block Diagram, Subsystem Block Diagram, System State Diagram, Division of Labor, Hardware, Software,* and *Interface*.

1) System Block Diagram

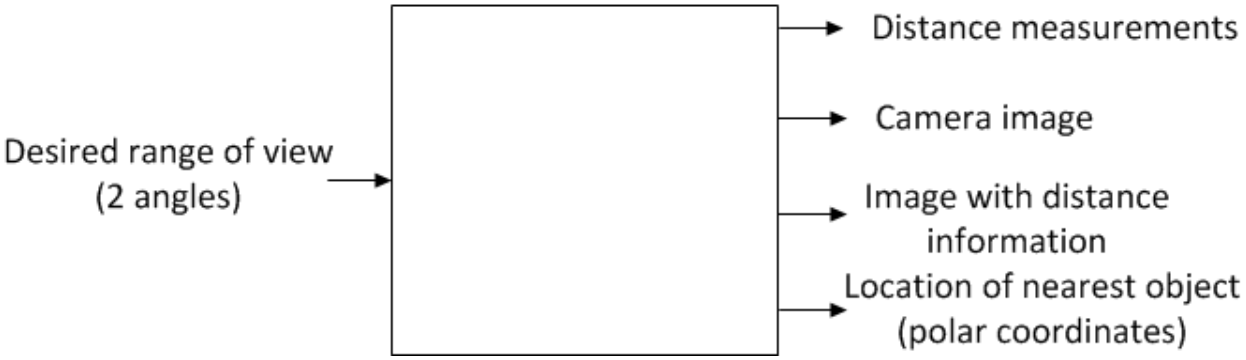

Fig. 1 Proposed solution black box diagram

As can be seen in Fig. 1, the inputs to the system will be two angles which represent a range of the boat's environment that is desired to be processed and mapped. The order of these angles matters as the range will be determined clockwise, from the first angle to the second angle. For example, if the two input angles were 270° and 90°, the desired range would be the front portion of the boat. Conversely, if the two input angles in order were 90° and 270°, the back portion of the boat would be returned, assuming

nothing was blocking the view of the laser scanner. An image depicting the angles can be seen in Fig. 2. Notice that the front (bow) of the boat is considered 0°.

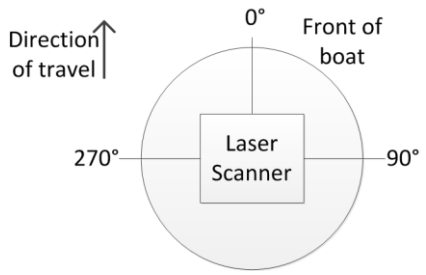

Fig. 2 Input/output angles

The system also has four outputs. First, the system returns distance measurements of the selected range of the 3D environment around the boat. These distances will be sent in the form of spherical coordinates (r, θ, φ), where r is distance, θ is the azimuthal angle, and φ is the altitudinal angle measured from the horizontal relative to the scanner. This can be seen in Fig. 3 below.

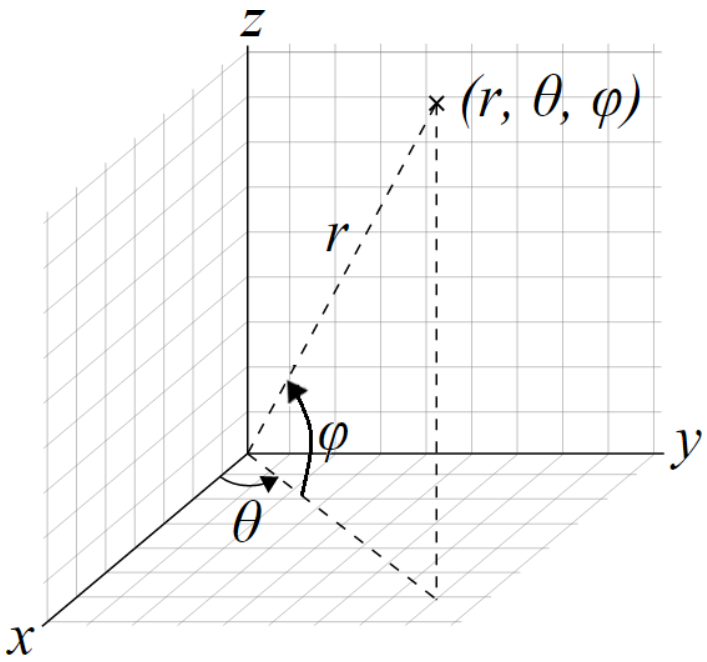

Fig. 3 Spherical coordinate system for returning distance measurements

The system will also return the location of the nearest object. This location will be returned using polar coordinates (r, θ), where r is the distance the object is from the boat, and θ is the azimuthal angle found in Fig. 2. The "nearest object" is determined to be the closest non-water object to the boat. This object will most likely be a buoy, the shoreline, or other hazards in the water. The third output will be a camera image. Finally, the fourth output will be an image registered with distance information. This image is, in essence, a combination of the distance measurements and the camera image. To represent distances in the image, a color overlay will be used. This image will be most helpful for diagnostic testing between competitions.

2) Subsystem Block Diagram

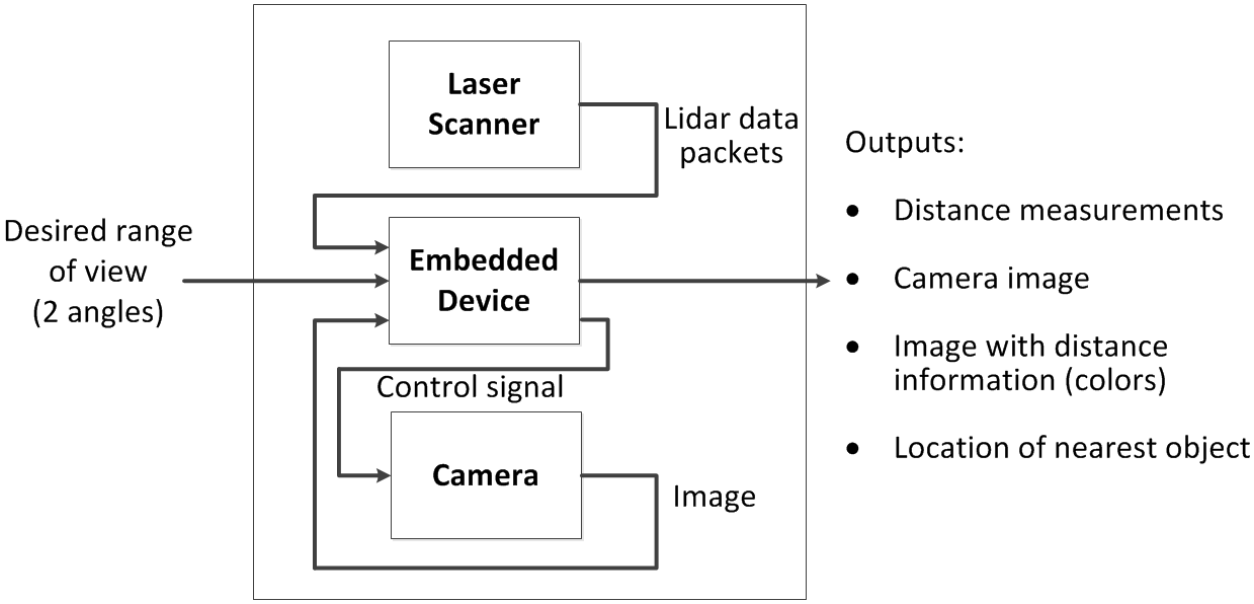

Fig. 4 System glass box diagram

Figure 4 depicts a high level of the internal components of the system and how they interact. There are three main components: a laser scanner, a camera, and an embedded device. The embedded device receives the two angles for the desired field of view. Once the distances have been measured by the scanner and the camera images been taken, the embedded device organizes and processes the data. The laser scanner measures using spherical coordinates, so the embedded device will need to convert this data into rectangular coordinates for use in registration. Registration of the image and the distance measurements will also be completed using the embedded device. Appendix F discusses the experimental results in detail. Using this information, the location of the nearest object can be determined.

3) System State Diagram

A flowchart depicting the high level design of the laser scanner data acquisition software can be seen in Fig. 5 below. First, user angle inputs must be provided, which signify the desired range to be further processed. Next, a "MasterBlock" object is initialized, which is used to store all of the incoming laser scanner data being sent from the VLP-16. Next, the "MasterBlock" is filled with data from each captured scan until a full 360° rotation has been completed. For more information on the MasterBlock object, please see the interface section below.

Next, lidar data and camera images are acquired. A full 360° scan will be taken by the laser scanner. In addition, the camera will record video, from which still images from the individual frames are obtained. All of this data will be sent directly to an external hard drive. Next, "cropping" occurs. For the recorded 360° view of data, the data that falls within the range specified by the input angles can be selected and returned as outputs. All other data in the 360° is unneeded for further processing except the 180° range of data of the front of the boat as seen in Fig. 2. This data will be used for registration. To register 3D distance information with a camera image, it will be necessary to convert the distance measurements from spherical coordinates to Cartesian (rectangular) coordinates. This conversion can be made using (4), (5), and (6) as seen in Appendix D. Once converted, a registration technique will be used to detect edges and objects in the distance and image data. An adaptation of the patented [2] SIFT (scale invariant feature transform) algorithm will be used, composed of feature detection, feature matching, a Transform Mode I estimation, image resampling, and transformation [3]. This technique will be used to

determine the nearest object and produce a registered image with color overlay. Once calculated, the nearest object and registered image will be returned as outputs.

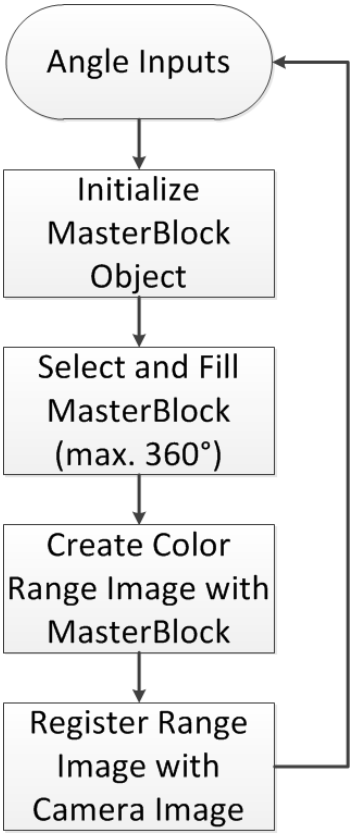

Fig. 5 High level flowchart of lidar system software

4) Division of Labor

The labor had been divided so that each team member was designated as a captain of one major project development category of tasks. As captain, each team member led the development of those tasks, though collaboration was recurrent and necessary throughout the project.

TABLE VII: 3D LIDAR DIVISION OF LABOR

Captain	**Task Category**
David Bumpus	Image registration and processing image data
Dan Kubik	Scan data acquisition and processing
Juan Vazquez	Camera and scanner interface and communication with embedded device

5) Hardware

To review, the system was designed around three main processes: data acquisition, data organization, and data processing. To obtain these three processes, three components were used: a laser scanner, an embedded device, and a camera.

The first process, data acquisition, involved acquiring the information associated with the boat's surrounding environment. There were two components responsible for this process, the laser scanner, the Velodyne VLP-16 Puck, and the camera, the Logitech C500. The laser scanner was responsible for

measuring distance and reflectivity values, which were sent to the embedded device through Ethernet data packets. The camera was responsible for producing a color image of the surrounding environment. The second process, data organization, involved arranging the acquired information into a format that could be used in the data processing stage. This final process, data processing, involved producing the project's designated functional requirements from the newly organized data.

The embedded device chosen, which is involved in all three processes, was the Odroid-XU4, responsible for storing and processing all of the data associated throughout the system.

6) Software

a. Nearest object detection

As previously discussed, one of the functional requirements for this project was to locate the nearest object to the boat and to pass the coordinates of this object to the boat's navigation system. A feature detection algorithm called the Scale Invariant Feature Transform (SIFT) was selected for the detection of the nearest object [2]. The SIFT algorithm detects areas of interest, or keypoints, within a dataset. Unique descriptors for each keypoint were used to match keypoints to similar datasets. This match, or registration, was intended to enable association of distances with objects in an image. This principle is illustrated in Fig. 11 of Appendix F.

As the boat records data from the environment around the boat, rocking in the water will cause rotation of webcam images with respect to previously captured images. Additionally, as the boat moves closer to objects, their apparent size will change. The primary motivation for using this algorithm was that SIFT is robust against changes to the scale and rotation of an image or dataset. This means that objects can be detected in an image and matched with similar images which have experienced a change in scale or rotation.

SIFT keypoint detection for lidar datasets, or point clouds, uses variations in distance to determine potential keypoints within the dataset. Keypoints are often found where adjacent data points indicate significant disparity in distance. This change in distance typically occurs at the edge of an object where adjacent data points are located farther behind the object. After determining the location of keypoints within the point cloud, the distance from the laser scanner to each keypoint was calculated. The nearest keypoint corresponds to the location of the nearest object. The three-dimensional coordinate location of the keypoint was used to calculate the distance, azimuthal angle, and altitudinal angle of that object. The open-source Point Cloud Library (PCL) contains an adaptation of the SIFT algorithm used for keypoint detection in lidar point clouds [4]. Similarly, Open Source Computer Vision (OpenCV) contains an adaptation of the SIFT algorithm used for keypoint detection in images [5].

b. Creation of Color Overlay

In order to achieve the desired color overlay image for use in registration, two challenges needed to be addressed. First, the lidar dataset needed to be reduced from three-dimensions (3D) down to two dimensions (2D) to be viewed as a color image. Second, because of the nature of the spacing of the lasers inside the scanner, large gaps existed where the lasers did not measure the distance in a particular scene. The final color image should visually represent the complete scene, so these gaps needed to be filled.

To fill the gaps of missing lidar data, Barnes interpolation [6] was used. Barnes interpolation was selected for its ability to generate any value in a 2D grid from an unevenly distributed dataset. As the lidar data is unevenly distributed, Barnes interpolation was well suited to generate the distance values used in the image. As Barnes interpolation is designed for interpolation in a 2D plane, it was necessary to reduce the 3D lidar dataset to two dimensions before the interpolation occurred.

This reduction was performed in a two-step process. First, the lidar data was converted from spherical coordinates to rectangular coordinates using (4), (5), and (6) in Appendix D. These equations allow the lidar data to be represented by an X (horizontal), Y (depth), and Z (vertical) coordinate scheme. Second, to visually create a 2D plane to interpolate within, the Y dimension was removed from the lidar dataset. This removal of depth effectively compressed all of the points in 3D space to the vertically positioned XZ plane in 2D. The Barnes interpolation was used within this XZ plane to interpolate between the values of known distances. The newly interpolated values in this plane became the values used in the overlay image.

Once the distances to fill the image were interpolated, a thresholding color scheme was defined. The distances associated with this color scheme were incremented by approximately 1.7 m per color (step) using a linear scale. From nearest to farthest, the nine colors used to represent these subdivisions were: green, dark green, yellow, light blue, blue, dark blue, orange, red, and maroon. This can be seen in Table XII in Appendix F.

7) Interface

Between the data acquisition and data processing, there was a discrepancy of data format. On the acquisition side, lidar data was being sent to the system in packets of encoded information. On the processing side, PCL point clouds were used to detect objects. In order to successfully bridge this formatting inconsistency, it was necessary to organize this lidar data as received from the laser scanner and to format it into a useful manner, one that can be translated into a PCL point cloud. This organization was achieved using a hierarchy of classes in C++. As data packets are received from the scanner, the data is read by an object of the highest level of the hierarchy called a "MasterBlock." If the data received falls within the desired range of information as identified by the two angle inputs received from the boat's control system, it will be stored in the MasterBlock object, taking advantage of different levels of organization provided by the lower levels of the hierarchy. The lowest level is a "dataPoint," which contains the individual distance, reflectivity measurement, and altitudinal angle that correspond to a particular data point. Because of the flexible nature of the MasterBlock class, a MasterBlock object can be many different sizes though can contain no more than one full 360° sweep of data. Once a full sweep of 360° has been presented to the interface of the MasterBlock object it will accept no more data and will activate a flag acknowledging it is "done." At this point, the MasterBlock can be used to easily access the lidar data for further processing such as creating the color overlay image or registration with an image. This process repeats for every instance registration needs to be achieved. For a more detailed explanation of the internal subdivisions of these classes, please see Appendix G.

D. Economic Analysis

The total cost for the proposed project is $8,275.97. Table VIII lists each component of the system, the manufacturer of the component, and the price. As can be seen, a majority of the total system cost comes from the price of the Velodyne VLP-16 Puck 3D scanner. While this may seem expensive, the initial costs of the project were actually projected to be higher; most commercial laser scanners are much larger and designed for larger vehicles such as cars. The Puck is reasonably priced for the features that it can provide and a good size for the RoboBoat application.

TABLE VIII: 3D LIDAR SYSTEM BUDGET

Brand	Part	Price
Velodyne LiDAR	Puck (VLP-16)	$8,094.00
Hardkernel	Odroid XU4	$75.95
Hardkernel	8GB eMMC 5.0 Module	$23.95
Logitech	Webcam C500	$14.43
WD Elements	1TB External Hard Drive	$54.99
Ameridroid	Shipping Estimate	$12.65
Total		$8275.97

III. Design Testing and Validation

Each subsystem was tested and validated independently. The embedded device communication with the laser scanner was verified by organizing the lidar data packet information into a format that could be displayed using PCL. The accuracy of the scanner was verified by observing the accuracy and precision of distance measurements taken from inside a cardboard box. The scanner was placed in the opening of the box and successfully measured the dimensions of the box. The image capture interfacing between the embedded device and webcam was validated by capturing an image and storing it in several file formats compatible with OpenCV image processing algorithms. These images were displayed and verified to be in color. Keypoint detection successfully determined the location of objects within the point cloud and image. As a result, point cloud keypoints were used to determine the nearest object location. Equations (7), (8), and (9) in Appendix D depict the calculation used to determine object distance, azimuth, and elevation. This measurement proved to be accurate over the desired 95% specification.

IV. Results

The MasterBlock class and subordinate hierarchy classes were tested using 300-400 text files each containing a data packet obtained from the laser scanner. A program was written that looped through these files, individually extracted the data, stored the data into an array, and then passed this array to a MasterBlock object. Different angle inputs were given to the MasterBlock object upon initialization to define different areas of interest that should be stored within the object. MasterBlock performed as desired; the data was reduced as directed by the angle inputs. Additional tests were performed to verify the functionality of the 360° "done" flag which were also successful; the MasterBlock would stop accepting additional data after it was determined at least 360° of data had been presented to it. For further pictures and explanations regarding the testing of the MasterBlock class and hierarchy, please see the detailed experimental results in Appendix F.

Keypoint detection for images and point clouds was successfully implemented. These results were determined from trials in two environments. In one environment, a red ball was placed on a chair in the corner of a room. A backpack also hung from a cabinet door. The keypoint detection was able to distinguish objects from walls and cabinets. The location of the nearest object was successfully determined with 95% accuracy. The error was calculated by measuring the distance and angle to the nearest object and comparing the result to the calculated distance. Visual representations of the keypoint detection obtained for the point cloud and image can be seen in Fig. 11 of Appendix F.

To test the creation of a color overlay, the laser scanner was brought to the lobby of the Bradley University basketball arena and set upon a chair. In front of the scanner, there were three large columns,

as well as two large pieces of foam that represented what a buoy may appear to look like on a lake. This setup can be seen in Fig. 13 in Appendix F.

Reducing the lidar dataset from 3D to 2D proved successful. When compressed into the XZ plane, the approximate shapes of objects could be discerned from the original data. Interpolating this dataset created a much more defined image; objects such as the buoys, the columns, and a back wall could be clearly identified using the color thresholds. The interpolated values were compared with the actual distance values to the objects in the scene with favorable results. Of the six major objects measured in the scene, five were assigned the correct color scheme as expected. One of the buoys was the exception, and was off by a significant margin (7 m). It is assumed this buoy error was caused by extraneous floor data, which is discussed in more depth in Appendix F. For further pictures and explanations regarding the interpolation, please also see Appendix F.

V. Summary and Conclusions

The purpose of this project was to develop a system which returns an accurate three dimensional representation of the environment around the autonomous boat used in the RoboBoat competition. A camera, laser scanner, embedded device were selected to meet these needs. There were significant developments within each subsystem. Successful communication between the devices enabled data capture and processing. However, full system integration remained incomplete because of timing issues. The system is able to process the acquired information, but the processing speed is too slow to produce an entire 360° scan. Using test data from the scanner the data organization scheme proved to be effective. Further processing of this data was successful in obtaining the location of the nearest object as well as a color overlay image that can be used in registration.

Potential future work for this project may include multi-threading to improve processing speed and aid complete system integration for practical use. In additions, registration of the image and point cloud keypoints will improve the precision of this overlay. This registration will reduce the effect of a time delay and 3D point cloud distortion when determining object location and distance within a 2D image.

VI. References

[1] "VLP-16 User Manual and Programming Guide 63-9243 Rev A." Velodyne, Aug-2015.

[2] Method and apparatus for identifying scale invariant features in an image and use of same for locating an object in an image by David G. Lowe, US Patent 6,711,293 (March 23, 2004). Provisional application filed March 8, 1999. Asignee: The University of British Columbia.

[3] Lowe, D.G., 1999. "Object recognition from local scale-invariant features." International Conference on Computer Vision, Corfu, Greece, pp. 1150 -1157.

[4] Dixon, Michael. "Overview." *Point Cloud Library (PCL): Module Keypoints*. Open Perception Foundation, 11 Apr. 2010. Web. 27 Apr. 2016.

[5] "OpenCV: Introduction to SIFT (Scale-Invariant Feature Transform)." *OpenCV: Introduction to SIFT (Scale-Invariant Feature Transform)*. Doxygen, 18 Dec. 2015. Web. 28 Apr. 2016.

[6] Stanley L. Barnes, "A Technique for Maximizing Details in Numerical Weather Map Analysis," Journal of Applied Meteorology, vol. 3, pp. 396–409, 1964.

[7] "Make a Cardboard Box City," *Kid Fun*. [Online]. Available: http://kidfun.com.au/images/8671140212.jpg.

Appendix A: Parts List

Table IX: List of required parts

Brand	Part
Velodyne LiDAR	Puck™ (VLP-16)
Hardkernel	Odroid XU4
Hardkernel	8GB eMMC 5.0 Module
Logitech	Webcam C500
WD Elements	1TB External Hard Drive

Table IX contains a list of the parts required for this project. It should be noted that the hard drive was never purchased due to time constraints.

Appendix B: Glossary

Keypoint: Point in a dataset of particular interest, namely, an area of high contrast in an image.

Registration: The process of transforming two or more different datasets into one coordinate system. These datasets may or may not be of similar type.

Azimuth: Horizontal direction expressed as the angular distance between directions of a fixed point and the direction of an object.

Appendix C: Detailed Test Procedures

To verify that the system is performing the required functions, a series of tests should be issued. First, the subsystems must be tested individually.

For the laser scanner, first, it must be identified that a full 360° by 30° field of view is being recorded and passed to the embedded device. This can be checked using the Veloview software. By running the scanner using the Ethernet connection, Veloview will visually represent the data being received. Additionally, the scanner must return distances at least 9.58 m away with 95% accuracy and reflectivity measurements from 0 to 255. These values can be obtained using Veloview, and the 95% can be determined by comparing the measurements seen in Veloview with measurements taken in the classroom. For example, the scanner can be placed in a box. This box is clearly identifiable and can be easily measured.

The embedded device must determine nearest object location and distance. This can be determined by setting up a mock environment where there are a select amount of objects in a room. For example, it has been mentioned that the atrium in the Renaissance Coliseum may be used. In this atrium, objects can be systematically placed and measured as to their whereabouts. Placing the system up high, looking down into the lobby can represent an open area like a lake. If the system can return the angle and distance to their nearest object, it can be determined that the test was a success. To determine if the embedded device is reducing the desired angle of view correctly and performing the registration properly, the outputs from the system sent to the external hard drive can be obtained and examined.

To test the capabilities of the camera, the camera can be connected to the embedded device and commanded to take pictures. By examining these images, it can be determined if the images contain objects further than 9.58 m away with distinguishable edges. In addition, the range of these images can be tested by comparing the size of the frame of the image and how wide the field of view is.

After all of the subsystems have been tested, full system testing can commence. Oscilloscopes, voltage meters, and current meters can be used to determine voltage, current, and power usage. Once calculated, these values can be used to calculate if the system can continuously run for 40 min. The weight requirement can be tested using a scale. The system can be measured using a meter stick to determine if the system will fit within a 25.4 cm cube. The timing of data packets in a scan is provided by the laser scanner and can be used to determine how long it takes to make a full 360° scan.

For a detailed account of the results of the tests performed, please see Appendix F.

Appendix D: Calculations

Initially, two requirements were given describing the distance measured by the laser scanner. These requirements requested that at a minimum, the laser scanner must be able to measure at least a horizontal distance of 9.1 m and at least a vertical distance of 3 m from the laser scanner. Because it is possible that an object can be both 9.1 m away and 3 m above the scanner, it is desired to know this maximum distance from the scanner that is necessary to meet the requirements.

Equation (1) below can be used to determine this distance. As can be seen, the Pythagorean Theorem is used to determine the maximum. It has been determined that the scanner must be able to measure at least 9.58 m away.

$$MaximumRequiredDistance = \sqrt{HorizontalMin^2 + VerticalMin^2} \quad (1)$$

$$MaximumRequiredDistance = \sqrt{(9.1\ m)^2 + (3\ m)^2} \quad (2)$$

$$MaximumRequiredDistance = 9.58\ m \quad (3)$$

Equations (4), (5), and (6) describe the conversion from spherical coordinates to rectangular (Cartesian) coordinates. R represents distance, ω represents the altitudinal angle and α represents the azimuthal angle.

$$X = R * \cos(\omega) * \sin(\alpha) \quad (4)$$

$$Y = R * \cos(\omega) * \cos(\alpha) \quad (5)$$

$$Z = R * \sin(\omega) \quad (6)$$

Equation (7), (8), and (9) are used to determine the nearest object location by calculating the azimuthal angle to the nearest object, the angle of elevation to the nearest object, and the distance to the nearest object. Notice that the distance equation (7) can be derived from the traditional 3D distance equation knowing that the center of the laser scanner is located at the origin (0, 0, 0).

$$Distance = \sqrt{x^2 + y^2 + z^2} \quad (7)$$

$$Azimuth = \sin^{-1}\left(\frac{z}{distance}\right) * \frac{180}{\pi} \quad (8)$$

$$Elevation = \tan^{-1}\left(\frac{y}{x}\right) * \frac{180}{\pi} \quad (9)$$

Appendix E: Metrics for Nonfunctional Requirements

TABLE XI: METRICS FOR 3D LIDAR SYSTEM NONFUNCTIONAL REQUIREMENTS

Objective: 3D lidar system should be weatherproof	
5	Can operate in rain, fog, wind, waves, and varying light intensities
4	Can operate in fog, wind, waves, and varying light intensities
3	Can operate in wind, waves, and varying light intensities
2	Can operate in wind
1	Only works indoors
Objective: 3D lidar system should be configurable	
5	System is highly configurable
4	System is configurable
3	System is somewhat configurable
2	System is not very configurable
1	System is not configurable
Objective: 3D lidar system should be mechanically stable	
5	Motion in system is contained
4	Motion in system is mostly contained
3	Motion in system is somewhat contained
2	Motion in system will impair other boat functions
1	Motion in system will disable boat
Objective: 3D lidar system should be lightweight	
5	Weight less than 2 kg
4	Weight less than 2.86 kg
3	Weight less than 3.72 kg
2	Weight less than 4.58 kg
1	Weight less than 5.44 kg
Objective: 3D lidar system should execute in a timely manner	
5	Scan time and processing less than 5 s
4	Scan time and processing less than 5.5 s
3	Scan time and processing less than 6 s
2	Scan time and processing less than 6.5 s
1	Scan time and processing less than 7 s

Appendix F: Detailed Experimental Results

I. Data Organization

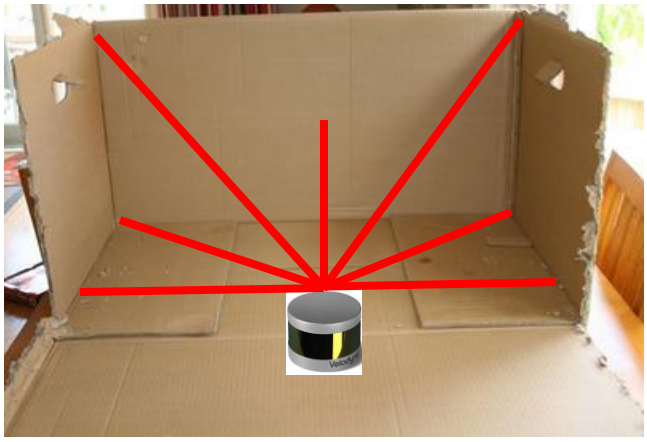

Fig. 6 Scanning a cardboard box [7]

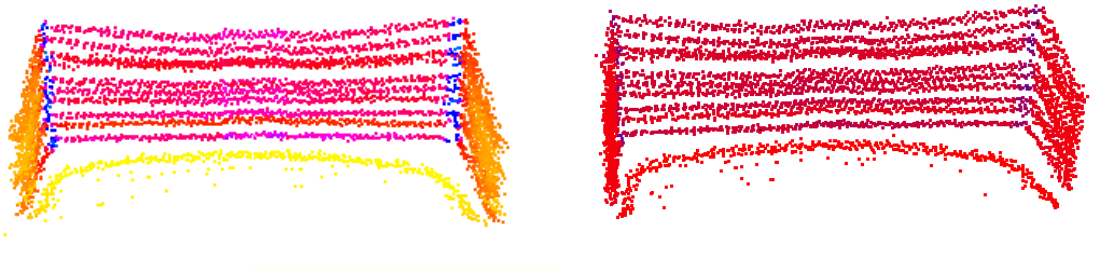

Fig. 7 Comparing lidar datasets: left is viewed using VeloView software, right is viewed through a PCL visualization

In Fig. 6, the setup of one of the initial lidar data acquiring tests is shown. In Fig. 7, the resulting lidar datasets can be seen. The left portion of Fig. 7 shows what the data is expected to look like; this is a screenshot of what is seen using the VeloView software included with the scanner. The right portion of Fig. 7 shows decoded information from text files as stored in a MasterBlock object. To visualize this dataset as seen in the image, the MasterBlock object was converted to a PCL (Point Cloud Library) point cloud, and visualized using the PCL viewer. Notice that the datasets are identical in size and shape, demonstrating the success of the MasterBlock scheme. The colors in these particular visualizations represent reflectivity. The colors from image to image appear different due to different color schemes used.

Figure 8 through Fig. 10 demonstrate the functionality of the selection range of the MasterBlock object. In Fig. 8, the complete 360° field of view of the lobby of the Bradley basketball arena can be seen. To save all of this information in a MasterBlock object, the inputs (0°, 359°) were used. Desiring only the frontal 180° area in front of the scanner was obtained using the angle inputs of (270°, 90°) as seen in Fig. 9. Notice how the lower 180° has been removed and is no longer stored in the MasterBlock. This can most easily be seen by the circle of data points picked up on the floor around where the laser scanner was located. Similarly, in Fig. 10 the range has been further defined to

the frontal 90°. Using this method, any range can be successfully obtained within the 360° of the full lobby.

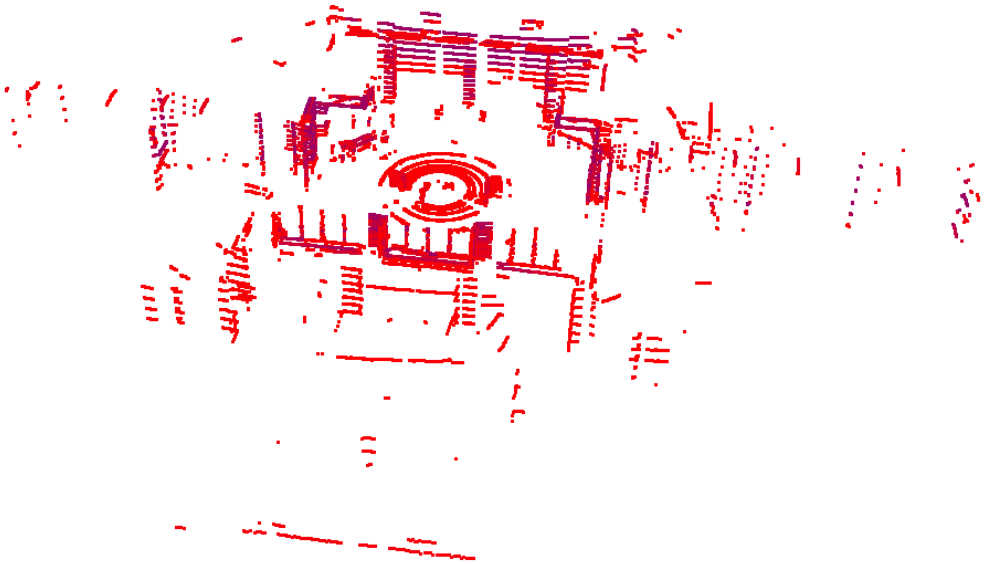

Fig. 8 Full 360° view of the Renaissance Coliseum lobby: angle input was (0°, 359°)

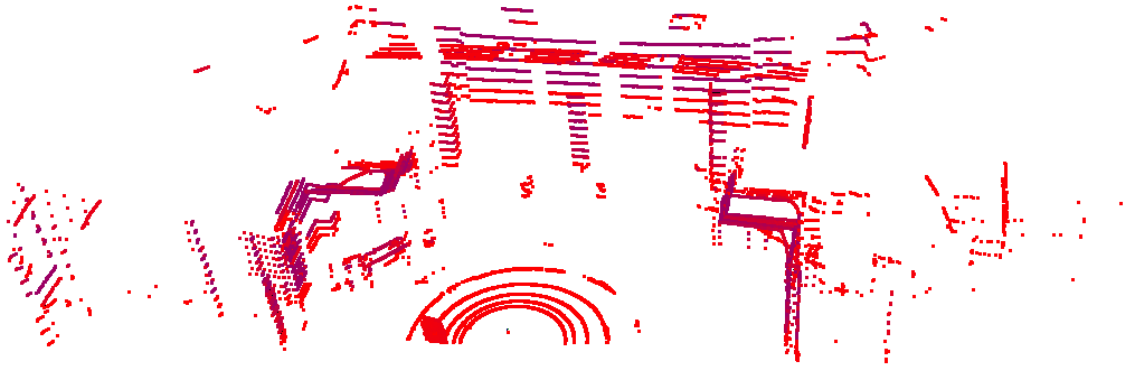

Fig. 9 Frontal 180° of the Renaissance Coliseum lobby: angle input was (270°, 90°)

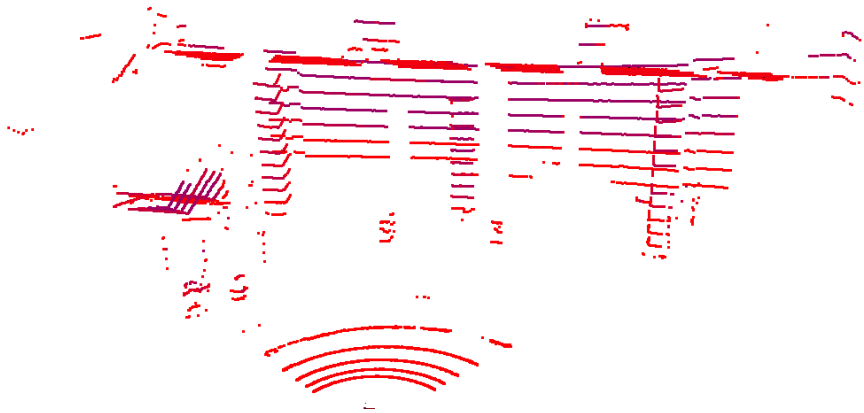

Fig. 10 Frontal 90° of the Renaissance Coliseum lobby: angle input was (315°, 45°)

II. Nearest Object Detection

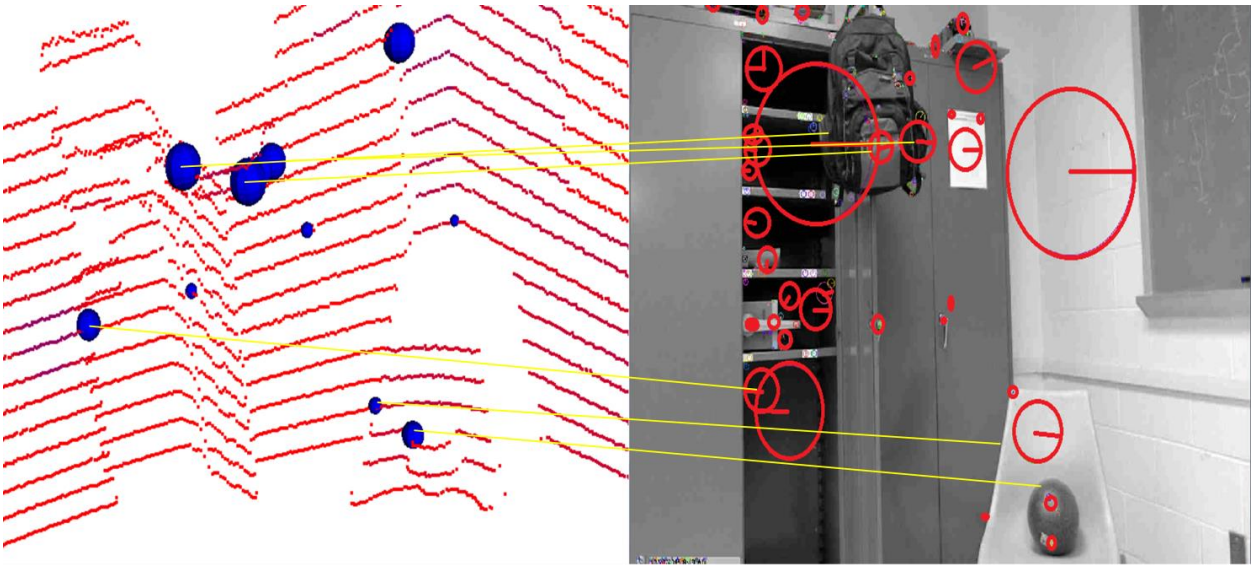

Fig. 11 Comparison of PCL Keypoints (left) and OpenCV SIFT keypoints (right)

Figure 5 depicts keypoint detection for an environment containing a red ball on a chair, with a backpack hanging from an open cabinet door. The keypoints occur at locations of high gradients. For the PCL point cloud keypoints, each keypoint is found by computing gradients of distance measurements. The keypoints are shown as blue spheres. These keypoints are found at the edges of objects and can be used to determine the nearest object location. For the OpenCV image keypoints, a red circle visually describes each keypoint. The SIFT algorithm uses an array of gradient vectors measuring changes in color throughout the image to describe each keypoint. This array, referred to as a "descriptor," can be matched with similar descriptors in other images for the purpose of registration. The descriptor is visually represented by a vector extending from the center of each circle.

```
Release>img_lidar_matching.exe RED_BALL_chair.jpg ball_and_chair.pcd

The nearest object is: 1.71135 meters away
 with an azimuth of 3.00008

There are 64 lidar keypoints detected.

There are 400 image keypoints detected.
```

Fig. 12. Command output of SIFT keypoint detection algorithm

Figure 12 displays the calculated azimuthal angle and distance to the nearest object. This output is generated from the dataset seen in Fig. 11. However, the point cloud spans 180°, which contains significantly more data than what can be seen in Fig. 11. The chalkboard on the right wall extends out of the field of view and is the closest object to the sensor. Reducing the field of view for the point cloud to match the perspective of the image produced the location of the nearest object in an image.

III. Creating a Depth Overlay

Fig. 13 Setup of the laser scanner in the lobby of the Bradley Renaissance Coliseum

Figure 13 depicts the setup of the laser scanner for data acquisition in the Bradley University basketball arena, the Renaissance Coliseum. The laser scanner is located on the red seat of the left chair. Notice the three main columns in the lobby, as well as the two pieces of black foam between the columns that represent what a buoy may look like. Notice also the shape of the ceiling, and the staircases on the left and the right of the image.

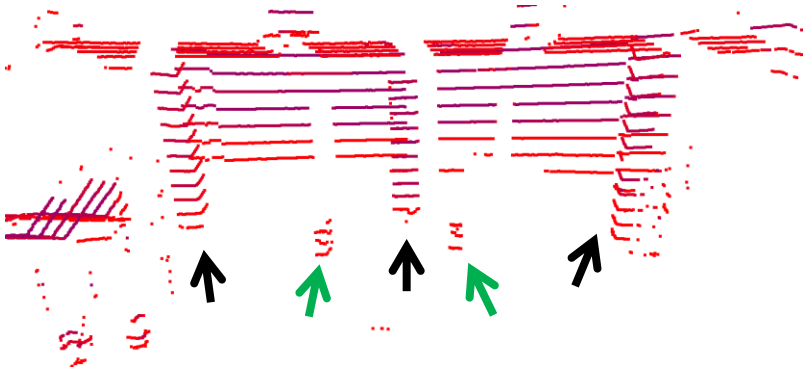

Fig. 14 Reduced lidar dataset of the Renaissance Coliseum setup

Figure 14 shows the lidar dataset used to create the color overlay image using the Barnes interpolation. Notice that this dataset is not a 360° representation of the lobby; this 90° dataset was reduced from the original 360° using the selection range functionality of the MasterBlock class with input angles of (315°,45°). The three columns can clearly be seen (black arrows), as well as the two buoys (green arrows). Floor data points also exist in the dataset, but have been cropped from this image, and are located closer to the scanner near the center of the image. A few of the floor data points can be seen below and to the left of the center black arrow.

Fig. 15 Barnes interpolation of lidar dataset with mostly removed floor data points

Figure 15 displays the Barnes interpolation for the lidar dataset. This interpolation was performed after removing the floor data points; it was determined that including the floor data will skew the visual results of the objects in the image.

Notice the clarity of some of the objects in this interpolation; the three pillars are very notable in orange and the two buoys, in green and blue, are visible in the center foreground. Even the irregular shapes of the ceiling can be seen in red and maroon between the pillars. While the stairs on the left and right of the image are not as clearly defined, this was expected as the shapes were larger and more complex. For the purpose of identifying objects on a lake, this level of visual identification of objects should be sufficient.

Table XII shows the ranges associated with each of the color thresholds used in Fig. 15. Each interpolated distance is assigned a color, with the number in each box representing the lower bounds of the range of that color. The farthest distance measured in the image was about 14.8 m from the scanner.

Table XII: Color range thresholds for Barnes interpolation.

0.0 m	1.8 m	3.5 m	5.3 m	7.0 m	8.8 m	10.5 m	12.2 m	14.0 m

Using the color scheme in Table XII, the expected color of each object was determined. These colors were compared to the colors seen in the interpolated values in Fig. 15 to determine the accuracy of the interpolation. The back walls, left and right columns, and right buoy all were interpolated correctly and assigned the color that was expected. The center column is slightly closer to the scanner, and therefore should have been assigned dark blue; the error from the actual to resultant values (dark blue to orange) was about 0.3 m. This error is understandable as the distances to the three pillars are on the verge of the dark blue/orange color border. The other notable error is the left buoy. Notice how the buoy is assigned the green color (nearest to the scanner) when it should have been assigned a blue color like the right buoy. This error is significant, and as mentioned in the body of the report, is believed to be caused by floor data points interfering with the interpolated values.

Floor data is suspected for a few reasons. The floor had to be initially removed from the dataset to improve upon the visual quality of the interpolation. With the floor in the interpolation, values became skewed to appear closer to the scanner than they were in reality. To remove this floor data, all data points with a Z component less than half of the height of the chair the scanner was located on was removed. Assuming the scanner is parallel with the floor, this should have removed all of the floor data, as well as the bottoms of parts of the columns and buoys. In reality, the scanner was not parallel with the floor and was slightly angled on the seat of the chair. This tilt limits the functionality of the Z component filter. Some floor data will not have been removed and will still affect the data in the rest of the image. This tilt will most undoubtedly be found in practice on the boat. A more complex filter of floor (or water) data will need to be developed to address this concern.

Appendix G: Structures of Storage Class Objects

Table XIII includes each of the classes in the MasterBlock hierarchy, from MasterBlock down to dataPoint. The right column describes which elements are included in each class. For the most part, each of these classes follow the general structure of a data packet as received by the laser scanner [REFERCENCE to MANUAL]. All classes are of a rigid size except the MasterBlock, which utilizes dynamic memory. Notice also that MasterBlock itself contains many HalfDataBlocks rather than DataPackets as may be expected. While DataPacket objects are sent to a MasterBlock to be read, from here they are are separated into HalfDataBlocks to provide greater flexibility of which azimuthal angles are actually stored into a MasterBlock. Data from the DataPackets are decoded as described in [MANUAL].

Table XIII: Storage classes and their features

Class Name	Features
MasterBlock	Ensures 180° frontal view and selected range are stored within. Contains multiple HalfDataBlocks
DataPacket	Contains twelve DataBlocks and a time stamp
DataBlock	Contains two HalfDataBlocks
HalfDataBlock	Contains sixteen DataPoints and an azimuth
DataPoint	Contains one distance, reflectivity, and altitudinal angle

Appendix H: Theory of Operation

Camera:

1. Install "OpenCV" libraries and application onto Odroid-XU4 (via Ubuntu Server)
2. Create camera directory
3. Download the image capture file "capture.cpp" and transfer into the camera directory
4. Compile the image capture file using the command "g++ capture.cpp –o capture `pkg-config -–cflags -–libs opencv`". It's important to note that the symbol "``" are grave accents and not apostrophes. If apostrophes are used the program will not allow for compiling.
5. Run the executable file using the command "./capture," which will then continuously capture and overwrite images using the Logitech C500 webcam.

Laser Scanner:

1. Create laser scanner directory (*No prior libraries or application installations are required*)
2. Download the Ethernet packet capture file "detection.c" and transfer into the laser scanner directory.
3. Compile the Ethernet packet capture file using the command "g++ TestClasses1.cpp dataBlock.cpp dataPacket.cpp halfDataBlock.cpp masterBlock.cpp recDataPoint.cpp sphDataPoint.cpp –o main –std=c++11".
4. Run the executable file using the command "./detection," which will then continuously capture incoming Ethernet packets from the laser scanner while filling and overwriting the "MasterBlock" object with the corresponding data.

The following is an explanation of operation for the SIFT keypoint detection algorithm described in Appendix D. Two open source software libraries are required and can be installed and compiled with Microsoft Visual Studio.

SIFT keypoint detection algorithm operation:

1. Install OpenCV and PCL (including all PCL dependencies)*
2. Set up visual studio properties sheet for compiling PCL and OpenCV *
3. Build project in release mode
4. Copy datasets (<img.jpg> and <pointcloud.pcd>) to the path of the release application
5. Open command prompt and navigate to directory of application (img_lidar_matching.exe)
6. Enter application name, image, and point cloud and press enter

*Steps 1 and 2 are extensive and involve many elements outside of the scope of this report.

Appendix I: Code

The following appendix contains a majority of the code created for this project. This appendix is broken into three main divisions: data acquisition, data organization, and data processing.

Data Acquisition:

Ethernet Detection Program:

This program was written for allowing for interface between the Logitech C500 camera and Odroid-XU4. Specifically, it allows for a continuous image capture from the webcam and conversion into .PPM format using OpenCV libraries and commands.

```
/*
* Name: Juan Vazquez
* Senior Project: 3D Environmental Mapping & Imaging for AUVSI RoboBoat
* Department Name: Electrical and Computer Engineering
* Institution Name: Bradley University
*
*
* ***********************************
* *       REFERENCES/CREDITS         *
* ***********************************
* The following code was altered from the video capture code provided from
"Reading and Writing Images and Video" provided by OpenCV
* Link1:
docs.opencv.org/modules/highgui/doc/reading_and_writing_images_and_video.html
*
* ***********************************
* *            PURPOSE               *
* ***********************************
* This program was written for allowing for interface between the Logitech
C500 camera and Odroid-XU4. Specifically it allows for
* a continuous image capture from the webcam and conversion into .PPM format
using OpenCV libraries and commands.
*
*/

#include "opencv2/opencv.hpp" //OpenCV libraries
#include <unistd.h> //Added for sleep command

using namespace cv;

int main(int, char**)
{
    unsigned int i, sleep(unsigned int seconds); //integer for loop count and
delay declarations
    char filename[4]; //array used for multiple images

    VideoCapture cap(0); //Opening Logitech C500 camera
    if(!cap.isOpened()) //Checking if camera is open
        return -1;
```

```
    while(1) //infinite loop for continous capture
    {
        for(i=1;i<4;i++) //continously capturing 3 images
        {
        sleep(5); //added for delay
            Mat image; //create a blank image
            cap >> image; //capture image using OpenCV

            printf("Image %d Captured\n", i); //Show that an image is being
captured
            sprintf(filename, "CapturedImage%d.ppm",i); //Store in .PPM
format
            imwrite(filename,image); //Write to .PPM format file

        }
    }

    return 0;
}
```

Ethernet Detection Program:

This program was written in order to allow for interface between the VLP-16 laser scanner and the Odroid-XU4. Specifically it scans for incoming raw and formatted Ethernet traffic (UDP) and records individual packets into corresponding text files. IT is mainly dependent on the “libpcap” library.

```
/*
* Name: Juan Vazquez
* Senior Project: 3D Environmental Mapping & Imaging for AUVSI RoboBoat
* Department Name: Electrical and Computer Engineering
* Institution Name: Bradley University
*
*
* ***********************************
* *       REFERENCES/CREDITS         *
* ***********************************
* The following code was altered from the packet sniffer code provided from
"Packet Sniffer Code in C using Linux Sockets (BSD)
* - Part 2" provided by Binary Tides
* Link1: http://www.binarytides.com/packet-sniffer-code-in-c-using-linux-
sockets-bsd-part-2/
*
* ***********************************
* *            PURPOSE                *
* ***********************************
* This program was written in order to allow for interface between the VLP-16
Laser Scanner and the Odroid-XU4. Specifically, it
```

```
* scans for incoming raw and formatted Ethernet traffic (TCP, UDP, ICMP,
IGMP) and records individual packets into corresponding
* text files. It is mainly dependent on the "libpcap" library.
*
*/

/*
* ************************************
* *            LIBRARIES              *
* ************************************
*/

//||Packet Classification Libraries||
#include <iostream>
#include "dataPacket.h"
#include "recDataPoint.h"
#include "masterBlock.h"
#include <cstdint>
#include <stdio.h>
#define _CRT_SECURE_NO_WARNINGS
#include <fstream> //For text file reading
#include <string>
using namespace std;
#include <sstream>//For string to hex conversion (and the backwards '>>')

//||Ethernet Detection Libraries||
//||Standard Libraries||
#include<stdlib.h>      //For use of "malloc()" command
#include<string.h>      //For use of "strlen()" command
#include<unistd.h>
#include<sys/ioctl.h> //For use of "ioctl()" command

//||Header Declarations||
#include<netinet/ip_icmp.h> //ICMP
#include<netinet/udp.h>     //UDP
#include<netinet/tcp.h>     //TCP
#include<netinet/ip.h>      //IP

//||Ethernet/Internet Protocol||
#include<netinet/if_ether.h>    //For ETH_P_ALL
#include<net/ethernet.h>        //For ether_header
#include<netinet/in.h>
#include<netdb.h>               //Internet Database
#include<sys/socket.h>
#include<arpa/inet.h>           //Internet Operations

//||Miscellaneous||
#include<errno.h>       //Error Codes
#include<sys/time.h>    //Time Types
#include<sys/types.h>   //Data Types

/*
* ************************************************
* *            FUNCTION DECLARATIONS              *
* ************************************************
*/
```

```
//||Packet Classification Declarations||/
void PacketClassification(MasterBlock &pointerToPermanentMB, int fileNumber,
string line); //References

//||Ethernet Detection Declarations||/
string EthernetDetection();
/*Changed from Dan Revision */
//char* EthernetDetection();

//Text File Creation
FILE *logfile;
char filename[] = "DataPacket1.txt"; //ADDED //PROBLEM FOR REMOVING |00|00|
or old text files
//Ethernet Detection
struct sockaddr_in source,dest;
int udp=0,total=0,i,j;
//const char* arr[3619]; //Global Variable? ||CHECK THIS CODE||

int main()
{


    while (1)
    {
        MasterBlock marco = MasterBlock(0,359); //Create new Master Block
        MasterBlock  &referenceToMarco = marco;//this is a NEW line. I am
creating a reference to marco called "referenceToMarco"
        int temporaryFileCounter = 1; //see below

        //Loop until Master Block is full
        while (! marco.isMasterBlockReadyYet())
        {
            //logfile=fopen("dataPacket1.txt","w");
            /*ETHERNET DETECTION FUNCTION*/
            //unsigned char *buffer = (unsigned char *) malloc(65536); //Its
Big!
            string line = EthernetDetection();
            printf("One Packet Printed\n");
            //cout << line;
            //COMEBACK sleep(.5);
            /*PACKET CLASSIFICATION FUNCTION*/ //Currently Disabling
//          PacketClassification(referenceToMarco,temporaryFileCounter,line);
//ADDED MASTERBLOCK
//          temporaryFileCounter++;

            /*REMOVE FUNCTION*/
            //remove(filename);

            /*PRINT ARRAY*/
            /*print("\nARRAY\n");
            int q;
            for(q = 0;q<3619;q++)
            {
                printf("%c", arr[q]);
            }
            print("\n");
            */
```

```
        }
        printf("\nMaster Block is Full\n"); //TROUBLESHOOTING: Shows each
time a Master Block is completed
        //break; //To break out of infinite loop and end program FOR TESTING
    }
    return 0;
}


/*Ethernet Detection VOID Start*/
string EthernetDetection()
/*Changed From Dan's Revision*/
//char * EthernetDetection()
{
Restart:
/*Ethernet Program START*/
    unsigned char *buffer = (unsigned char *) malloc(65536); //It's Big
    int saddr_size , data_size;
    struct sockaddr saddr;
    char arr[3619];
    //COMEBACK char* arr[3619]; //Previously was without error
    //char* arr[3619];
    char str[3619];
//  logfile=fopen("dataPacket1.txt","w"); //Creating Text File 1

//  if(logfile==NULL) //If Text File Can't Open
//  {
//      printf("Unable to create log.txt file.");
//  }

    printf("\nEthernet Detection Started\n");
    int sock_raw = socket( AF_PACKET , SOCK_RAW , htons(ETH_P_ALL)) ;

    if(sock_raw < 0) //Error Message
    {
        //Print the error with proper message
        perror("Socket Error");
        //return void(1);
    }

    while(1) //Writing Ethernet Data INFINITE LOOP
    {
        saddr_size = sizeof saddr;
        //Receive a packet
        data_size = recvfrom(sock_raw , buffer , 65536 , 0 , &saddr ,
(socklen_t*)&saddr_size); //Reading Size of Incoming Data
        if(data_size <0 ) //Error Message
        {
            printf("Recvfrom error , failed to get packets\n");
            //return void(1);
        }

        //Now process the packet
        //||PROCESS PACKET START
            //Get the IP Header part of this packet , excluding the Ethernet
header
```

```
            struct iphdr *iph = (struct iphdr*)(buffer + sizeof(struct
ethhdr));
            ++total;
            switch (iph->protocol) //Check the Protocol and do accordingly...
            {
            /*VLP-16 Only Sends UDP Packets */
                case 17: //UDP Protocol
                    ++udp;
                //||UDP PACKET PRINT START||
                    unsigned short iphdrlen;
                    struct iphdr *iph = (struct iphdr *)(buffer +
sizeof(struct ethhdr));
                    iphdrlen = iph->ihl*4;
                    struct udphdr *udph = (struct udphdr*)(buffer + iphdrlen
+ sizeof(struct ethhdr));
                    int header_size =  sizeof(struct ethhdr) + iphdrlen +
sizeof udph;

                    //Move the pointer ahead and reduce the size of string
                    //PrintData(buffer + header_size , data_size -
header_size);
                    //||PRINT DATA START||
                        unsigned char* data;
                        int Size;
                        data = buffer + header_size;
                        Size = data_size - header_size;

                        int i , j;
                        //Restart:
                        /*PRINTING DATA LOOP*/
                        //While i < (Total Size of Data Packet) --> Print
Data
                        int a = 0;//ADDED THIS
                        for(i=0 ; i < Size ; i+=2) //Size = Data_Size from
Ethernet Detection Void Function
                        {
                            /*TEXT FILE PRINTING*/
                            if ( i > 3 ) //Start at |FF|EE| //What is I?
                            {
                                //str[3 - i ] = "|%02X",(unsigned
int)data[i];
                                /*Array Implementation Start*/
                                //COMEBACK arr[ 3 - i ] = "|%02X",(unsigned
int)data[i]; //Store
                                arr[ a ] = '|';
                                printf("%X",arr[a]);
                                a++;
                                arr[ a ] = data[i];
                                printf("%X",arr[a]);
                                a++;
                                arr[ a ] = data[i+1];
                                printf("%X",arr[a]);
                                a++;

                                //string threeletters = "|%02X", (unsigned
int)data[i]; //need to use "" instead of '' o.w. they are character literals
```

```
                                //string threeletters = "|%02X",(unsigned
int)data[i];
                                //arr[3*i] = threeletters[0];
                                //arr[(3*i)+1] = threeletters[1];
                                //arr[(3*i)+2] = threeletters[2];

                                /*Array Implementation End*/



//                              fprintf(logfile , "|%02X",(unsigned
int)data[i]); fflush(logfile);
                                /*REMOVE ZERO PACKETS*/
                                if ((unsigned int)data[4] == 0) //Checking
for |00|00|
                                {
                                    printf("\nZero Packet Detected\n");
                                    i = 0;
//                                  fclose(logfile);
                                    //logfile=fopen("dataPacket1.txt","w");
//Creating Text File 1
                                    sleep(5);
                                    goto Restart;
                                }
                            }
                            if( i==Size-1)  //print the last spaces
                            {
                                //str[ i ] = "|";
                                /*Array Implementation Start*/
                                arr[ a ] = '|';//arr[ i ] = '|';
                                //COMEBACK arr[ i ] = "|";
                                /*Array Implementation End*/


//                              fprintf(logfile , "|"); fflush(logfile);
//Print the last line
                            }
                        //string line(arr[3619]);
                        //std::string line = string(arr[3619]);
                        }
                    //||PRINT DATA END||
                //||UDP PACKET PRINT END||
                    break;
            }
        //||PROCESS PACKET END
        break; //Once packet processed break out of loop
    }
//  fclose(logfile); //Close text file
    //std::string line = string(arr[3619]); //This will print empty array
//COMEBACK  std::string line = string(arr);

    //cout << line << "\n";
//COMEBACK  printf("String: %s\n",line.c_str());
//COMEBACK  return line;

/*Changed for String from Original Form  */  //POSS COME BACK
```

```
//char*  thethingwereturn = arr;
//return  thethingwereturn;

std::string line = string(arr);
return line;

}


/*Dan Program VOID Start*/
void PacketClassification(MasterBlock &pointerToPermanentMB, int fileNumber,
string line) //ADDED MASTERBLOCK //PROBLEM: Created for POINTER
{
/*Dan's Program START*/
/*
    int number = 1;
    string line; //reading a text file
    string Result;          // string which will contain the result
    ostringstream convert;   // stream used for the conversion
    convert << number;      // insert the textual representation of 'Number'
in the characters in the stream
    //converting loop#^^ to a string
    Result = convert.str(); // set 'Result' to the contents of the stream
e.g. now = "123"
    string hereIsMyFileName = "dataPacket" + Result + ".txt";

    ifstream myfile (hereIsMyFileName); //e.g. "dataPacket.txt"
    if (myfile.is_open())
    {
        while (! myfile.eof() )
        {
            getline (myfile,line);
            cout << line << endl;
        }
        myfile.close();
    }

    else cout << "Unable to open file";
*/

//  cout << line << endl;

    uint8_t dataIn[1206];
    int spot = 0;

    /*Classifying Different Data*/
    for (int i=0;i<3618;i+=3) //cause that is how long the variable "line" is
in characters (essentially, length(line))
    //math checks out: 3618/3 = 1206
    {
        //create a string from two characters:
        std::string myByteString = std::string() + line[i+1] + line[i+2];
//e.g. 'f'+'f' = "ff"

        //turn a string into a number (accepts many types)
```

```
        uint16_t x;   //for some reason, the function below doesn't like
uint8_t though, but works for uint16_t >> so, I will convert later (see
below).
        std::stringstream ss;
        ss << std::hex << myByteString; //<<here's that string I made
        ss >> x;

        uint8_t y = (uint8_t)x; //convert from uint16_t to uint8_t >> will
work cause the biggest grabbed values will be a byte

        dataIn[spot] = y; //load it into our array
        spot = spot+1;
    }

    DataPacket myPacket = DataPacket(dataIn);
    pointerToPermanentMB.readNewPacket(myPacket);

/*Dan's Program END*/
}
```

Data organization:

The following code is the hierarchy of classes in C++ used to organize the lidar data, as well as a main file (main.cpp) demonstrating how to send data packets in text file format to a MasterBlock object. Notice that in this example, every text file is named in the format "dataPacket[#].txt." Point Cloud Library (PCL) is necessary for the visualization of the data in a MasterBlock. The file mbtopcl.cpp is used to convert MasterBlock objects to PCL format to be used in this visualization.

To create text files in the appropriate format, please see the MATLAB (.m) file at the bottom of the data organization section. Also notice that header (.h) files were not included in this document to save space, but can be reconstructed using the functions as seen in the provided .cpp files.

```
//Dan Kubik
//Bradley University
//Department of Electrical and Computer Engineering
//April 11, 2016

//main.cpp
//Converting incoming text files of lidar data to PCL point clouds and .pcd files

#include <iostream>
#include "recDataPoint.h"
#include "masterBlock.h"
#include "mbtopcl.h"

#include <cstdint>

#include <stdio.h>
#define _CRT_SECURE_NO_WARNINGS
```

```
//for reading a text file
#include <fstream>
#include <string>
using namespace std;
//for the string to hex conversion (and the backwards '>>')
#include <sstream>

#include <pcl/visualization/cloud_viewer.h> //for cloud viewer
#include <memory>

//for writing to a pcd file
#include <pcl/io/pcd_io.h>
#include <pcl/point_types.h>

boost::shared_ptr<pcl::visualization::PCLVisualizer>
rgbVis(pcl::PointCloud<pcl::PointXYZRGB>::ConstPtr cloud)
{
	// --------------------------------------------
	// -----Open 3D viewer and add point cloud-----
	// --------------------------------------------
	boost::shared_ptr<pcl::visualization::PCLVisualizer> viewer(new
pcl::visualization::PCLVisualizer("3D Viewer"));
	viewer->setBackgroundColor(0, 0, 0);
	pcl::visualization::PointCloudColorHandlerRGBField<pcl::PointXYZRGB>
rgb(cloud);
	viewer->addPointCloud<pcl::PointXYZRGB>(cloud, rgb, "sample cloud");
	viewer-
>setPointCloudRenderingProperties(pcl::visualization::PCL_VISUALIZER_POINT_SIZE,
3, "sample cloud");
	viewer->addCoordinateSystem(1.0);
	viewer->initCameraParameters();
	return (viewer);
}


unsigned int text_id = 0;
void keyboardEventOccurred(const pcl::visualization::KeyboardEvent &event,
	void* viewer_void)
{
	boost::shared_ptr<pcl::visualization::PCLVisualizer> viewer =
*static_cast<boost::shared_ptr<pcl::visualization::PCLVisualizer> *>
(viewer_void);
	if (event.getKeySym() == "r" && event.keyDown())
	{
		std::cout << "r was pressed => removing all text" << std::endl;

		char str[512];
		for (unsigned int i = 0; i < text_id; ++i)
		{
			sprintf(str, "text#%03d", i);
			viewer->removeShape(str);
```

```
            }
            text_id = 0;
      }
}

void mouseEventOccurred(const pcl::visualization::MouseEvent &event,
      void* viewer_void)
{
      boost::shared_ptr<pcl::visualization::PCLVisualizer> viewer =
*static_cast<boost::shared_ptr<pcl::visualization::PCLVisualizer> *>
(viewer_void);
      if (event.getButton() == pcl::visualization::MouseEvent::LeftButton &&
            event.getType() ==
pcl::visualization::MouseEvent::MouseButtonRelease)
      {
            std::cout << "Left mouse button released at position (" <<
event.getX() << ", " << event.getY() << ")" << std::endl;

            char str[512];
            sprintf(str, "text#%03d", text_id++);
            viewer->addText("clicked here", event.getX(), event.getY(), str);
      }
}

int main()
{
      MasterBlock marco = MasterBlock(315,45);
      //MasterBlock marco = MasterBlock(0,359);

      for (int loop=1;loop<336;loop++) //loop through the files called
dataPacket1 through 335
      //for (int loop=1;loop<2;loop++)
      {
            // reading a text file
              string line;

              //converting int to string: (the loop number)
              string Result;          // string which will contain the result
              ostringstream convert;   // stream used for the conversion
              convert << loop;      // insert the textual representation of
'Number' in the characters in the stream
              //converting loop#^^ to a string
              Result = convert.str(); // set 'Result' to the contents of the
stream e.g. now = "123"

              string hereIsMyFileName = "dataPacket" + Result + ".txt";


              ifstream myfile (hereIsMyFileName); //e.g. "dataPacket.txt"
              if (myfile.is_open())
              {
                  while (! myfile.eof() )
```

```
                {
                  getline (myfile,line);
                  cout << line << endl;
                }
                myfile.close();
              }

              else cout << "Unable to open file";

              uint8_t dataIn[1206];
              int spot = 0;


              for (int i=0;i<3618;i+=3) //cause that is how long the variable
"line" is in characters (essentially, length(line))
                    //math checks out: 3618/3 = 1206
              {
                    //create a string from two characters:
                    std::string myByteString = std::string() + line[i+1] +
line[i+2]; //e.g. 'f'+'f' = "ff"

                    //turn a string into a number (accepts many types)
                    uint16_t x;   //for some reason, the function below doesn't
like uint8_t though, but works for uint16_t >> so, I will convert later (see
below).
                    std::stringstream ss;
                    ss << std::hex << myByteString; //<<here's that string I made
                    ss >> x;

                    uint8_t y = (uint8_t)x; //convert from uint16_t to uint8_t >>
will work cause the biggest grabbed values will be a byte

                    dataIn[spot] = y; //load it into our array
                    spot = spot+1;
              }

            DataPacket myPacket = DataPacket(dataIn);

            marco.readNewPacket(myPacket); //the old packet should be destoyed
after every loop, and the default constructor should take effect

            if (marco.isMasterBlockReadyYet()) //stop if the masterBlock is
filled
            {
                  break;
            }

      }//end for-loop

      //^^At this point in code, MasterBlock has now been created and is ready to
use!
```

```
	pcl::PointCloud<pcl::PointXYZRGB> convertedCloud =
mbtopclRGB(marco.getSelectionPointer(),marco.getSelectionSize());
	pcl::PointCloud<pcl::PointXYZRGB>::Ptr cldptr(&convertedCloud);

	//HalfDataBlock * gimmeThe180Data = marco.get180Pointer();

	//HalfDataBlock sel0 = gimmeThe180Data[904];
	//HalfDataBlock sel1 = gimmeThe180Data[15];
	//HalfDataBlock sel2 = gimmeThe180Data[101];
	//HalfDataBlock sel3 = gimmeThe180Data[304];

	//HalfDataBlock * gimmeThe180Data = marco.get180Pointer();

	boost::shared_ptr<pcl::visualization::PCLVisualizer> viewer;
	viewer = rgbVis(cldptr);

	//--------------------
	// -----Main loop-----
	//--------------------

	//WRITE PCD
	//pcl::io::savePCDFileASCII ("ren_90.pcd", convertedCloud);
    //std::cerr << "Saved " << convertedCloud.points.size () << " data points to
test_pcd.pcd." << std::endl;

	//for (size_t i = 0; i < convertedCloud.points.size (); ++i)
	//std::cerr << "    " << convertedCloud.points[i].x << " " <<
convertedCloud.points[i].y << " " << convertedCloud.points[i].z << std::endl;



	//RUN VIEWER
	while (!viewer->wasStopped())
	{
		viewer->spinOnce(100);
		//boost::this_thread::sleep(boost::posix_time::microseconds(100000));
//not sure why this doesn't work...don't need it though
		//pcl_sleep(0.01); //This is added by me (from a Bastian file) -
builds...but has apparently no affect.
	}


	////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////

	//breakpoint so I can see the console window...
	return 0;
}


//Dan Kubik
//Bradley University
```

```
//Department of Electrical and Computer Engineering
//April 11, 2016

//sphDataPoint.cpp - the actual functions that make up the class (defined in the
.h file)

#include <iostream>
#include "sphDataPoint.h"

//constructors
SphDataPoint::SphDataPoint() //default constructor
{
	distance = 0.0;
	reflectivity = 0;
	altitudinalAngle = 0;
}

SphDataPoint::SphDataPoint(double dist, unsigned int refl, unsigned int
laserNumber)
{
	distance = dist; //gimme
	reflectivity = refl; //gimme

	if (laserNumber == 0)
		{altitudinalAngle = -15;}
	else if (laserNumber == 1)
		{altitudinalAngle = 1;}
	else if (laserNumber == 2)
		{altitudinalAngle = -13;}
	else if (laserNumber == 3)
		{altitudinalAngle = 3;}//the one mistyped on datasheet (fixed here)
	else if (laserNumber == 4)
		{altitudinalAngle = -11;}
	else if (laserNumber == 5)
		{altitudinalAngle = 5;}
	else if (laserNumber == 6)
		{altitudinalAngle = -9;}
	else if (laserNumber == 7)
		{altitudinalAngle = 7;}
	else if (laserNumber == 8)
		{altitudinalAngle = -7;}
	else if (laserNumber == 9)
		{altitudinalAngle = 9;}
	else if (laserNumber == 10)
		{altitudinalAngle = -5;}
	else if (laserNumber == 11)
		{altitudinalAngle = 11;}
	else if (laserNumber == 12)
		{altitudinalAngle = -3;}
	else if (laserNumber == 13)
		{altitudinalAngle = 13;}
	else if (laserNumber == 14)
```

```
            {altitudinalAngle = -1;}
      else if (laserNumber == 15)
            {altitudinalAngle = 15;}
      else
            {altitudinalAngle = 0;} //if this happens, bad on me
}

SphDataPoint::~SphDataPoint() //destructor
{
      //nothing! yay
}

//methods (functions)

double SphDataPoint::getDistance()
{
      return distance;
}

unsigned int SphDataPoint::getReflectivity()
{
      return reflectivity;
}

signed int SphDataPoint::getAltitudinalAngle()
{
      return altitudinalAngle;
}



//Dan Kubik
//Bradley University
//Department of Electrical and Computer Engineering
//April 11, 2016

//sphDataPoint.cpp - the actual functions that make up the class (defined in the
.h file)

#include <iostream>
#include "recDataPoint.h"
#include <math.h>   //for sin/cos
#define PI 3.14159265

//constructors
RecDataPoint::RecDataPoint() //default constructor
{
      X = 0;
      Y = 0;
      Z = 0;
      reflectivity = 0;
}
```

```
RecDataPoint::RecDataPoint(SphDataPoint point, double azimuth)
{
	X =
point.getDistance()*cos(point.getAltitudinalAngle()*PI/180)*sin(azimuth*PI/180);
	Y =
point.getDistance()*cos(point.getAltitudinalAngle()*PI/180)*cos(azimuth*PI/180);
	Z = point.getDistance()*sin(point.getAltitudinalAngle()*PI/180);
	reflectivity = point.getReflectivity();
}

RecDataPoint::~RecDataPoint() //destructor
{
	//nothing! yay
}

//methods (functions)

double RecDataPoint::getX()
{
	return X;
}
double RecDataPoint::getY()
{
	return Y;
}
double RecDataPoint::getZ()
{
	return Z;
}
unsigned int RecDataPoint::getReflectivity()
{
	return reflectivity;
}



//Dan Kubik
//Bradley University
//Department of Electrical and Computer Engineering
//April 11, 2016

//halfDataBlock.cpp

#include <iostream>
#include "halfDataBlock.h"

//constructors
HalfDataBlock::HalfDataBlock() //default constructor
{
	azimuth = 0.0;

	for (int i=0;i<16;i++) //counts 0 to 15
```

```
	{dataPoints[i]=SphDataPoint();} //construct a SphDataPoint (16 times)
                        //  ^^^using default constructor - exciting!

}

HalfDataBlock::HalfDataBlock(uint8_t data[], double azimuthAngle) //receiving 48
bytes of unsorted data...(here is where the sorting will go down)
{
	//const int sizeOfData = 48; //cause 16*3 bytes each (2 distance, 1
reflectivity)

	azimuth = azimuthAngle; //pass along value

	for (int i=0;i<16;i++) //counts 0 to 15 for each of the SphDataPoint's it's
going to create
	{
		double distTemp = ((double)((data[(i+1)*3-2]<<8) + data[(i+1)*3-
3]))/100;
		unsigned int reflTemp = (int)(data[(i+1)*3-1]);
		dataPoints[i] = SphDataPoint(distTemp, reflTemp, i); //create a new
SphDataPoint (x16)
	}

}

HalfDataBlock::~HalfDataBlock() //destructor
{
	//nothing! yay (cause never used new/delete)
}

//methods (functions)
SphDataPoint HalfDataBlock::getDataPoint(unsigned int n)
{
	return dataPoints[n];
}

double HalfDataBlock::getAzimuth()
{
	return azimuth;
}



//Dan Kubik
//Bradley University
//Department of Electrical and Computer Engineering
//April 11, 2016

//dataBlock.cpp

#include <iostream>
#include "dataBlock.h"
```

```
//constructors
DataBlock::DataBlock() //default constructor
{
	for (int i=0;i<2;i++) //counts 0 to 1
	{halfBlocks[i]=HalfDataBlock();} //construct a HalfDataBlock (2 times)
			//  ^^^using default constructor - exciting!
}

DataBlock::DataBlock(uint8_t data[], double nextAzimuth) //receiving 98 bytes of
unsorted data
{
	//const int sizeOfData = 98; //cause (16*3 bytes each (2 distance, 1
reflectivity) *2) + 2 for leading azimuth\

	double currentAzimuth = ((double)((data[1]<<8) + data[0]))/100; //will
always be first two values in data array
	double derivedAzimuth = 0;

	//first, adjust for a rollover from 359.99 to 0 degrees
	if (nextAzimuth < currentAzimuth)
	{nextAzimuth = nextAzimuth+360;}

	//perform the interpolation
	derivedAzimuth = currentAzimuth + ((nextAzimuth - currentAzimuth)/2);

	//correct for any rollover over from 359.99 to 0 degrees
	if (derivedAzimuth>360)
	{derivedAzimuth = derivedAzimuth-360;}

	halfBlocks[0] = HalfDataBlock(&data[2], currentAzimuth); //create a new
HalfDataBlock //points to first distance measurement in data array
	halfBlocks[1] = HalfDataBlock(&data[50], derivedAzimuth); //create the
second HalfDataBlock //points to 48 bytes later than 1st halfBlock

}

DataBlock::~DataBlock() //destructor
{
	//nothing! yay (cause never used new/delete)
}

//methods (functions)
HalfDataBlock DataBlock::getHalfDataBlock(unsigned int n)
{
	if (n<2)
	{return halfBlocks[n];}
	else
	{//something's wrong! (my bad)
		//not sure how we want to have an error signal pop up...but if I did,
it'd go here
		return HalfDataBlock();
```

```
	}
}



//Dan Kubik
//Bradley University
//Department of Electrical and Computer Engineering
//April 11, 2016

//dataPacket.cpp

#include <iostream>
#include "dataPacket.h"

//constructors
DataPacket::DataPacket() //default constructor
{
	for (int i=0;i<12;i++) //counts 0 to 11
	{blocks[i]=DataBlock();} //construct a DataBlock (12 times)
				//  ^^^using default constructor - exciting!
	firstAzimuth = 0;
	lastAzimuth = 0;
	timeStamp = 0;
}

DataPacket::DataPacket(uint8_t data[]) //receiving 1206 bytes of unsorted data
(header removed)
{	//alternate constructor

	for (int i=0;i<12;i++)
	{
		if (i<11)
		{
			blocks[i] = DataBlock(&data[(100*i)+2],
(((double)((data[(100*(i+1)+3)]<<8) + data[100*(i+1)+2]))/100));
			//^^ sends 98 bytes and the azimuth of the next block
		}
		else //i==11
		{
			double previousBlockAzimuth = (((double)((data[(100*(i-
1)+3)]<<8) + data[100*(i-1)+2]))/100);
			double currentBlockAzimuth = (((double)((data[(100*i)+3]<<8) +
data[(100*i)+2]))/100);
			double estimatedNextAzimuth = 0;

			if (currentBlockAzimuth>previousBlockAzimuth) //did not cross
the 360 threshold
			{
				estimatedNextAzimuth = currentBlockAzimuth*2 -
previousBlockAzimuth;
			}
```

```
                    else //it moved around the 360 hump
                    {
                          estimatedNextAzimuth = currentBlockAzimuth*2 + 360 -
previousBlockAzimuth;
                    }

                    if (estimatedNextAzimuth >=360)
                    {estimatedNextAzimuth = estimatedNextAzimuth-360;}

                    blocks[i] = DataBlock(&data[(100*i)+2], estimatedNextAzimuth);
             }
      }

      firstAzimuth = blocks[0].getHalfDataBlock(0).getAzimuth();
      lastAzimuth = blocks[11].getHalfDataBlock(1).getAzimuth();
      timeStamp = (double)((data[1203]<<24) + (data[1202]<<16) + (data[1201]<<8)
+ data[1200]);

}

DataPacket::~DataPacket() //destructor
{
      //nothing! yay (cause never used new/delete)
}

//methods (functions)
DataBlock DataPacket::getDataBlock(unsigned int n)
{
      return blocks[n]; //must be within 0 to 11
}

double DataPacket::getTimeStamp()
{
      return timeStamp;
}

double DataPacket::getLastAzimuth()
{
      return lastAzimuth;
}
double DataPacket::getFirstAzimuth()
{
      return firstAzimuth;
}

//Dan Kubik
//Bradley University
//Department of Electrical and Computer Engineering
//April 11, 2016

//masterBlock.cpp
```

```
#include <iostream>
#include "masterBlock.h"
#define FRONTRANGE 180 //desired range is 180 degrees in front of boat

//constructors
MasterBlock::MasterBlock() //default constructor
{
	desiredStartingAngle = 0;
	desiredEndingAngle = 0;
	ourStartAngle = 0;
	doneFlag = false;
	firstAssignedYet = false;
	pointer180 = 0; //points to the array
	pointerSelection = 0; //points to the array
	howMany180HalfBlocks = 0; //essentially a counter
	howManySelectionHalfBlocks = 0; //essentially a counter
	LastAzimuthFromLastPacket = 0;
}

MasterBlock::MasterBlock(double startAngle, double endAngle)
{
	desiredStartingAngle = startAngle;
	desiredEndingAngle = endAngle;
	ourStartAngle = 0; //no clue yet
	doneFlag = false;
	firstAssignedYet = false;

	double selectionSize;

	if (endAngle < startAngle)
	{selectionSize = 360+endAngle-startAngle;}
	else
	{selectionSize = endAngle-startAngle;}

	int size180 = ((int)(FRONTRANGE/0.1))+10;//10 additional scan overflow
safeties
	int selection = ((int)(selectionSize/0.1))+10;//10 additional scan overflow
safeties

	//pointer = new type [number_of_elements]
	pointer180 = new HalfDataBlock [size180];//if range is 180, this will be
1810
	pointerSelection = new HalfDataBlock [selection];
	howMany180HalfBlocks = 0; //to be added to later
	howManySelectionHalfBlocks = 0; //to be added to later

	LastAzimuthFromLastPacket = -1;//we haven't read anything yet...
}

MasterBlock::~MasterBlock() //destructor
{
	delete [] pointer180;
```

```
	delete [] pointerSelection;
}

//methods (functions)
bool MasterBlock::isMasterBlockReadyYet()
{
	return doneFlag;
}

HalfDataBlock * MasterBlock::get180Pointer()
{
	return pointer180;
}
HalfDataBlock * MasterBlock::getSelectionPointer()
{
	return pointerSelection;
}

unsigned int MasterBlock::get180Size()
{
	return howMany180HalfBlocks;
}

unsigned int MasterBlock::getSelectionSize()
{
	return howManySelectionHalfBlocks;
}

void MasterBlock::readNewPacket(DataPacket newPacket) //receiving 1206 bytes of
unsorted data (header removed)
{

	if (doneFlag==false)//if it's 'TRUE', we shouldn't want to read any more
data for this MasterBlock
	{

		bool firstTime = false; //asume at start
		bool originalInThisPacket = false; //asume at start
		bool wePassedIt = false; //asume at start

		if (firstAssignedYet==false) //check if this is our first packet or
not
		{
			ourStartAngle = newPacket.getFirstAzimuth(); //here is the
azimuth we start with!
			firstAssignedYet=true; //override flag
			firstTime = true; //to remember for the rest of this packet
reading
			//LastAzimuthFromLastPacket = -1; //I think this should still
not have a known value...
		}
```

```
            //check: is the original azimuth in THIS (newly received)
dataPacket?? (not the first packet)
            if (firstTime==false) //not the first packet read in the masterBlock
            {

////////////////////////////////////////////////////////////////////////////////
////////////
                  //check for if the end azimuth to look for is in current packet
                  if (newPacket.getLastAzimuth() < newPacket.getFirstAzimuth())
//the packet straddles the 0 degree mark
                  {
                        if(ourStartAngle>=newPacket.getFirstAzimuth())
                        {originalInThisPacket = true;} //get ready to shut down
the read function of this packet
                        else if(ourStartAngle<=newPacket.getLastAzimuth())
                        {originalInThisPacket = true;} //get ready to shut down
the read function of this packet
                  }
                  else //normal packet: does not straddle 0 degree mark
                  {
                        if(ourStartAngle>=newPacket.getFirstAzimuth() &&
ourStartAngle<=newPacket.getLastAzimuth())
                        {originalInThisPacket = true;} //get ready to shut down
the read function of this packet
                  }

      ////////////////////////////////////////////////////////////////////////
///////
                  //now, check for the strange case that the original azimuth has
been missed;
                  //it happened to fall between two separate packets the second
time around
                  if (newPacket.getFirstAzimuth() <
LastAzimuthFromLastPacket)//this packet and the last packet straddles the 0 degree
mark
                  {
                        if(ourStartAngle>=LastAzimuthFromLastPacket) //shouldn't
be equal to...but w/e couldn't hurt
                        {wePassedIt = true;}
                        else if(ourStartAngle<=newPacket.getFirstAzimuth())
                        {wePassedIt = true;}
                  }
                  else //this packet and last packet do not straddle 0 mark
                  {
                        if(ourStartAngle>=LastAzimuthFromLastPacket &&
ourStartAngle<=newPacket.getFirstAzimuth())
                        {wePassedIt = true;}
                  }
            }
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////
```

```
//Now for the big question:

            //Do we want this information?
            //(loop through all information and check to see if this data fits in
either of the 180 and selected ranges
            if (wePassedIt == false)//if it is TRUE, we do NOT want to add it to
this MasterBlock
            {
                  for (int blockCounter=0;blockCounter<12;blockCounter++) //loop
through blocks (0-11)
                  {
                        if (doneFlag==false) //because it can become true inside
of this loop
                        {
                              double temp0 =
newPacket.getDataBlock(blockCounter).getHalfDataBlock(0).getAzimuth(); //known
azimuth
                              double temp1 =
newPacket.getDataBlock(blockCounter).getHalfDataBlock(1).getAzimuth();
//interpolated one

                              if (originalInThisPacket==true) //we need to be
cautious //notice: (only true beyond 1st packet)
                              {
                                    if(newPacket.getLastAzimuth() <
newPacket.getFirstAzimuth())//this packet straddles 0 degree mark
                                    {

      if(temp0<LastAzimuthFromLastPacket)//We have entirely passed 0 degrees
(since the last block in this packet)
                                          {//somewhat of a misnomer^^^: here it
technically means lastAzimuthRead (old temp1)

      if(ourStartAngle>350)//originalAzimuth was on the 350s "high" side (not our
side - we have already passed it)
                                                { //(this shouldn't happen
lol). Since original is in this packet, and this is only true on after a full 360
                                                      //degrees rotation, if we
have gone beyond it and hit 0, something is up
                                                      doneFlag==true; //kill it
for sure this time
                                                }
                                                else //originalAzimtuh is on
the 0s-10s side (our side) of the 0 degree (we have to look out for it)
                                                {

      if(ourStartAngle<=temp0)//it is in between this temp0 and the last temp1
(or == temp0)
                                                      {//aka we want nothing
anymore! Kill the read function!
                                                            doneFlag = true;
                                                      }
```

```
                                                        else
if(ourStartAngle>temp0 && ourStartAngle<=temp1)//it is in between temp0 and temp1
or is temp1
                                                        {//we still need to check
temp0, but temp1 is unwanted

      if(check180Range(temp0)==true)

      {addToMasterBlock(newPacket.getDataBlock(blockCounter).getHalfDataBlock(0),
true);}


      if(checkSelectedRange(temp0)==true)

      {addToMasterBlock(newPacket.getDataBlock(blockCounter).getHalfDataBlock(0),
false);}

                                                              doneFlag = true;
//kill read function!
                                                        }
                                                        else //it's in this
packet, but NOT near these two azimuths (check as normal)
                                                        {

      if(check180Range(temp0)==true)

      {addToMasterBlock(newPacket.getDataBlock(blockCounter).getHalfDataBlock(0),
true);}


      if(checkSelectedRange(temp0)==true)

      {addToMasterBlock(newPacket.getDataBlock(blockCounter).getHalfDataBlock(0),
false);}


      if(check180Range(temp1)==true)

      {addToMasterBlock(newPacket.getDataBlock(blockCounter).getHalfDataBlock(1),
true);}


      if(checkSelectedRange(temp1)==true)

      {addToMasterBlock(newPacket.getDataBlock(blockCounter).getHalfDataBlock(1),
false);}
                                                        }
                                                  }
                                             }
                                             else if(temp0>temp1)//0 degrees is
smack dab between temp0 and temp1
                                             {
```

```
      if(ourStartAngle>350)//originalAzimuth was on the 350s "high" side
                                                {

      if(ourStartAngle<=temp0)//we've passed it
                                                      {doneFlag==true; //kill
it if it isn't already dead
                                                      }
                                                      else//temp0 is still
valuable
                                                      {

      if(check180Range(temp0)==true)

      {addToMasterBlock(newPacket.getDataBlock(blockCounter).getHalfDataBlock(0),
true);}


      if(checkSelectedRange(temp0)==true)

      {addToMasterBlock(newPacket.getDataBlock(blockCounter).getHalfDataBlock(0),
false);}

                                                            doneFlag=true;
//now kill it
                                                      }
                                                }
                                                else //originalAzimtuh is on
the 0s-10s side
                                                {

      if(ourStartAngle>temp1)//both temp1 and temp0 are valuable (we haven't hit
it yet)
                                                      {

      if(check180Range(temp0)==true)

      {addToMasterBlock(newPacket.getDataBlock(blockCounter).getHalfDataBlock(0),
true);}


      if(checkSelectedRange(temp0)==true)

      {addToMasterBlock(newPacket.getDataBlock(blockCounter).getHalfDataBlock(0),
false);}


      if(check180Range(temp1)==true)

      {addToMasterBlock(newPacket.getDataBlock(blockCounter).getHalfDataBlock(1),
true);}
```

```
      if(checkSelectedRange(temp1)==true)

      {addToMasterBlock(newPacket.getDataBlock(blockCounter).getHalfDataBlock(1),
false);}
                                                                  }
                                                                  else //
(ourStartAngle<=temp1) //only temp0 is valuable
                                                                  {

      if(check180Range(temp0)==true)

      {addToMasterBlock(newPacket.getDataBlock(blockCounter).getHalfDataBlock(0),
true);}


      if(checkSelectedRange(temp0)==true)

      {addToMasterBlock(newPacket.getDataBlock(blockCounter).getHalfDataBlock(0),
false);}

                                                                        doneFlag = true;
//kill it
                                                                  }
                                                            }
                                                      }
                                                      else if(temp0==0)
                                                      {

      if(ourStartAngle>350)//originalAzimuth was on the 350s "high" side
                                                            {//we've already passed it
                                                                  doneFlag=true;
                                                            }
                                                            else //originalAzimtuh is on
the 0s-10s side
                                                            {

      if(ourStartAngle>temp1)//both temp1 and temp0 are valuable (we haven't hit
it yet)
                                                                  {

      if(check180Range(temp0)==true)

      {addToMasterBlock(newPacket.getDataBlock(blockCounter).getHalfDataBlock(0),
true);}


      if(checkSelectedRange(temp0)==true)

      {addToMasterBlock(newPacket.getDataBlock(blockCounter).getHalfDataBlock(0),
false);}
```

```
      if(check180Range(temp1)==true)

      {addToMasterBlock(newPacket.getDataBlock(blockCounter).getHalfDataBlock(1),
true);}


      if(checkSelectedRange(temp1)==true)

      {addToMasterBlock(newPacket.getDataBlock(blockCounter).getHalfDataBlock(1),
false);}
                                                            }
                                                            else //
(ourStartAngle<=temp1) //only temp0 is valuable
                                                            {

      if(check180Range(temp0)==true)

      {addToMasterBlock(newPacket.getDataBlock(blockCounter).getHalfDataBlock(0),
true);}


      if(checkSelectedRange(temp0)==true)

      {addToMasterBlock(newPacket.getDataBlock(blockCounter).getHalfDataBlock(0),
false);}

                                                                  doneFlag = true;
                                                            }
                                                      }
                                                }
                                                else if(temp1==0)
                                                {

      if(ourStartAngle>350)//originalAzimuth was on the 350s "high" side
                                                      {
                                                            if(ourStartAngle <=
temp0)//we've already passed it
                                                            {doneFlag=true;
                                                            }
                                                            else //temp0 is valuable
temp1 forgetaboutit
                                                            {

      if(check180Range(temp0)==true)

      {addToMasterBlock(newPacket.getDataBlock(blockCounter).getHalfDataBlock(0),
true);}


      if(checkSelectedRange(temp0)==true)
```

```
      {addToMasterBlock(newPacket.getDataBlock(blockCounter).getHalfDataBlock(0),
false);}

                                                                  doneFlag = true;
                                                            }
                                                      }
                                                      else //originalAzimtuh is on
the 0s-10s side
                                                      {
                                                            if (ourStartAngle >=
temp1)//temp0 is valuable
                                                            {

      if(check180Range(temp0)==true)

      {addToMasterBlock(newPacket.getDataBlock(blockCounter).getHalfDataBlock(0),
true);}


      if(checkSelectedRange(temp0)==true)

      {addToMasterBlock(newPacket.getDataBlock(blockCounter).getHalfDataBlock(0),
false);}

                                                                  doneFlag = true;
                                                            }
                                                            else //neither temp0 or
temp1 is valuable
                                                            {doneFlag=true;
                                                            }
                                                      }
                                                }
                                                else //we have not yet hit 0 degrees,
though it will come sometime in this packet
                                                {//treat it like normal...
                                                      if(ourStartAngle<=temp0)//it is
in between this temp0 and the last temp1 (or temp0)
                                                      {//aka we want nothing anymore!
Kill the read function!
                                                            doneFlag = true;
                                                      }
                                                      else if(ourStartAngle>temp0 &&
ourStartAngle<=temp1)//it is in between temp0 and temp1 or is temp1
                                                      {//we still need to check
temp0, but temp1 is unwanted

      if(check180Range(temp0)==true)

      {addToMasterBlock(newPacket.getDataBlock(blockCounter).getHalfDataBlock(0),
true);}
```

```
      if(checkSelectedRange(temp0)==true)

      {addToMasterBlock(newPacket.getDataBlock(blockCounter).getHalfDataBlock(0),
false);}

                                                            doneFlag = true; //kill
read function!
                                                      }
                                                      else //it's in this packet, but
NOT near these two azimuths (check as normal)
                                                      {

      if(check180Range(temp0)==true)

      {addToMasterBlock(newPacket.getDataBlock(blockCounter).getHalfDataBlock(0),
true);}


      if(checkSelectedRange(temp0)==true)

      {addToMasterBlock(newPacket.getDataBlock(blockCounter).getHalfDataBlock(0),
false);}


      if(check180Range(temp1)==true)

      {addToMasterBlock(newPacket.getDataBlock(blockCounter).getHalfDataBlock(1),
true);}


      if(checkSelectedRange(temp1)==true)

      {addToMasterBlock(newPacket.getDataBlock(blockCounter).getHalfDataBlock(1),
false);}
                                                      }
                                                }
                                          }
                                          else//this packet does NOT straddle 0 degree
mark (yay)
                                          {
                                                if(ourStartAngle<=temp0)//it is in
between this temp0 and the last temp1 (or == temp0)
                                                {//aka we want nothing anymore! Kill
the read function!
                                                      doneFlag = true;
                                                }
                                                else if(ourStartAngle>temp0 &&
ourStartAngle<=temp1)//it is in between temp0 and temp1 or is temp1
                                                {//we still need to check temp0, but
temp1 is unwanted
                                                      if(check180Range(temp0)==true)
```

```
      {addToMasterBlock(newPacket.getDataBlock(blockCounter).getHalfDataBlock(0),
true);}


      if(checkSelectedRange(temp0)==true)

      {addToMasterBlock(newPacket.getDataBlock(blockCounter).getHalfDataBlock(0),
false);}

                                                      doneFlag = true; //kill read
function!
                                                }
                                                else //it's in this packet, but NOT
near these two azimuths (check as normal)
                                                {
                                                      if(check180Range(temp0)==true)

      {addToMasterBlock(newPacket.getDataBlock(blockCounter).getHalfDataBlock(0),
true);}


      if(checkSelectedRange(temp0)==true)

      {addToMasterBlock(newPacket.getDataBlock(blockCounter).getHalfDataBlock(0),
false);}

                                                      if(check180Range(temp1)==true)

      {addToMasterBlock(newPacket.getDataBlock(blockCounter).getHalfDataBlock(1),
true);}


      if(checkSelectedRange(temp1)==true)

      {addToMasterBlock(newPacket.getDataBlock(blockCounter).getHalfDataBlock(1),
false);}
                                                }
                                          }
                                    }
                                    else //we don't need to be cautious...check as
normal...
                                    {
                                          if(check180Range(temp0)==true)

      {addToMasterBlock(newPacket.getDataBlock(blockCounter).getHalfDataBlock(0),
true);}

                                          if(checkSelectedRange(temp0)==true)

      {addToMasterBlock(newPacket.getDataBlock(blockCounter).getHalfDataBlock(0),
false);}
```

```
                                    if(check180Range(temp1)==true)

      {addToMasterBlock(newPacket.getDataBlock(blockCounter).getHalfDataBlock(1),
true);}

                                    if(checkSelectedRange(temp1)==true)

      {addToMasterBlock(newPacket.getDataBlock(blockCounter).getHalfDataBlock(1),
false);}
                              }

                              //need to update this: LastAzimuthFromLastPacket -
updates every time cause we will use it for other stuff too (see misnomer)
                              LastAzimuthFromLastPacket = temp1;
                        }
                  }//<<<end for loop
            }
            else //wePassedIt==true
            {doneFlag=true;//kill the read functionality!
            }

      }//end doneFlag==false
}


void MasterBlock::addToMasterBlock(HalfDataBlock entry, bool trueIs180)
{
      if (trueIs180==true)//add to 180 block
      {
            pointer180[howMany180HalfBlocks] = entry;
            howMany180HalfBlocks = howMany180HalfBlocks + 1; //essentially a
counter
      }
      else//add to selection block
      {
            pointerSelection[howManySelectionHalfBlocks] = entry;
            howManySelectionHalfBlocks = howManySelectionHalfBlocks + 1;
      }
}

bool MasterBlock::check180Range(double halfBlockAzimuth)
{//the assumption is that this is a value still within the 360 range
      double halfRange = FRONTRANGE/2.0;

      if((halfBlockAzimuth >= 360-halfRange) || (halfBlockAzimuth <= halfRange))
//yes, in "180" range
      {return true;}
      else
      {return false;}
}
```

```
bool MasterBlock::checkSelectedRange(double halfBlockAzimuth)
{//the assumption is that this is a value still within the 360 range

	if(desiredStartingAngle>desiredEndingAngle)//selected range straddles 0
degrees
	{
		if(halfBlockAzimuth>=desiredStartingAngle ||
halfBlockAzimuth<=desiredEndingAngle)
		{return true;}
		else
		{return false;}
	}
	else //selected range is "normal" and does not straddle 0 degrees
	{
		if(halfBlockAzimuth>=desiredStartingAngle &&
halfBlockAzimuth<=desiredEndingAngle)
		{return true;}
		else
		{return false;}
	}
}



//Dan Kubik
//Bradley University
//Department of Electrical and Computer Engineering
//April 11, 2016

//mbtopcl.cpp

#include <iostream>
#include "mbtopcl.h"
#include <pcl/io/pcd_io.h>
#include <pcl/point_types.h>


//function mbtopcl:
//Converts array HalfDataBlocks to XYZ (recDataPoints) to to PCL point cloud of
type XYZ
pcl::PointCloud<pcl::PointXYZ> mbtopcl(HalfDataBlock * hdbPointer, unsigned int
size)
{
	pcl::PointCloud<pcl::PointXYZ> cloud; //creates type XYZI type of PCL point
cloud (can be changed later...)

	//Change cloud properties:
	cloud.width    = size*16; //because 16 points per HalfDataBlock
	cloud.height   = 1; //set to 1 for unorganized datasets (aka ours)
	cloud.is_dense = false; //no points will have infinite as their value
	cloud.points.resize (cloud.width * cloud.height);

	int cloudTracker = 0;
```

```
	for (int i1 = 0; i1<size; i1++) //loop through the HalfDataBlocks (through
main array of them)
	{
		HalfDataBlock hdb = hdbPointer[i1];
		for (int i2 = 0; i2<16; i2++) //looping through the 16 sphDataBlocks
		{
			SphDataPoint newSphPoint = hdb.getDataPoint(i2); //extract
points 0-15
			RecDataPoint newRecPoint =
RecDataPoint(newSphPoint,hdb.getAzimuth()); //create X,Y,Z,R rectangular Data
Point

			cloud.points[cloudTracker].x = newRecPoint.getX();
			cloud.points[cloudTracker].y = newRecPoint.getY();
			cloud.points[cloudTracker].z = newRecPoint.getZ();

			cloudTracker = cloudTracker +1; //keep looping through the PCL
cloud - no real organization...
		}
	}

	return cloud;
}

pcl::PointCloud<pcl::PointXYZI> mbtopclI(HalfDataBlock * hdbPointer, unsigned int
size)
{
	pcl::PointCloud<pcl::PointXYZI> cloud; //creates type XYZI type of PCL
point cloud (can be changed later...)

	//Change cloud properties:
	cloud.width    = size*16; //because 16 points per HalfDataBlock
	cloud.height   = 1; //set to 1 for unorganized datasets (aka ours)
	cloud.is_dense = false; //no points will have infinite as their value
	cloud.points.resize (cloud.width * cloud.height);

	int cloudTracker = 0;


	for (int i1 = 0; i1<size; i1++) //loop through the HalfDataBlocks (through
main array of them)
	{
		HalfDataBlock hdb = hdbPointer[i1];
		for (int i2 = 0; i2<16; i2++) //looping through the 16 sphDataBlocks
		{
			SphDataPoint newSphPoint = hdb.getDataPoint(i2); //extract
points 0-15
			RecDataPoint newRecPoint =
RecDataPoint(newSphPoint,hdb.getAzimuth()); //create X,Y,Z,R rectangular Data
Point
```

```
                    cloud.points[cloudTracker].x = newRecPoint.getX();
                    cloud.points[cloudTracker].y = newRecPoint.getY();
                    cloud.points[cloudTracker].z = newRecPoint.getZ();

                    cloud.points[cloudTracker].intensity =
newRecPoint.getReflectivity();

                    cloudTracker = cloudTracker +1; //keep looping through the PCL
cloud - no real organization...
             }
      }

      return cloud;
}

pcl::PointCloud<pcl::PointXYZRGB> mbtopclRGB(HalfDataBlock * hdbPointer, unsigned
int size)
{
      pcl::PointCloud<pcl::PointXYZRGB> cloud; //creates type XYZI type of PCL
point cloud (can be changed later...)

      //Change cloud properties:
      cloud.width    = size*16; //because 16 points per HalfDataBlock
      cloud.height   = 1; //set to 1 for unorganized datasets (aka ours)
      cloud.is_dense = false; //no points will have infinite as their value
      cloud.points.resize (cloud.width * cloud.height);

      int cloudTracker = 0;


      for (int i1 = 0; i1<size; i1++) //loop through the HalfDataBlocks (through
main array of them)
      {
            HalfDataBlock hdb = hdbPointer[i1];
            for (int i2 = 0; i2<16; i2++) //looping through the 16 sphDataBlocks
            {
                    SphDataPoint newSphPoint = hdb.getDataPoint(i2); //extract
points 0-15
                    RecDataPoint newRecPoint =
RecDataPoint(newSphPoint,hdb.getAzimuth()); //create X,Y,Z,R rectangular Data
Point

                    cloud.points[cloudTracker].x = newRecPoint.getX();
                    cloud.points[cloudTracker].y = newRecPoint.getY();
                    cloud.points[cloudTracker].z = newRecPoint.getZ();

                    //making up a color scheme: lower reflectivity = red, highest
reflectivityfarthest = blue
                    cloud.points[cloudTracker].r = 255 -
newRecPoint.getReflectivity();
                    cloud.points[cloudTracker].g = 0;
```

```
                    cloud.points[cloudTracker].b = newRecPoint.getReflectivity();

                    cloudTracker = cloudTracker +1; //keep looping through the PCL
cloud - no real organization...
                }
        }

        return cloud;
}

//Alternate way to load a point:

        //pcl::PointXYZRGB o;
        //o.x = newRecPoint.getX();
        //o.y = newRecPoint.getY();
        //o.z = newRecPoint.getZ();
        //o.r = 255 - newRecPoint.getReflectivity();
        //o.g = 0;
        //o.b = newRecPoint.getReflectivity();

        //cloud.points[cloudTracker] = o;


%{
    Dan Kubik
    ECE 498
    Senior Project

    Designed to read a Wireshark formatted text file and create multiple
    files each based on a packet that can be sent in to my C++ program
    involving MasterBlocks

    February 9, 2016

    createTexts.m
%}

clc;
close all;
clear all;
%% Initial file read and data extraction
readname = 'C:\frameGrab.txt';

fileID_read = fopen(readname,'r');

scanIn = textscan(fileID_read,'%s'); %saves each "word" as a string
scanIn = scanIn{1};

fclose(fileID_read);
%% Writing to Multiple Files
writeCounter =0;

for loop=1:length(scanIn)
```

```
    tempScanIn = scanIn(loop);
    tempScanIn = tempScanIn{1}; %converted to string
    if ~isempty(strfind(tempScanIn, '|ff|ff|ff|')) %there is an
|ff|ff|ff|...blah etc.
        temp = strfind(tempScanIn, '|ff|ee|'); %looking for our starting
point
        if ~isempty(temp)%(it exists)
%             temp = temp{1};%convert into array of doubles
            writeCounter = writeCounter + 1; %starts at file#1
            writename = ['C:\dataPacket', num2str(writeCounter), '.txt'];
            fileID_write = fopen(writename,'w');

            stringToWrite = tempScanIn;
            stringToWrite = stringToWrite(temp(1):length(tempScanIn));

            fprintf(fileID_write,stringToWrite);

            fclose(fileID_write);
        end
    end
end
```

Data Processing:

The following code was used to create the overlay image using Barnes interpolation and to implement the SIFT algorithm. Subsequent iterations of the Barnes interpolation are included in the file, but were not found to be necessary for the lidar application, as they did not significantly improve the quality of the image and created long run time of the code.

The SIFT file requires PCL to run.

```
%{
Dan Kubik
Bradley University
Department of Electrical and Computer Engineering
4/11/2016

barnesInterpolation.m

Testing the Barnes Interpolation/Analysis Method
Will import PCD Data using pcl functions I found online and writing the
barnes algorithm to interpolate data in a grid.

After reading Barnes' 1964 paper:
"A Technique for Maximizing Details in Numerical Weather Map Analysis"
%}

clc;
clear all;
close all;
```

```
% answer = loadpcd('ball_and_chair_45.pcd'); %it prints in the command window
the type of PCD
answer = loadpcd('ren_90.pcd'); %it prints in the command window the type of
PCD
X = answer(1,:);
Y = answer(2,:);
Z = answer(3,:);
R = answer(4,:);
G = answer(5,:);
B = answer(6,:);

% gridSizeX = 27;
% gridSizeZ = 18;
gridSizeX = 500; %just using X - See Below
% gridSizeZ = 54;

%% Desampling for improving interpolation speed

%Every 5?
sizeX = size(X);
sizeX = sizeX(2);
count = 1;
for loop = 1:5:sizeX
    newX(count) = X(loop);
    newZ(count) = Z(loop);
    newY(count) = Y(loop);
    newR(count) = R(loop);
    count = count + 1;
end

X = newX;
Z = newZ;
Y = newY;
R = newR;

v = Y;

%% Removing "Floor"/water by a filter of height

sizeX = size(X);
sizeX = sizeX(2);
count = 1;
for loop = 1:sizeX
    if (Z(loop)>=-0.46) %<< Using value since I know how tall scanner was on
chair
        %(It's good!)...so add
        newX2(count) = X(loop);
```

```
        newZ2(count) = Z(loop);
        newY2(count) = Y(loop);
        newR2(count) = R(loop);
        count = count + 1;
    end

end

X = newX2;
Z = newZ2;
Y = newY2;
R = newR2;

v = Y;

%%
% v = Y;
maxX = max(X);
maxZ = max(Z);
minX = min(X);
minZ = min(Z);
rangeZ = maxZ-minZ;
rangeX = maxX-minX;
sizeX = size(X);
sizeZ = size(Z);
sizeZ = sizeZ(2);
colors = zeros(sizeX(2),3);


%% Begin regular stuff

ratio1 = rangeX/gridSizeX;
gridSizeZ = round(rangeZ/ratio1);

xpoints = linspace(minX, maxX, gridSizeX);
zpoints = linspace(minZ, maxZ, gridSizeZ);



M = sizeX(2);
k = 5;

pointSetX = zeros(1, gridSizeX*gridSizeZ);
pointSetZ = zeros(1, gridSizeX*gridSizeZ);

plotcount = 1;
```

```
%CREATE GRID
for loop = 1:gridSizeX
    for loop2 = 1:gridSizeZ
        pointSetX(plotcount) = xpoints(loop);
        pointSetZ(plotcount) = zpoints(loop2);
        plotcount = plotcount+1;
    end
end

gridSize = gridSizeX*gridSizeZ;
firstGuess = zeros(gridSize,1); %initial guess!

%ACTUAL INTERPOLATION
for gLoop = 1:gridSize
    numSum = 0;
    denSum = 0;
    for jay = 1:M %X and Z are same size
        r = sqrt((pointSetX(gLoop)-X(jay))^2 + (pointSetZ(gLoop)-Z(jay) )^2);
%distance to that point
        nthing = exp((-1*r^2)/4*k);
        numSum = numSum + (r*nthing*v(jay));
        denSum = denSum + r*nthing;
    end
  firstGuess(gLoop) = numSum/denSum;
end

%{
"At each data point (known), subtract from the reported value the value
obtained in the first-guess analysis at that point. The data point is
regarded as being in the center of the square formed by the 4 nearest grid
points, regardless of it's actual position within that area (use
coordinates). The interpolated value at the data point is determined by the
arithmatic average of the interpolated values at the 4 nearest grid points.
%}
 % If additional iterations are desired:
% v2 = zeros(M,1); %probably would over-write original values, but will keep
here for visualization purposes.
% guessAtKnown = zeros(M,1);
%
% for jay = 1:M %looping through known values
%     %1 Determine nearest coordinates (Find closest Grid Points)
%     xHigh = 0;
%     xLow = 0;
%     zHigh = 0;
%     zLow = 0;
%     xCoorHigh = 0;
%     xCoorLow = 0;
%     zCoorHigh = 0;
%     zCoorLow = 0;
```

```
%     xFlag = 0;
%     zFlag = 0;
%
%     for quickCheckX = 1:gridSizeX-1
%         if xFlag ==0
%             if(X(jay)>xpoints(quickCheckX) &&
X(jay)<xpoints(quickCheckX+1))
%                 xHigh = xpoints(quickCheckX+1);
%                 xLow = xpoints(quickCheckX);
%                 xCoorHigh = quickCheckX+1;
%                 xCoorLow = quickCheckX;
%                 xFlag = 1;
%             elseif (X(jay)==xpoints(quickCheckX))
%                 xHigh = xpoints(quickCheckX+1);
%                 xLow = xpoints(quickCheckX);
%                 xCoorHigh = quickCheckX+1;
%                 xCoorLow = quickCheckX;
%                 xFlag = 1;
%             elseif (quickCheckX==(gridSizeX-1) &&
X(jay)==xpoints(quickCheckX+1)) %could only ever happen on last iteration
%                 xHigh = xpoints(quickCheckX+1);
%                 xLow = xpoints(quickCheckX);
%                 xCoorHigh = quickCheckX+1;
%                 xCoorLow = quickCheckX;
%                 xFlag = 1;
%             end
%         end
%     end
%     for quickCheckZ = 1:gridSizeZ-1
%         if zFlag==0
%             if(Z(jay)>zpoints(quickCheckZ) &&
Z(jay)<zpoints(quickCheckZ+1))
%                 zHigh = zpoints(quickCheckZ+1);
%                 zLow = zpoints(quickCheckZ);
%                 zCoorHigh = quickCheckZ+1;
%                 zCoorLow = quickCheckZ;
%                 zFlag = 1;
% %                 fprintf('argument 1 = success, M=%d\n',jay);
% %                 fprintf('%d\n',quickCheckZ);
%             elseif (Z(jay)==zpoints(quickCheckZ))
%                 zHigh = zpoints(quickCheckZ+1);
%                 zLow = zpoints(quickCheckZ);
%                 zCoorHigh = quickCheckZ+1;
%                 zCoorLow = quickCheckZ;
%                 zFlag = 1;
% %                 fprintf('argument 2 = success\n');
%             elseif (quickCheckZ==(gridSizeZ-1) &&
Z(jay)==zpoints(quickCheckZ+1)) %could only ever happen on last iteration
%                 zHigh = zpoints(quickCheckZ+1);
```

```
%                 zLow = zpoints(quickCheckZ);
%                 zCoorHigh = quickCheckZ+1;
%                 zCoorLow = quickCheckZ;
%                 zFlag = 1;
% %                 fprintf('argument 3 = success\n');
%             end
%         end
%     end
%     %Coordinates identified!^
%
% %     fprintf('x1=%d x2=%d z1=%d
z2=%d\n',xCoorLow,xCoorHigh,zCoorLow,zCoorHigh);
%
%     value1 = zCoorLow*gridSizeX - mod(xCoorLow,gridSizeX);
%     value2 = value1 + 1;
%     value3 = zCoorHigh*gridSizeX - mod(xCoorLow,gridSizeX);
%     value4 = value3 + 1;
%
%     guessAtKnown(jay) =
(firstGuess(value1)+firstGuess(value2)+firstGuess(value3)+firstGuess(value4))
/4;
%     %^"First guess Analysis" from values from first iteration
% end
%
% % 2nd Round of Interpolation
%
% secondGuess = zeros(gridSize,1); %initial guess!
%
% %Defining new v ('v2')
% for loop = 1:M
%     if (isnan(guessAtKnown(loop)))
%         v2(loop) = v(loop)-guessAtKnown(loop);
%     else
%         v2(loop) = 0; %have no helpful information to add
%     end
% end
%
% for gLoop = 1:gridSize
%     numSum = 0;
%     denSum = 0;
%     for jay = 1:M %X and Z are same size
%         r = sqrt((pointSetX(gLoop)-X(jay))^2 + (pointSetZ(gLoop)-Z(jay)
)^2); %distance to that point
%         nthing = exp((-1*r^2)/4*k);
%         numSum = numSum + (r*nthing*v2(jay));
%         denSum = denSum + r*nthing;
%     end
%   secondGuess(gLoop) = numSum/denSum;
% end
```

```
%% Colors and Plotting

colors2 = zeros(gridSize,3);


% % COLORS FOR CALCULATED DATA - 1 run
% colorRange = max(firstGuess) - min(firstGuess); %e.g. 3 between high and
low
% for colLoop = 1:gridSize
%     colors2(colLoop, 1) = (firstGuess(colLoop)-min(firstGuess))/colorRange;
%     colors2(colLoop,2) = 0;
%     colors2(colLoop,3) = 1 - colors2(colLoop, 1);
% end

%Thresholded Colors for 1st Run
thresholds = linspace(min(firstGuess),max(firstGuess),10);
for colLoop = 1:gridSize

    if
(firstGuess(colLoop)>=thresholds(1)&&firstGuess(colLoop)<thresholds(2))
        colors2(colLoop,1) = 0.96;
        colors2(colLoop,2) = 0.87;
        colors2(colLoop,3) = 0.7;
    elseif
(firstGuess(colLoop)>=thresholds(2)&&firstGuess(colLoop)<thresholds(3))
        colors2(colLoop,1) = 0.13;
        colors2(colLoop,2) = 0.54;
        colors2(colLoop,3) = 0.13;
    elseif
(firstGuess(colLoop)>=thresholds(3)&&firstGuess(colLoop)<thresholds(4))
        colors2(colLoop,1) = 1;
        colors2(colLoop,2) = 1;
        colors2(colLoop,3) = 0;
    elseif
(firstGuess(colLoop)>=thresholds(4)&&firstGuess(colLoop)<thresholds(5))
        colors2(colLoop,1) = 0.53;
        colors2(colLoop,2) = 0.81;
        colors2(colLoop,3) = 0.98;
    elseif
(firstGuess(colLoop)>=thresholds(5)&&firstGuess(colLoop)<thresholds(6))
        colors2(colLoop,1) = 0;
        colors2(colLoop,2) = 0;
        colors2(colLoop,3) = 1;
    elseif
(firstGuess(colLoop)>=thresholds(6)&&firstGuess(colLoop)<thresholds(7))
```

```
        colors2(colLoop,1) = 0;
        colors2(colLoop,2) = 0;
        colors2(colLoop,3) = 0.5;
    elseif
(firstGuess(colLoop)>=thresholds(7)&&firstGuess(colLoop)<thresholds(8))
        colors2(colLoop,1) = 1;
        colors2(colLoop,2) = 0.65;
        colors2(colLoop,3) = 0;
    elseif
(firstGuess(colLoop)>=thresholds(8)&&firstGuess(colLoop)<thresholds(9))
        colors2(colLoop,1) = 1;
        colors2(colLoop,2) = 0;
        colors2(colLoop,3) = 0;
    elseif
(firstGuess(colLoop)>=thresholds(9)&&firstGuess(colLoop)<thresholds(10))
        colors2(colLoop,1) = 1;
        colors2(colLoop,2) = 0.41;
        colors2(colLoop,3) = 0.71;
    else %white is error
        colors2(colLoop,1) = 1;
        colors2(colLoop,2) = 1;
        colors2(colLoop,3) = 1;
    end
%     colors2(colLoop, 1) = (firstGuess(colLoop)-min(firstGuess))/colorRange;
%     colors2(colLoop,2) = 0;
%     colors2(colLoop,3) = 1 - colors2(colLoop, 1);
end

% colors3 = zeros(gridSize,3);
% % COLORS FOR CALCULATED DATA - 2 runs
% colorRange = max(secondGuess) - min(secondGuess); %e.g. 3 between high and
low
% for colLoop = 1:gridSize
%     colors3(colLoop, 1) = (secondGuess(colLoop)-
min(secondGuess))/colorRange;
%     colors3(colLoop,2) = 0;
%     colors3(colLoop,3) = 1 - colors3(colLoop, 1);
% end



% %COLORS FOR CALCULATED DATA - 2 runs LIDAR
% finalGuess = firstGuess + secondGuess;
% colors3 = zeros(gridSize,3);
% maxDist = max(finalGuess);
% % minDist = min(finalGuess);
% for colLoop = 1:gridSize
%     colors3(colLoop,1) = finalGuess(colLoop)/maxDist;
%     colors3(colLoop,2) = 0;
%     colors3(colLoop,3) = 1 - (finalGuess(colLoop)/maxDist);
```

```
% end

% scatter (X,Z, 10); %plot initial data

hold on;

% axis([-10 8 -4 5]);
% scatter (X,Z, 10, colors); %plot initial data
% scatter(pointSetX, pointSetZ, 10);


% scatter (X,Z, 10, colors); %plot initial data
%
scatter(pointSetX, pointSetZ, 10, colors2);

% scatter(pointSetX, pointSetZ, 10, colors3);

%% Write an Image File (you will have to rotate it)

%% BMP
% A = zeros(gridSizeX,gridSizeZ,3);
% gridcounter = 1;
% for loop = 1:gridSizeX
%
%   for loop2 = 1:gridSizeZ
%   A(loop, loop2, 1) = colors2(gridcounter,1);
%   A(loop, loop2, 2) = colors2(gridcounter,2);
%   A(loop, loop2, 3) = colors2(gridcounter,3);
%   gridcounter = gridcounter +1;
%
%   end
%
% end
% imwrite(A,'Test1.bmp');

%% JPEG
% A = zeros(gridSizeX,gridSizeZ,3);
% gridcounter = 1;
% for loop = 1:gridSizeX
%
%   for loop2 = 1:gridSizeZ
%   A(loop, loop2, 1) = colors2(gridcounter,1);
%   A(loop, loop2, 2) = colors2(gridcounter,2);
%   A(loop, loop2, 3) = colors2(gridcounter,3);
%   gridcounter = gridcounter +1;
%
%   end
%
```

```
% end
% imwrite(A,'Test1.jpeg');
```

SIFT keypoint detection algorithm:

```
#include <iostream>
#include <fstream>
#include <cstdint>
#include <stdio.h>
#include <iomanip>
#include <math.h>
// PCL
#include "pcl/point_types.h"
#include "pcl/point_cloud.h"
#include "pcl/io/pcd_io.h"
#include "pcl/keypoints/sift_keypoint.h"
#include <pcl/visualization/pcl_visualizer.h>
// OpenCV
#include "opencv2/core/core.hpp"
#include "opencv2/features2d/features2d.hpp"
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/nonfree/nonfree.hpp"

#define PI 3.14159265
#define COEFF 0.190631263 // Coefficient for converting LiDAR Cartesian
coordinates to meters

using namespace cv;

static void help()
{
	printf("\nThis program generates features for an image and a point cloud,
then matches them and returns the distance to the nearest object.\n"
		"Using the SIFT desriptor:\n"
		"\n"
		"Usage:\n img_lidar_matching <image.jpg> <pointcloud.pcd> \n");
}

void
detect_keypoints(pcl::PointCloud<pcl::PointXYZRGB>::Ptr &points,
	float min_scale, int nr_octaves, int nr_scales_per_octave, float
min_contrast,
	pcl::PointCloud<pcl::PointWithScale>::Ptr &keypoints_out)
{
	pcl::SIFTKeypoint<pcl::PointXYZRGB, pcl::PointWithScale> sift_detect;

	// Use a FLANN-based KdTree to perform neighborhood searches
	sift_detect.setSearchMethod(pcl::search::KdTree<pcl::PointXYZRGB>::Ptr(new
pcl::search::KdTree<pcl::PointXYZRGB>));

	// Set the detection parameters
	sift_detect.setScales(min_scale, nr_octaves, nr_scales_per_octave);
	sift_detect.setMinimumContrast(min_contrast);

	// Set the input
	sift_detect.setInputCloud(points);

	// Detect the keypoints and store them in "keypoints_out"
	sift_detect.compute(*keypoints_out);
}
```

```
void visualize_keypoints(const pcl::PointCloud<pcl::PointXYZRGB>::Ptr points,
	const pcl::PointCloud<pcl::PointWithScale>::Ptr keypoints)
{
	// Add the points to the vizualizer
	pcl::visualization::PCLVisualizer viz;
	viz.setBackgroundColor(256, 256, 256);
	pcl::visualization::PointCloudColorHandlerRGBField<pcl::PointXYZRGB>
rgb(points);
	viz.addPointCloud(points, "points");
	viz.setPointCloudRenderingProperties(pcl::visualization::PCL_VISUALIZER_POI
NT_SIZE, 3, "points");

	// Draw each keypoint as a sphere
	for (size_t i = 0; i < keypoints->size(); ++i)
	{
		// Get the point data
		const pcl::PointWithScale & p = keypoints->points[i];

		// Pick the radius of the sphere *
		float r = 2 * p.scale;
		// * Note: the scale is given as the standard deviation of a Gaussian
blur, so a
		//   radius of 2*p.scale is a good illustration of the extent of the
keypoint

		// Generate a unique string for each sphere
		std::stringstream ss("keypoint");
		ss << i;

		// Add a black sphere at the keypoint
		viz.addSphere(p, 2 * p.scale, 0.0, 0.0, 1.0, ss.str());
	}

	// Give control over to the visualizer
	viz.spin();
}

double dist_calc(double x, double y, double z)
{
	double distance = sqrt((x*x) + (y*y) + (z*z));
	return(distance);
}
double azimuth_calc(double z, double distance)
{
	double azimuth = asin(z/ distance) * 180 / PI;
	return(azimuth);
}
double elevation_calc(double y, double x)
{
	double elevation = atan2(y, x) * 180 / PI;
	return(elevation);
}

int keypoints(const char * filename1, const char * filename2)
{
	// Create some new point cloud to hold data
	pcl::PointCloud<pcl::PointXYZRGB>::Ptr points(new
pcl::PointCloud<pcl::PointXYZRGB>);
```

```
	// Original
	//pcl::PointCloud<pcl::PointWithScale>::Ptr keypoints_ptr(new
pcl::PointCloud<pcl::PointWithScale>);

	// Create keypoint object
	pcl::PointCloud<pcl::PointWithScale> keypoints;
	pcl::PointCloud<pcl::PointWithScale>::Ptr keypoints_ptr(&keypoints);

	// Load a point cloud
	pcl::io::loadPCDFile(filename2, *points);

	// Compute keypoints
	const float min_scale = 0.2; // Tune to dataset of interest, originally
0.01
	const int nr_octaves = 2; // Tune to dataset of interest, originally 3
	const int nr_octaves_per_scale = 2; // Tune to dataset of interest,
originally, 3
	const float min_contrast = 1; // Tune to dataset of interest, originally
10.0
	detect_keypoints(points, min_scale, nr_octaves, nr_octaves_per_scale,
min_contrast, keypoints_ptr);


	//////////////////////////////////////////////////////////////
	// load image
	Mat img1 = imread(filename1, CV_LOAD_IMAGE_COLOR);
	if (img1.empty())
	{
		printf("Cannot read image\n");
		return -1;
	}

	int nfeatures = 400;
	int nOctaveLayers = 3;
	double contrastThreshold = 0.05;
	double edgeThreshold = 10;
	double sigma = 2;

	Ptr<FeatureDetector> detector = new SIFT(nfeatures, nOctaveLayers,
contrastThreshold, edgeThreshold, sigma);
	Ptr<DescriptorExtractor> extractor = new SIFT(nfeatures, nOctaveLayers,
contrastThreshold, edgeThreshold, sigma);

	vector<KeyPoint> img_keypoints;
	detector->detect(img1, img_keypoints);

	Mat descriptors1;
	extractor->compute(img1, img_keypoints, descriptors1);

	// Search for nearest object keypoint distance and angle
	float minDist = 1000;
	float distance = 0;
	int closest = 0;
	float x = 0;
	float y = 0;
	float z = 0;
	float azimuth = 0;
	float elevation = 0;
	for (int m = 0; m < keypoints.width; m++)
	{
```

```
            x = COEFF * keypoints.points[m].data[0];
            y = COEFF * keypoints.points[m].data[1];
            z = COEFF * keypoints.points[m].data[2];
            distance = dist_calc(x, y, z);
            azimuth = azimuth_calc(z, distance);
            elevation = elevation_calc(y, x);
            if (distance < minDist)
            {
                  closest = m;
                  minDist = distance;
            }
      }

      // Determine closest keypoint in meters and calculate azimuth
      x = COEFF * keypoints.points[closest].data[0];
      y = COEFF * keypoints.points[closest].data[1];
      z = COEFF * keypoints.points[closest].data[2];
      distance = dist_calc(x, y, z);
      azimuth = azimuth_calc(z, distance);
      elevation = elevation_calc(y, x);

      cout << "\n\nThe nearest object is: ";
      cout << distance;
      cout << " meters away\n with an azimuth of ";
      cout << azimuth;

      // Print number of Lidar keypoints to command prompt
      cout << "\n\nThere are ";
      cout << keypoints.width;
      cout << " lidar keypoints detected.";

      cout << "\n\nThere are ";
      cout << nfeatures;
      cout << " image keypoints detected.";

      // drawing the results
      namedWindow("keypoints", WINDOW_NORMAL);
      Mat disp_img_keypoints;
      drawKeypoints(img1, img_keypoints, disp_img_keypoints, Scalar::all(-1),
DrawMatchesFlags::DRAW_RICH_KEYPOINTS);
      imshow("keypoints", disp_img_keypoints);

      // Display Pointcloud and Keypoints
      visualize_keypoints(points, keypoints_ptr);

      waitKey(0);
      return (0);
}

////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////

int main(int argc, char** argv)
{
      if (argc != 3)
      {
            help();
            return -1;
      }
      keypoints(argv[1],argv[2]);
```

```
        return 0;
}
```

Appendix J: Recommendations for Future Work

In the data acquisition stage, future work may include the implementation of multi-threading on the embedded device. Multi-threading will improve the processing speed and improve data acquisition to allow complete system integration.

In the data processing stage, future work should include registration of the point cloud and image data, which will improve the precision of the 3D distance overlay for the 2D image. Implementing registration will allow each keypoint to have a corresponding interpolated distance, and will eliminate any distortion created from the compression of 3D to 2D.