UM::Autonomy Wolvemarine, A Highly Capable Autonomous Surface Vehicle

Eric Rossetti, Andrew Beck, Devin Witt,
David Devecsery, Ryan Wolcott, Anthony Bonkoski, Alexander Prog

University of Michigan
Ann Arbor, MI 48109

Figure 1: The completed Wolvemarine in the water.

ABSTRACT

Wolvemarine is a fully autonomous surface vehicle with a pontoon hull design. This year the team has put considerable effort into designing a robust, powerful three-dimensional hybrid camera-LIDAR vision system, capable of reliable detection of a wide array of obstacles impossible to detect with other systems. The system has been designed, implemented, and tested with the goal of competing in the Fourth Annual AUVSI Autonomous Surface Vehicle Competition, where it will demonstrate its ability by attempting the “Four Elements" challenge presented in this year’s competition.

1. INTRODUCTION

Wolvemarine is the UM::Autonomy submission to the Fourth Annual AUVSI Autonomous Surface Vehicle Competition. It was designed and created with the goal of completing all of the events in the “Four Elements” themed competition. As Wolvemarine is a fully-autonomous vehicle and the system took over two years, when considering the design time placed into Mjolnir, Wolvemarine’s predecessor, we cannot possibly discuss every aspect of the system in this paper. Thus we will spend the remaining space highlighting the differences between this year’s entry and last year’s entry.

Last year we discussed the importance of the design, build, test methodology in constructing Mjolnir. We have continued with this approach and built Wolvemarine following the same, successful strategy. After analyzing Mjolnir’s successes and weaknesses at the competition, we developed a new iteration of our ongoing project, Wolvemarine. While Wolvemarine may appear similar to Mjolnir on the surface, it is actually a far more powerful and capable system with major changes to many core components, such as the vision system and hull fabrication.

Wolvemarine’s redesign began from the ground up, with the hulls. We replaced our previous hulls with new, custom built, hulls providing better control in-water. We also completely overhauled our robot vision system with a 3D LIDAR-camera hybrid system. This system is by far the largest and most complex undertaking we have accomplished this year. Finally, Wolvemarine now includes a simulataneous localization and mapping system (SLAM), further enhancing the localization and mapping capabilities provided by our 3D vision system. Throughout the remainder of this paper we will discuss these, as well as several other more subtle modifications and improvements in much more detail, then discuss the new, powerful debugging and control systems we have added to make Wolvemarine by far our most promising submission to this competition yet.

2. HULL

2.1 Design

Wolvemarine’s hull is a redesign of Mjolnir’s. That design was based off a four hulled small water area twin hull (SWATH) which we modified to become a twin hull. This year we made modifications to the hull’s shape and bow to allow for better performance on the water. We designed the hulls to be more streamlined to help with tracking and reduce the water flow separation from the hull. Also, we redesigned the bow of the hulls to resemble the bow of a pontoon vessel. This design allows for the hulls to be semi-planing and allow the autonomous surface vehicle to travel at much faster

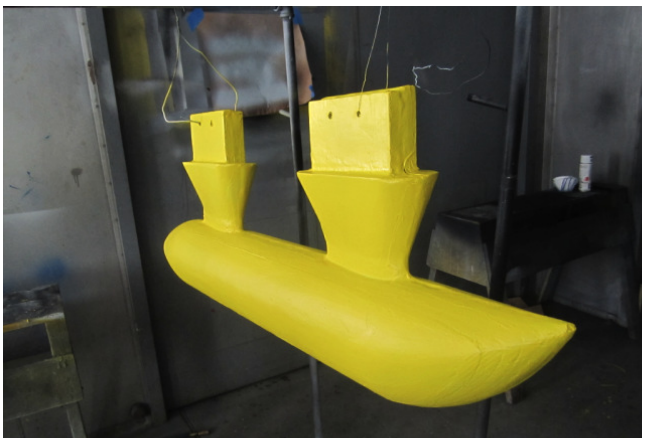

Figure 2: One of the two hulls which shows the streamline of the hulls and the new semi-planing bow design.

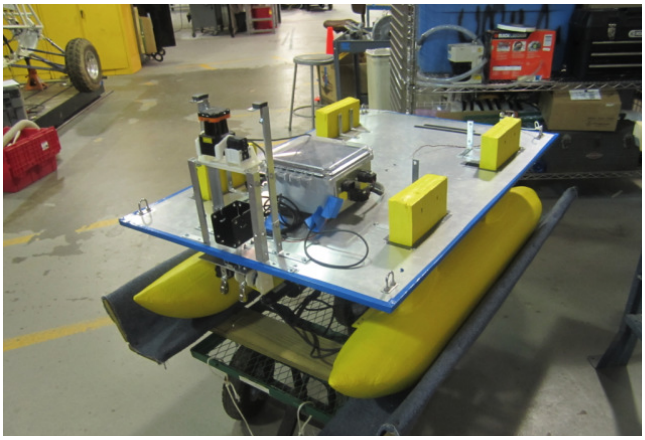

Figure 3: The hulls and deck attached to one another. The deck is currently missing the main computer box, battery, and the cameras.

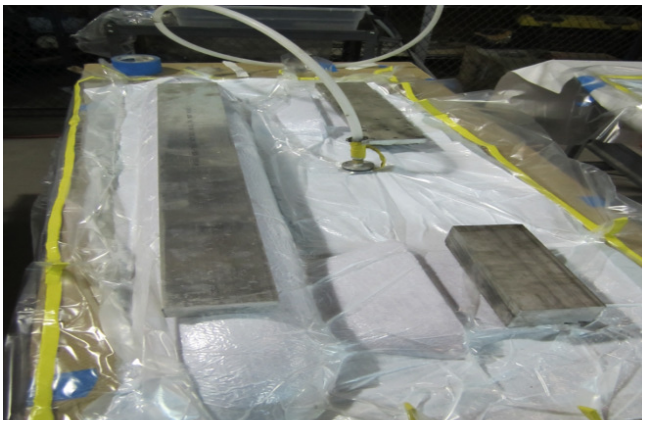

Figure 4: The vacuum bagging process involved vacuum bagging only half of a hull to get a better finish. The metal bars lying on the hulls were used the keep the hulls as flat as possible during the resin curing process.

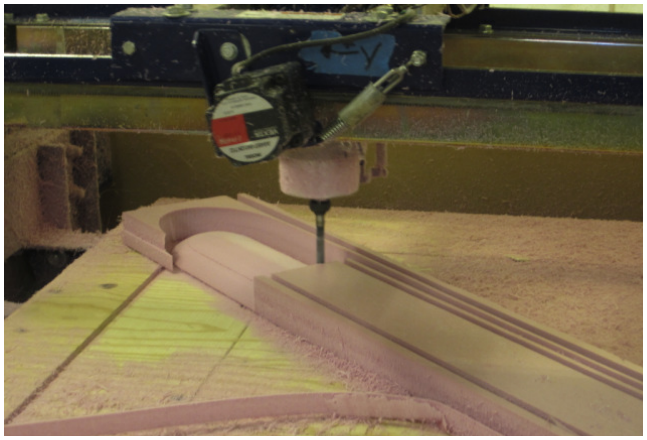

Figure 5: The 3-axis CNC router cutting out a piece of the hull. There was a total of 16 pieces to be cut by the router and glued together to make the final products.

speeds safely. The redesigned hulls are very stable in pitch, roll, and yaw and thus provide a stable platform for the vehicle's electronic systems. The deck of the vessel has been kept similar to the previous design. The deck has been modified in dimensions (32" x 46") to allow for easy transfer through doorways. The deck design incorporates a modular theory. Every component is designed to be removed from the deck with ease by utilizing quick release pins for both the electrical box and the hulls.

2.2 Fabrication

The hull was fabricated using a male mold fiberglassing process. High-density foam molds were fabricated on our 3-axis CNC machine by inputting our 3D model into Catia and creating machining code to accurately cut our molds. The molds were then sanded and pieces glued together to create the desired product. Then two layers of fiberglass were laid up onto the male molds, and vacuum bagged to harden smoothly. Once the fiberglass had set, it was sanded and the hull halves were glued together and finished with a fiberglass seam. Then an epoxy layer was coated on the hulls to further waterproof them.

3. ELECTRICAL BOX

While our previous year's electrical box provided a firm foundation for Wolvemarine it had several fundamental flaws which we have addressed in this year's submission. Many of the components of last year's box were modularly constructed and well designed, such as the motor power system, our peripheral power hub, our wire routing scheme, and the two-PC system. Consequently we either left these systems alone or only provided minimal modifications. The systems we did have to change were our external interfacing ports, in particular to our cameras and laser scanner, and our internal network.

As we added the new components required to fully power

our vision system we discovered that our previous 80W peripheral power regulation system was inadequate, often resulting in voltage brownouts, even when only some peripherals were attached. To compensate for this we have replaced it with a 250W automotive regulator, providing more then enough power for all of our non-PC components.

We have additionally removed our communication Wafer PC. Mjolnir used a very small AMD Geode Wafer for serial port connections, and device interfacing. The Wafer presented a fundamental issue to our internal network. It had a 100MBit network connection, while each PC and the router operated at 1GBit speeds, and consequently the Wafer was a bottleneck in our inter-PC communication within our electrical box, which became relevant when pushing large amounts of 3D LIDAR data across the network. To amend this we have removed the Wafer and introduced a 4 port RS232 to USB hub which is connected directly to one of our PCs.

The final significant change we made to our box was to replace our previously hacked together E-stop, water gun, LED signal beacon, and actuator perf boards with our new PCBs. These PCBs provide a firm foundation for the remainder of the box and remove a great amount of issues with shorting and breaking wires. With these modifications Wolvemarine has improved and stabilized its electrical box, providing a solid foundation for our peripherals and software system.

4. VISION PIPELINE

One of the most difficult challenges in this year's competition is to identify relevant obstacles in the world around the vehicle. To accomplish this task we have generated a powerful, three-dimensional vision system capable of identifying a diverse range of generic features around our vehicle and apply them to task-specific challenges. The system incorporates a Hokuyo UTM-30LX LIDAR, Dyanamixel AX12 servo to control the LIDAR angle, and two Point Grey FireFly MV CMOS cameras for point cloud coloring. To accomplish generic vision for our boat we divide the task into four primary steps: colored point cloud generation, point cloud segmentation, geometric feature detection, and task-specific object recognition.

4.1 Calibration

A key step in developing our 3D vision system is co-registration of camera and LIDAR points. To accomplish co-registration, we developed a calibration system to accurately determine the intrinsic parameters of the camera and the rigid body transformation from the LIDAR frame of reference to the camera frame of reference. The system involves collecting multiple image and point cloud samples of a checkerboard pattern. The checkerboard in the image is detected and analyzed using OpenCV functions, which produce camera intrinsics and position and orientation of the checkerboard in the camera reference frame. The checkerboard in the point cloud is detected and analyzed using a capture program that we developed, where the point cloud is projected onto a 2D space and colored by depth. A user then selects the checkerboard by clicking on the four corners of the plane. Once a region has been selected, RANSAC is applied to determine the inliers of the plane in the selected region. Once the image and point cloud checkerboards have been detected and analyzed, the rigid body transformation is calculated. This involves computing an initial estimate using the closed form equations described in [9] then refining this estimate by solving the non-linear optimization problem described in [5] with an implementation of the Levenberg Marquardt algorithm.

4.2 Point Cloud Generation

Wolvemarine's vision system is primarily centered around our LIDAR point cloud. We use a servo mount to vary the angle of the Hokuyo LIDAR to collect a three-dimensional point cloud of distances of objects in front of the boat. The point cloud code depends on the state of the boat, the LIDAR data, and the position of the servo.

We define a "slice" as a single message of LIDAR data published to LCM by the Hokuyo driver. The point cloud creates a slice using the data from the LIDAR, which includes a timestamp, an initial angle, an angular step, the number of angles, and an array of distances. This essentially creates an array of points characterized by a distance r and an angle ϕ from the front of the boat.

The servo driver publishes a timestamp and the servo angle. This publication occurs too infrequently to receive an accurate angle multiple times within a given slice. However, a constant angular velocity model of the servo can serve as a good approximation, allowing an estimate of the servo angle θ at the time of the slice to be determined using the previous and following servo angles and timestamps.

Using the distance and angles, we can determine the rectangular coordinates of a LIDAR point in space:

$$x = r * cos(\phi)cos(\theta)$$
$$y = r * sin(\phi)$$
$$z = r * cos(\phi)sin(\theta)$$

The state of the boat is required because the slices are created at a given boat state, but the boat changes state between slices. To compensate for this, we store the state of the boat at the time of the slice and at the time of creating the point cloud to create differential state consisting of six degrees of freedom:

$\Delta\alpha$=change in roll
$\Delta\beta$=change in pitch
$\Delta\gamma$=change in yaw
Δx=change in x
Δy=change in y
Δz=change in z

We then update each slice by applying the 3D rigid body transformation as found in Figure [8].

Finally, we publish arrays of points for x', y', and z'.

Figure 6: The overlay of our point cloud plane segment into the checkerboard image. Please note the segment is of the plane containing the image, not the image itself.

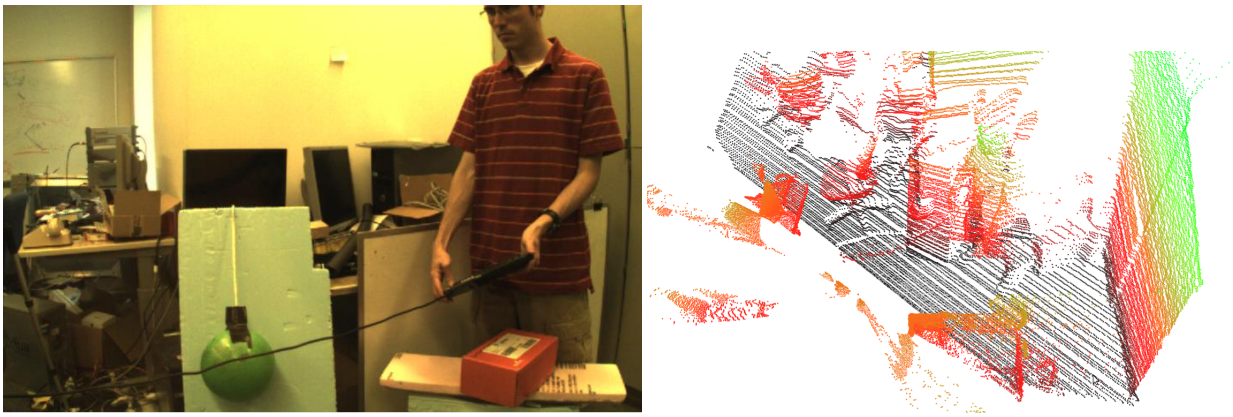

Figure 7: A visualization of a fully formed uncolored point cloud

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} cos(\Delta\gamma)cos(\Delta\beta) & cos(\Delta\gamma)sin(\Delta\beta)sin(\Delta\alpha) - sin(\Delta\gamma)cos(\Delta\alpha) & cos(\Delta\gamma)sin(\Delta\beta)cos(\Delta\alpha) - sin(\Delta\gamma)sin(\Delta\alpha) & \Delta x \\ sin(\Delta\gamma)cos(\Delta\beta) & sin(\Delta\gamma)sin(\Delta\beta)sin(\Delta\alpha) + cos(\Delta\gamma)sin(\Delta\alpha) & sin(\Delta\gamma)sin(\Delta\beta)cos(\Delta\alpha) - cos(\Delta\gamma)sin(\Delta\alpha) & \Delta y \\ -sin(\Delta\beta) & cos(\Delta\beta)sin(\Delta\alpha) & cos(\Delta\beta)cos(\Delta\alpha) & \Delta z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Figure 8: Transformation between states

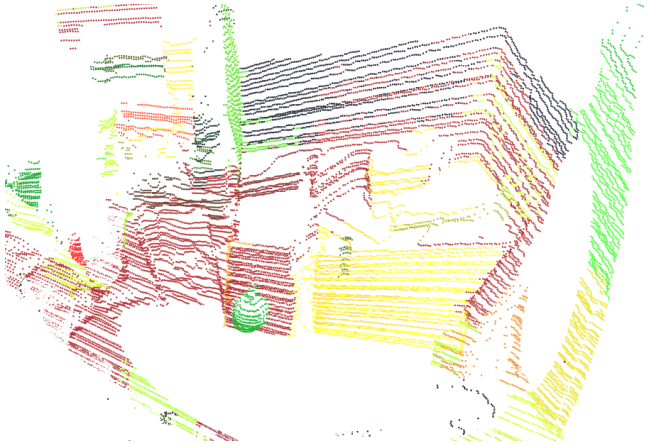

Figure 10: The colored point cloud after segmentation. Please note how the ball, plane behind the ball, and plane next to it are all segmented separately for fast feature detection.

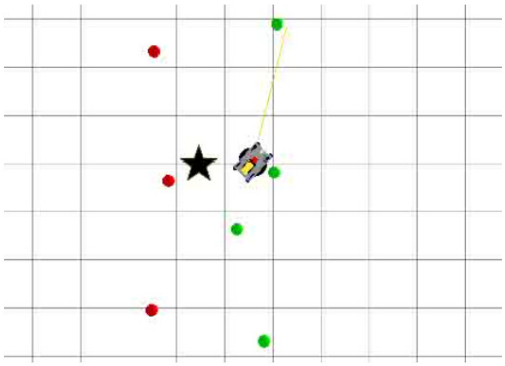

Figure 11: Resulting feature-based EKF-SLAM map. Star is GPS reference

4.3 Segmentation

While the motivation for the point cloud and its generation are fairly straight forward, the process of abstracting usable features to be fed into SLAM or our route planning algorithm is a much less intuitive task. The segmentation step provides a foundation for the remainder of the vision system by splitting the massive point cloud structure into a series of smaller "segments", each of which ideally represent a distinct object or feature.

Our segmentation algorithm was designed with two primary goals in mind. First we need to properly segment simple geometric shapes generated through the point cloud to identify candidates for our geometric feature detector. Second we must be aware of color changes when segmenting to identify shapes or pictures on objects, such as in the "air" and "fire" portions of this year's task. To accomplish this we have adapted a hybrid LIDAR-camera segmentation algorithm which operates on a colored point cloud. We chose to implement the algorithm found in [7]. This algorithm is a variation of the highly cited [2] paper, taking normals of the point cloud into effect along with image color data.

Our segmentation algorithm is graph-based, with each point of the point cloud composing a vertex of the graph. Edges are created between all adjacent points, each edge given two weights, one of the edge as the RGB color difference ($wrgb$) between the two points, $sqrt(\Delta R^2 + \Delta G^2 + \Delta B^2)$ and the other the difference between the surface normals of the two points ($wnorm$). After the edge is calculated for each point within the graph, the edges are sorted by wrgb, approximately ordering the graph by edge weights.

After the edges are computed, the heart of the algorithm begins. Each vertex of the graph is initially assigned to its own segment using a union-find data-structure, then each edge is tested to see if it meets the threshold for joining. Two thresholds for each segment are checked to determine if the segments on an edge may be joined. Those thresholds are knorm and krgb, which are initially specified as parameters to the algorithm. After an edge is joined the segment calculates a new threshold as: $knorm = wnorm + knorm_{init}/(N)$ where N is the number of points in the segment. This process is done for each edge in the graph. Then any segment which is deemed too small is combined with its closest matching segment, completing the segmentation process.

4.4 Feature Detection

The ASV competition necessitates the ability to accurately identify objects in the world such as buoys, signs, ramps, and docks. All of these objects can be described through a combination of general shapes and their context in the world. Our object detection system focuses on the identification of planes, spheres, and cylinders. Once the point cloud has been segmented, the object detection module analyzes each segment to determine if it matches a plane, sphere, or cylinder using the methods described in [6]. The detection must then meet certain size constraints and have a great enough consensus in order to be passed on to SLAM. SLAM is able to map these general objects and localize off of them, producing a reliable map of objects as its output. Looking at these objects in the context of the map allows us to classify each general shape as a mission-specific object (buoy, sign, ramp, etc.). Delaying this classification makes for a more generic system and provides extensibility for future additions.

5. SLAM

Simultaneous localization and mapping is an excruciat-

Figure 9: Colored Point cloud before segmentation and original camera image. Please note the spherical ball hanging in front of a plane, and the person standing next to the ball are clearly visible in the point cloud.

ingly difficult task for robotic platforms. However, the selection of different SLAM algorithms allows the problem to become tractable enough to generate maps that are useful for platform navigation. To handle Wolvemarine's localization and mapping, we implemented a feature-based EKF-SLAM algorithm. Surface vehicles provide an ideal scenario for using a feature-based SLAM algorithm because of the sparsity of obstacles on the water plane. These obstacles on the water plane can be reliably, repeatably detected, which is essential for the success of the mapping algorithm. As inputs into our SLAM algorithm, Wolvemarine uses GPS data, compass data, and feature data, which is provided from the point cloud feature extractor.

For our application, we do not have any available control inputs, so we will be limited to a constant velocity, constant angular velocity motion model [1]. Because our platform's environment can be reduced to a planar environment, we can confine our state estimates to 3-DOF that we must augment with planar velocity and angular velocity about the robot's z-axis. The resulting 5-DOF robot's state vector is:

$$x_r = \begin{bmatrix} x \\ y \\ \theta \\ v \\ \omega \end{bmatrix}$$

At each time-step, the velocity and angular velocity is modeled having an additive zero-mean Gaussian white noise to correct for the deficiencies in the constant velocity framework. Compass and GPS observations, both of which are 10 Hz, provide a linear update to the Kalman filter. The combination of these two sensors alone provides a fairly accurate localization and allows for suitable vehicle navigation.

The addition of feature observations (buoys, signs, gate buoys, etc.) turns the problem into a true SLAM problem and allows for even more accurate localization and a high-fidelity map for path planning [8]. By constraining our features to the 2D world, all feature observations are restricted to a relative distance and bearing from the vessel. These observations provide nonlinear updates into the Kalman filter, therefore, we linearize via the first-order Taylor series approximation. This method is known as the extended Kalman filter. Furthermore, on each iteration, data association between observations and the current map are done using Joint Compatibility Branch and Bound, which provides a robust method for associating features in complex environments [3].

In addition to generating a feature-based map, Wolvemarine also uses the raw point cloud data generated from the 2D laser scanner to create an occupancy grid map. This grid map feeds into the path planner and provides awareness for obstacle avoidance. While the feature-based map provides information on where the vessel should navigate next, the occupancy grid map tells us how to get there safely.

6. ROUTE PLANNING

After generating the point cloud, color tagging it, segmenting, detecting geometric features, and SLAMming it all together we have created a solid foundation for the high level system. We will now discuss the design of this high level decision engine, our Route Planner. The route planner has been modularized and decentralized into many different components and different stages. At the highest level there is a manager which determines which portion of the competition we are in, and allows that handling algorithm to run. Each handling algorithm is then coded in its own process and decides if it is capable of running. It communicates this information to the route planner and then begins controlling the boat when granted permission. This allows us to easily recycle code and inherently handles automatic retry of failed tasks.

6.1 Route Planner Manager

Inspired from shortcomings in last year's route planner we designed a Route Planner Manager to simplify and modularize the route planner.

There are numerous tasks that must be completed in any given run (speed gates, buoy channel, etc.) Each of these tasks are very different in nature, and thus a separate handler is required for each. These tasks must be organized such that we work on only one task at a time. The straight-forward approach would be to write a top-level route planner that is akin to a Finite-state machine (i.e. Do each task in a hardwired order). However, an issue will arise if the boat gets "stuck" on a task.

To remedy this and other issues, we designed a Route Planner Manager. Each task works independently from the others, and the manager will decide which task is allowed to run. Thus, each task will continuously publish its status to the manager. The manager will listen to and save this status information, using it to decide which should be handled. The status consists of information such as task priority and runability.

Overall, this system will make the route planner much more robust. By correctly setting the priorities of various tasks, we can let the boat decide the ordering of the tasks at run-time. We no longer have to be confined to a predetermined route-plan. Also, if the boat fails a task we can simply change the priority. The boat can then work on a different task and reattempt the failed task later.

6.2 Individual Tasks

With our firm underlying foundation, the challenge of navigating the tasks becomes far less daunting, and many are reduced to trivial code. At the high level all each task has to do is monitor our SLAM world, containing cylinder, plane, and sphere objects, as well as an occupancy grid, see if they have the information needed to run, ask permission from the route planner, then publish their goal destinations to our boat control system. Here is a breakdown of each task:

6.2.1 Speed Gates

The speed gate is trivial for our system, we look for cylindrical objects, identify the red and green buoy and publish a way-point between each.

6.2.2 Buoy Channel

The buoy channel is slightly more involved. We find each sphere in the water plane, then identify its color, and tag it as a buoy. We then match red and green buoys based on distance, direction and angle criteria to determine the next nearest buoy pair to navigate through, we then set a way-point between that pair. When there are no more valid pairs we stop navigating the channel. It is important to note here that we do not avoid obstacles at this stage. That is handled later by our obstacle avoidance algorithm.

6.2.3 Shore Following

Given our underlying system, shore following becomes much simpler. Shore is a long connected area of nearly vertical planes with a large horizontal plane on top. After identifying the shore we can easily choose to follow it until a task specific algorithm receives control.

6.2.4 Air Challenge

The air challenge is very manageable with our system. Because of our segmentation algorithm, each sign, with the exception of air, will be segmented into two different plane segments. We may find the vertical planes, then find the color of the smaller segment, this will tell us which sign is earth, water, or fire. A vertical white segment without a color is air. We have also calibrated an FLIR PathFinderIR IR Camera into our system, in a manner similar to our other cameras, allowing us to determine the relative temperature of each sign. After this data is gathered it is trivial to transmit it over the network.

6.2.5 Water Challenge

The water challenge is also easy to accomplish. We search the shore for a red segment hanging out over the water, identify it as the button, turn off obstacle avoidance, and navigate to contact it.

6.2.6 Fire Challenge

The fire challenge is nearly a subset of the air functionality. We find the plane of the boat floating in the water, find the red hole segment, then aim our boat, and consequently water gun, towards the hole, where we fire until the red flag appears.

6.2.7 Earth Challenge

The earth challenge is by far the most difficult for us. Finding the ball is actually rather easy, it is a pink sphere near the EZ dock, a plane running into the water, but getting it is another matter. Because of the size and construction of our boat it is impossible to ascend the ramp and retrieve the ball. We have chosen to deploy a small amphibious RC car to grab the ball for us. This presents several issues, we first must navigate the car to the ball, and then retrieve the car from the land. We have attached a safety line that we will retrieve the car with after it has completed the task. The most difficult part is getting the car to the ball. Our vision system requires a static environment, meaning that we cannot track the car in real-time, so we use the April Tag System [4] to track the car's 6DOF state. We then only need to send wireless RC signals to the car, and navigate it to the ball using offshoots of our underlying boat navigation systems.

6.3 Obstacle Avoidance

We have implemented an obstacle-avoiding algorithm to help us dodge the shore and other objects in our way such as the yellow buoys in the channel. Our obstacle avoidance algorithm takes advantage of the SLAM occupancy grid, and uses an A* algorithm to find the optimal path.

Our implementation of the obstacle avoidance is fairly simple, as we do not have many obstacles to avoid. We begin with our occupancy grid, our destination from the running task, and our state. We then blur the occupancy grid to compensate for the size of our boat, and run a typical A* algorithm, using the Manhattan method for the predicted distance to our goal. After we have created a route we determine the fewest number of destination points we can include without coming near an obstacle, and broadcast these points to our motor control system.

7. PROCESS MANAGER

Our software system is designed around a networked message-passing framework (LCM framework). As a result, the complete system is composed of numerous processes spread across two computers. Therefore, traditional process management quickly becomes unwieldy. To this end, we developed a specialized process manager.

Due to the design, our process manager can be used both for test/debug sessions and competition runs. The process manager is invoked with a configuration file as an argument. The configuration file names all the processes that should be run. Because the configuration is not hard-coded, we can use a different configuration when testing as opposed to a trial run. In addition, the process manager sports an interactive command-line for managing the processes. This allows the developer to start and stop a process or view the process's output without searching for the terminal it was started from. In addition, all process output is stored in a log file for later reference. These features make the process manager a phenomenal debug tool.

The process manager also attempts to ensure that processes are running and working as intended. For this, the process manager will restart a process if it dies. Also, the process manager listens to the LCM channels of the managed processes. Using the frequency of publishes the process manager can determine if a process has become stuck even if it has not terminated. These features aim to improve robustness by preventing a total system failure due to a minor bug.

8. SIMULATOR

Due to weather restrictions within Michigan, we are unable to test our robotic platform for much of the time that our team members are available. To mitigate this inability to test new algorithms, we created a full robotic simulator. Within this simulator, we can build up a mock course that includes the gate buoys, a buoy channel, shoreline, and several other features that we could expect in the environment. This simulator is able to completely spoof all sensor readings, sampling the ground-truth from a Gaussian distribution, and responds to vehicle control inputs to drive the simulator. Therefore, the simulator is able to perfectly mirror the actual environment, which allows us to rapidly test new ideas, algorithms, and strategies. A screen shot of the simulator can be seen in Figure [12]

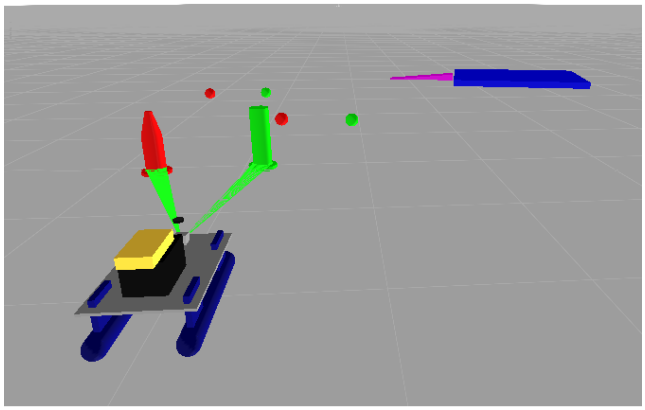

Figure 12: A screenshot of the simulator visualization

9. CONCLUSION

We have presented Wolvemarine, our submission to the fourth AUVSI Autonomous Surface Vehicle Competition. Wolvemarine took the Design, Build, Test methodology and redesigned Mjolnir. We have improved our hulls and deck design, lightening our boat and increasing our control. We then implemented a new vision, SLAM, and route planning methodology which should be powerful and robust enough for years of competitions to come.

10. REFERENCES

[1] A. Davison, I. Reid, N. Molton, and O. Stasse. Monoslam: Real-time single camera slam. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 29(6):1052 –1067, june 2007.

[2] P. F. Felzenszwalb and D. P. Huttenlocher. Efficient graph-based image segmentation. pages 167–181, 2004.

[3] J. Neira and J. Tardos. Data association in stochastic mapping using the joint compatibility test. *Robotics and Automation, IEEE Transactions on*, 17(6):890 –897, dec 2001.

[4] E. Olson. Apriltag: A robust and flexible multi-purpose fiducial system. Technical report, University of Michigan APRIL Laboratory, May 2010.

[5] G. Pandey, J. McBride, S. Savarese, and R. Eustice. Extrinsic calibration of a 3d laser scanner and an omnidirectional camera. In *7th IFAC Symposium on Intelligent Autonomous Vehicles*, volume 7, Leece, Italy, 2010.

[6] R. Schnabel, R. Wahl, and R. Klein. Efficient ransac for point-cloud shape detection. *Computer Graphics Forum*, 26(2):214–226, June 2007.

[7] J. Strom, A. Richardson, and E. Olson. Graph-based segmentation for colored 3D laser point clouds. In

Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), October 2010.

[8] S. Thrun, W. Burgard, and D. Fox. *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents)*. The MIT Press, 2005.

[9] R. Unnikrishnan and M. Hebert. Fast extrinsic calibration of a laser rangefinder to a camera. Technical report.