Husky Rose

An Entry from ***Rose-Hulman Institute of Technology*** in the

2012 Intelligent Ground Vehicle Competition

(IGVC 2012)

Tayler Burns, Kyle Green, Vismay Modi, Zhihao Ni,

Cameron Spry and Jiaren Wu

Faculty Advisor:

David Mutchler, Professor of Computer Science and Software Engineering

Faculty statement: I hereby certify that the design and development of the robot discussed in this technical report has involved significant contributions by the aforementioned team members, consistent with the effort required in a Senior Design course.

____

David Mutchler, Professor of Computer Science and Software Engineering

Table of Contents

Table of Contents 2
1 Team Overview 3
2 Design Process 3
3 Hardware 4
3.1 The Husky Robot from Clearpath Robotics 4
3.2 Hardware We Added to the Basic Husky Robot 4
3.3 Hardware Innovation 5
3.4 Costs of Equipment 5
4 Software 5
4.1 Platform – Robot Operating System (ROS) 5
4.2 Vision 7
4.2.1 Vision for Lane Following (staying within the white lines) 7
4.2.2 Vision for detecting barrels 9
4.2.3 Vision for Obstacle Detection 10
4.3 LIDAR 12
4.4 GPS 12
4.5 Navigation 12
4.6 Software Innovation 14
5 Performance analysis / testing 15
6 Conclusion 15
7 References 16

Note: The outline of this report was taken from the Princeton entry in the 2008 IGVC [1]. We are grateful for their example of an award-winning design report.

1 Team Overview

The Husky Rose team is organized via a new course offered this year at Rose-Hulman: CSSE 290 *Software Challenges in Autonomous Vehicle Navigation*. Twenty-five students took the course over 3 terms, for a total of student 50 credit-hours over the year. Six students are in the course for the spring term and authored this report.

This is our first year in IGVC. We are still implementing our ideas. We will update this design report at the oral presentation at the IGVC contest.

In this problem-based course, students design and develop software to solve challenges in navigation faced by autonomous vehicles (robots). The problems are real problems faced by real robots in a real competition. The software to be developed is for challenges faced by a robot that will be entered in the 2012 Ground Vehicles Competition (IGVC), although the software will be designed to appl Intelligent y to other robots as well. Challenges include location, vision, planning and control:

- Where am I, and where will I be in X seconds?
- What obstacles are ahead of me?
- What sensors should I use to learn the above? How do I deal with uncertain and conflicting information?
- How will I avoid the obstacles ahead of me? Where do I want to go next?
- At each step, what speed should I be driving, to accomplish my overall goal of winning the competition? What sub-goals should I set to accomplish that overall goal?
- What commands do I give to motors to accomplish what I want to do next?
- What information do I need from sensors to accomplish what I want to do next?

Figure 1. A portion of the course description of CSSE 290 *Software Challenges in Autonomous Vehicle Navigation*, Rose-Hulman Institute of Technology.

2 Design Process

In the fall term, we were starting from scratch. We knew some of the challenges and knew that we wanted to use the ***Robot Operating System (ROS)*** [2] platform developed by Willow Garage [3]. We spent the term becoming familiar with ROS and the challenges.

In the winter term, we chose our robot: A ***Husky A200*** [4] that we bought from Clearpath Robotics [5]. Our work narrowed to more specific tasks, but we were still very much in "learning mode."

In the spring term, we used a simplified version of the ***Agile*** [6] software development process called ***Scrum*** [7] in which we set out deliverables due each week and re-planned each week based on what we did (and did not) accomplish. The diagram to the right summarizes the Scrum development process.

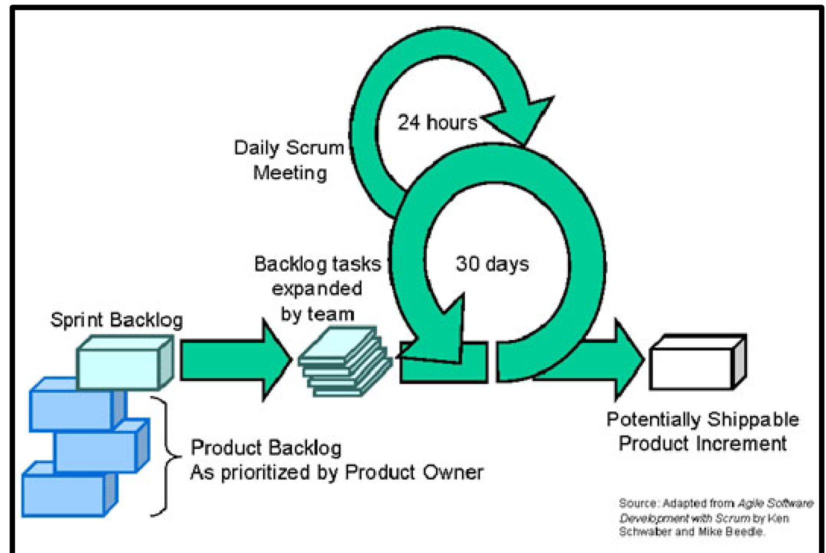


Figure 2. The Scrum development process **[15]**

3 Hardware

3.1 The Husky Robot from Clearpath Robotics

Our team wanted to focus on software challenges, per the course driving the team. For that reason, we bought a Husky A200 robot from Clearpath Robotics that provided a very strong, solid base – but only the base: wheels, motors, chassis, battery etc. No sensors and no brain.

- Dimensions: About 3 feet long, 2 feet wide, 15 inches tall, with a 5 inch clearance. It weighs about 100 pounds.
- Speed: It runs in rugged terrain (snow, etc) quite nicely, with a max speed of about 2.3 mph. It turns in a tight radius with a wheel base of about 2 feet.
- Power: Battery powered. It runs for several hours on a charge.

More details of the robot are at http://www.clearpathrobotics.com/husky.

3.2 Hardware We Added to the Basic Husky Robot

- A mounting system made from 80/20 aluminum t-slotted tube framing [8].
- Four cameras (mounted front and sides): Logitech HD Pro Webcam C920 [9].
- LIDAR: A used LIDAR loaned to use from an alumni.
- GPS: A BU-353-S4 Weather-proof USB GPS Receiver by GlobalSat [10].

- Computer: HP/Compaq Tablet computer, model TC 4400 (several years old).
- Safety system as required for IGVC.

Additionally, we are considering adding an electronic compass for heading information.

3.3 Hardware Innovation

We believe that the following will prove to be innovative about our hardware design: An off-the-shelf robot to provide stable, reliable locomotion, thus letting us focus on software challenges. Multiple cameras used for multiple purposes.

3.4 Costs of Equipment

Item	Retail cost	Cost to team
Husky A200 robot, with 1 extra battery (shipped, with tax)	$ 14,700	$ 7,600
Cameras (4) – Logitech HD Pro Webcam C920	$ 310	$ 310
LIDAR – SICK LMS291-S14	$ 5,000	$ 0
GPS – BU-354-S4 Weather-proof USB GPS Receiver.	$ 50	$ 60
Computer (HP/Compaq Tablet computer, model TC 4400)	$ 2,500	$ 0
Mounting hardware	$ 500	$ 100
Safety system (light, remote control, etc)	$ 800	$ 400
TOTAL	**$ 23,860**	**$ 8,470**

4 Software

This section describes the software we developed for our robot. ***Our development is still in progress – we will give a more complete report at the oral presentation at the contest.***

4.1 Platform – Robot Operating System (ROS)

The software platform chosen for use on our robot is ***Robot Operating System (ROS)*** [2], an open source software framework for use with robotics. It is a mostly self-contained system, which according to the ROS Wiki, provides “the services you would expect from an operating

system, including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. [2]” More simply put, it is similar to an operating system (although it has to run on a host OS; Ubuntu Linux is the only one officially supported), and uses a message-passing graph architecture to facilitate communication between various inputs, processing nodes, and outputs.

A more detailed overview of ROS is as follows (most of this is summarized from the ROS concepts page [11] located on the ROS Wiki): ROS has three layers of concepts and functionality: ***Filesystem Level***, ***ROS Graph***, and ***Community***.

The most important parts of the ***Filesystem Level*** are ***Packages*** and ***Stacks***. ***Packages*** are large units that may contain ROS nodes, ROS libraries, ROS configuration, or anything else that needs to go together. ***Stacks*** are collections of related packages. An example would be the Clearpath Husky stack; it contains packages for Husky initialization, Husky remote teleoperation, and Husky simulation.

The ***Graph Level*** of ROS is where most of the work is done. The graph level consists of ***nodes***: modular processes that, like Unix processes (and that similarly adhere to the philosophy “do one thing, and do it well”), typically do one specific task (control GPS, transform coordinate frames, control LIDAR, etc). Nodes communicate to each other by sending messages to each other over a routing system that utilizes named ***topics***. Nodes can either ***publish*** messages to a specific topic or ***subscribe*** to a topic, receiving (and handling) any messages that are published to it. Nodes may publish and subscribe to several different topics. There are also ***services***, which allow for synchronous communication (request/reply), but these are used relatively infrequently. “***Bags***” are another useful part of the Graph Level of ROS. They are a format for saving and replaying ROS message data, allowing one to work on algorithms without having to collect new data each time.

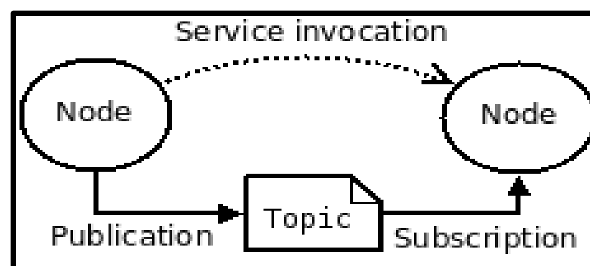


Figure 3. The ROS message-passing structure (from **[11]**)

The third level of ROS is the ***Community*** level, which consists of the online distributions and repositories of ROS. Our Husky robot is currently running the 4th version of ROS, ***Electric Emys***.

We chose ROS due to the large amount of available software for it, the large amount of compatibility, and its ease of development, due to being open-source. Our production/deployment system is running on Ubuntu 11.10, and our development systems are all running either Ubuntu 11.10 or Ubuntu 12.04. ROS supports code written in Python and in C++. Most of our high-level code is currently written in ***Python***, with calls to ***libraries implemented in C++***.

4.2 Vision

4.2.1 Vision for Lane Following (staying within the white lines)

The lane following algorithm implemented by our Python code is deceptively elegant. We chose to use the open source vision library ***OpenCV*** [12] because it is built into ROS and because, being freshman, it was a good way to be introduced to image processing.

An image read by OpenCV is stored as a two-dimensional array of pixels, where each pixel is a 3-tuple (e.g. the red/green/blue characteristics of the pixel). We realized early on that our system needed to process only a portion of each image, mainly just the ground. This revelation led to a ***crop*** method in our code. When an image is read into our code, the first process it goes through is the crop method which cuts off the entire top portion and leaves only the robot's immediate line of vision. The crop is accomplished by a call to an OpenCV function that restricts the field of interest (FOI) to the fixed area of the image that is the ground. (We know that area because we know how the camera is mounted, in particular, the angle at which it points toward the ground.)

While working on the crop method, we had also been thinking about a way to detect lines. Several options presented themselves to us. Since the lines would be white, we attempted to use a ***flood-fill*** operation and highlight everything white in the image. This process was eventually ruled out due to the severely discordant behavior of the operation. The flood-fill colored in not only the white lines, but every single white pixel in the image. Realizing that a more complex process would be needed, our team began to develop new ways to detect the while lines. Trawling through OpenCV's documentation and through code for prior projects, we did not leave empty handed. We realized that we needed to use the ***Hough Transform*** in order to draw lines onto the image. Though the Hough function can find straight lines in an image, it

would not be enough to implement it directly. After several weeks of experimentation, we came up with our current method to process the image for the Hough transform:

1. ***Crop*** the image to get just the ground (no sky).

2. The cropped image is sent to an OpenCV function that converts colored images to ***grayscale***. A purely black and white image gets rid of many issues and inconsistencies.

3. The gray image is then ***thresholded*** for white colors. The method of black and white thresholding turns every single pixel above a given value (in the range 0-255) to white and every pixel below the value to black. At first we used a fixed value as the threshold, but later we used an adaptive algorithm to choose the thresholding constant (see discussion below).

4. The preceding steps gave us an image where the lines were clearly defined as collections of pure white pixels, but it did not take into account that there are many random appearances of white still all over the picture. In order to detect the white lines properly, we needed to reduce the effects of the stray occurrences of white. A simple way to reduce the effects was to apply a ***blur*** to the image. We did not comprehend at the time that the blur would perform multiple functions. It would reduce the effects of stray white spots around the picture, and it would allow the Canny Edge Detection method to work more effectively.

5. The ***Canny Edge Detection*** method is a built-in OpenCV method that extracts the edges from a 2-d image. An edge can be defined as an inherent surface marking in the image that reflects the geometry of the shapes, or a sharp change in color. Therefore, the Canny Edge Detector is able to see the sharp contrast between the white lines and the rest of the black areas in the image. It would seem like edges would be easy to detect using Canny, however, the thresholding algorithm doesn't isolate the line as a single object; instead, the line becomes a heavily concentrated collection of white pixels. Therefore, Canny cannot identify a single solid edge, but it detects the line as a jagged edge with many fractures and

intervals of blank black space. The coordinates for each pixel in the edge are recorded in the image.

6. The Cannied image is finally sent to the ***Hough Transform*** method. This finds straight lines of a pre-set length in an image. The value returned by the Hough method is an array of lines defined by two points on the original image. We took the returned array from Hough lines and iterated through it, simultaneously drawing each line onto the original image. The result was the original cropped image with red computer generated lines along the white painted grass. These lines are then sent to ROS to use in its navigation algorithms, as described later in this document.

At this point we thought that our line following algorithm was done and we could focus on other aspects of the project, but we were wrong. In attempts to test our line following algorithm on other test pictures, we found that our threshold did not work for images with differing values of brightness. As the overall value of brightness in the image changed, our value to threshold needed to be changed also. Therefore, our ***adaptive thresholding*** method was instated to perform that operation.

In order to find the value of brightness to threshold for in a given image, we had to find the overall brightness of the image. The OpenCV function ***calcHistogram*** was used to find the number of pixels at each level of brightness from 0 to 255 in the grayscale image. Based on the skew of the histogram, we could threshold for only the brightest values (which would be coming from white lines). Our thresholding operation iterates through the histogram to find the value of brightness with the largest number of corresponding pixels. It then iterates through the rest of the histogram until it reaches a brightness at which there only about a sixth of those pixels. That value of brightness is recorded as the threshold value.

4.2.2 Vision for detecting barrels

For our barrel detection system we were unsure of how to effectively isolate and record the positions of the barrels from the camera. The main reason for this was that the barrels can be any color, from orange to green. As a result, we decided to break our algorithm development process into several phases. The first phase was to find the barrels based on their orange and white alternating colors. We do this by stepping from the bottom row of the image up to the

top, row by row. We then look for a block of orange pixels in the row. Once this is found, we step upwards; looking for a block of white pixels above. Once this is found, we can safely say that we have found a barrel and can return its coordinates.

The second phase was to find an algorithm to isolate a barrel based somewhat on color and mostly on shape. This way we can operate on any barrel regardless of color. The algorithm we are using for this is the ***Histogram of Oriented Gradients*** algorithm, or ***HOG*** [13]. The HOG algorithm is commonly used to detect pedestrians in crosswalks. It can successfully find people regardless of the color of their clothes. This algorithm shows much promise for identifying any and all barrels from the camera. We are in the process of implementing and applying HOG to finding barrels; we will report more on this in the oral presentation at the contest.

4.2.3 Vision for Obstacle Detection

We detect obstacles, including their distance from the robot, using a vision algorithm that J.P. Mellor, Professor of Computer Science and Software Engineering, taught us. The algorithm works using a single camera, mounted pointing straight away and somewhat toward the ground, that takes pictures over time.

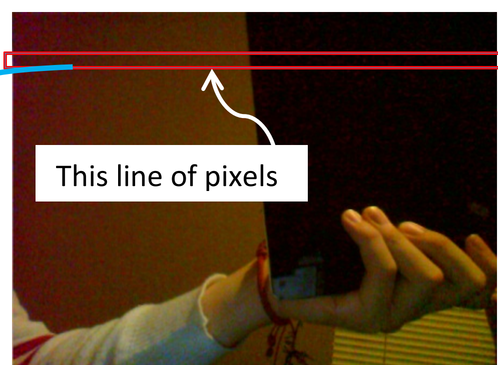


Figure 4. Capture one line of pixels

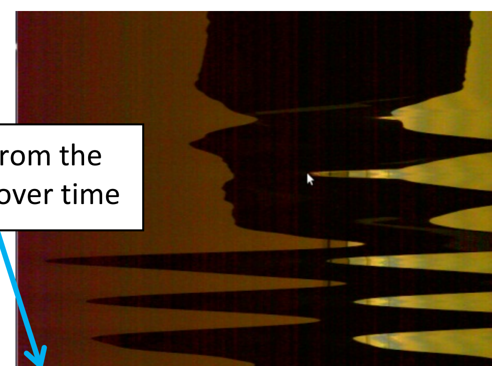


Figure 5. Append lines to the bottom of the *stack graph image*

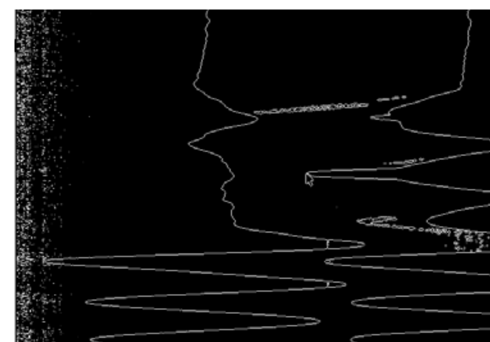


Figure 6. Threshold to grayscale

__The algorithm__:

Start with a blank ***"stack graph" image*** and repeat the following every 10 milliseconds:

1. Capture one line of pixels from the camera (Figure 4.) While any line could work, we chose a line to have accurate distance information for barrels when they get within about 20 feet.
2. Append the new line to the bottom of the stack graph image (Figure 5). Thus, as this loop continues over time, the stack graph image shows the same line of pixels repeated over time, with time being the y-axis.
3. Apply the OpenCV Canny function (as described earlier) with appropriate thresholds to get edges of the objects (Figure 6).

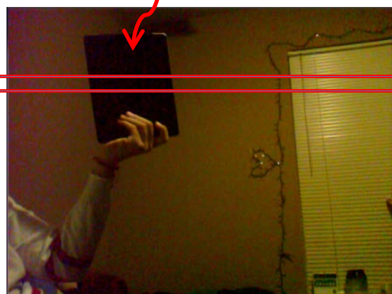
4. In the thresholded image, the lines are edges of objects. Lines that are near vertical are for objects whose edges are not changing over time – those are objects that are far away. Lines that are not near vertical are for objects whose edges *are* changing over time – those are objects that are close. The smaller the slope of the line, the closer the object. In particular, the distance of the object whose edges are lines in the stack graph image is given by the formula (where we use the slopes near the bottom of the stack graph image):

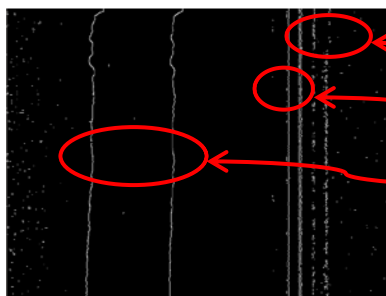
$$Distance = k * slope\ of\ its\ lines * speed\ of\ the\ robot$$

<u>Why the algorithm works</u>: Here is an example to illustrate the algorithm. In this example, the camera is moving toward the black object held in my hand, thus simulating the robot with its camera moving toward the black object. The images on the left are snapshots ***over time*** of the actual image; the images on the right are the ***stack graph images*** developed over time.

The black object that the camera will move toward.

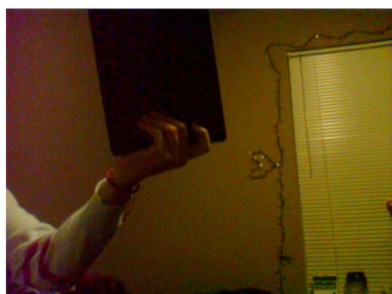
The line of pixels to be sampled repeatedly over time.

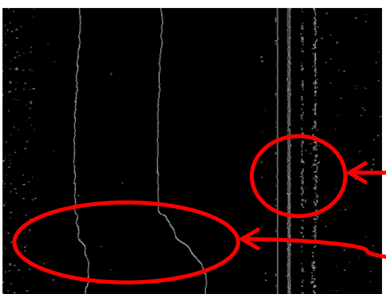




At first, nothing is moving, so all the lines are vertical. The lines are from:

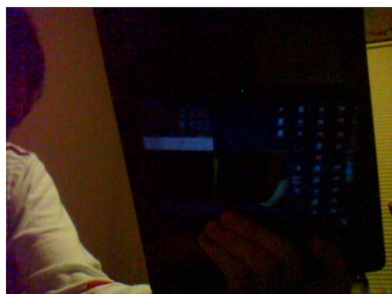
- The window
- The black cord next to the window
- The left and right edges of the black object in my hand

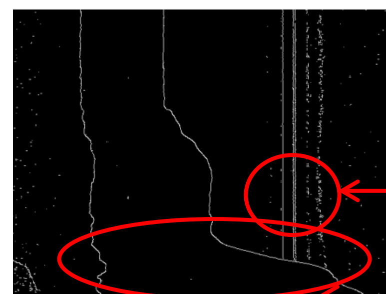




Now the camera has moved closer to the black object in my hand.

- The far-away objects have not changed much in the images sampled over time, so their lines remain near vertical.
- The lines for the black object start to change their slopes toward horizontal, indicating that the black object is closer.





Now the camera has moved even closer to the black object in my hand.

- The far-away objects still have not changed much in the images sampled over time, so their lines remain near vertical.
- The lines for the black object are even more toward horizontal, indicating that the black object is still closer.

4.3 LIDAR

Although we have a LIDAR mounted on the robot and we know that ROS can handle LIDAR data, we have not yet written code to take advantage of that. We hope that our vision algorithm for detecting distances of objects will suffice; eventually we want to use that vision algorithm along with LIDAR data to increase reliability of our distance sensing.

4.4 GPS

We have written code that takes in GPS data. The next section shows how that data will be used for navigating to waypoints.

4.5 Navigation

For navigation functionality, the ROS ***Navigation Stack*** is used. It contains a variety of components that work together with other nodes as part of a larger system. The simplest explanation of it is that it takes sensor data and localization data, then signals the robot's control surfaces to take it to a given destination.

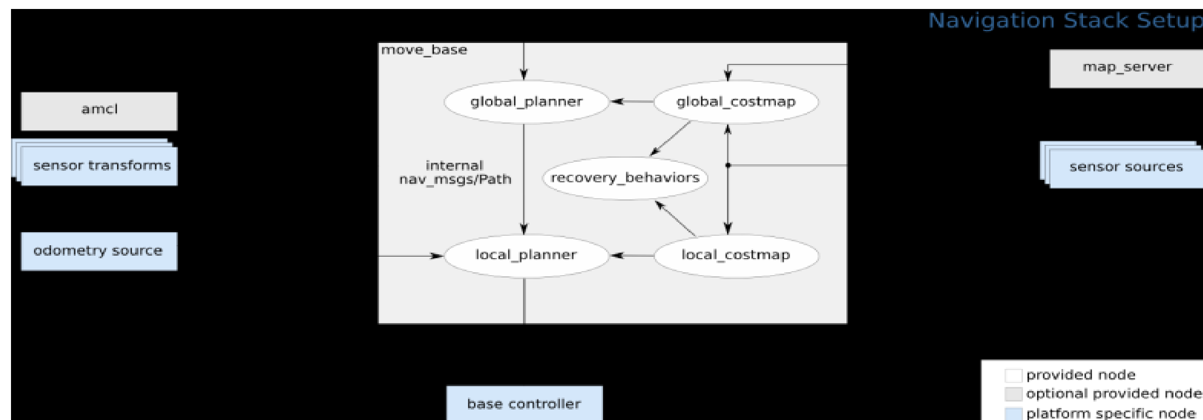


Figure 7. ROS Navigation Stack Layout

A more complex explanation of the ROS Navigation stack is as follows: The navigation stack is a series of nodes that work together to give the robot navigation functionality. The internals of the navigation stack consist of a ***global costmap***, a ***local costmap***, a ***global planne***r, a ***local planner***, and a ***recovery behaviors*** node.

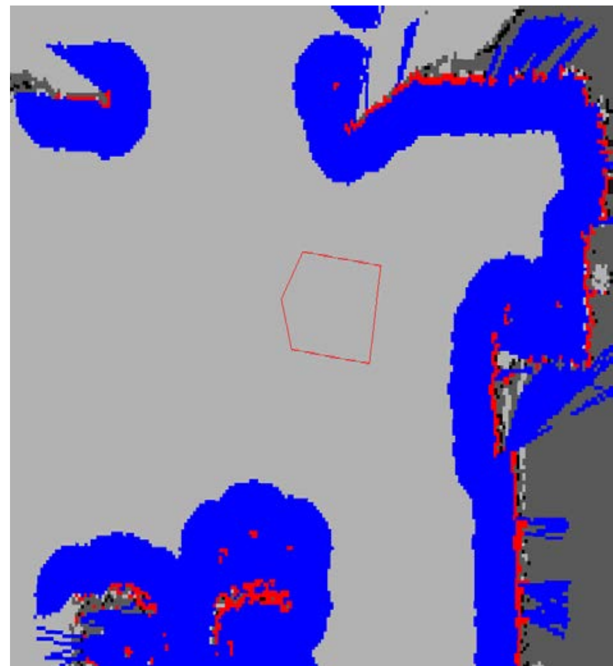


Figure 8. A Costmap Visualization

Costmaps are the primary way by the robot knows where it can and cannot go, and what paths are best to take. They are built by taking in sensor data (whether from a laser scan [usually output by LIDAR] or point cloud) and using this to produce an occupancy grid (a 2D or 3D grid of places the robot can and cannot go). The “cost” part of the costmap is that each grid space has a “cost” associated with it. The robot avoids spaces with high cost (corresponding to high probability of collision), and prefers to go through spaces with low cost (corresponding to low or no probability of collision). These are calculated by giving the robot an “inscribed radius” (corresponding to a circle drawn inside the robot's body) and a “circumscribed radius” (corresponding to a circle drawn outside the robot's body). Obstacles that would fall within the robot's inscribed radius have a very high cost, while obstacles that fall within the robot's circumscribed radius have a slightly lower cost, and obstacles that fall within neither have practically no cost. The use of costmaps allows the ROS navigation stack to deal with many types of obstacles in an elegant manner.

The second part of the navigation stack is the ***planner*** nodes. These use the costmap to attempt to find the “cheapest” route from the robot to a specified goal. After such a route is found, it sends differential and angular velocities to the robot. The algorithm used by the planner node is as follows (quoted from [14]):

1. Discretely sample in the robot's control space (dx, dy, dtheta)
2. For each sampled velocity, perform forward simulation from the robot's current state to predict what would happen if the sampled velocity were applied for some (short) period of time.
3. Evaluate (score) each trajectory resulting from the forward simulation, using a metric that incorporates characteristics such as: proximity to obstacles, proximity to the goal, proximity to the global path, and speed. Discard illegal trajectories (those that collide with obstacles).
4. Pick the highest-scoring trajectory and send the associated velocity to the mobile base.

5. Rinse and repeat.

Finally, the recovery node rotates the robot 360 degrees when the robot appears to be “stuck”.

To make the navigation stack functional, a few things are required: sensor inputs, robot controller output, odometry inputs, and a map server (for sharing the map with other nodes, as well as saving it to the disk). For sensor inputs, we are using a SICK LIDAR system mounted to the front of the robot, as well as a mounted camera. For odometry, the robot has an NMEA-compatible GPS device, as well as encoders on the wheels. Given these various sources of input, some are combined by using an EKF filter, and they are sent to the navigation stack, enabling navigation. At this point, the navigation stack simply needs to receive a “goal message”, specifying a desired destination location. To do this, given GPS information, a “gps_common” node is used, which converts the spherical projection of GPS coordinates into UTM coordinates, which are rectangular coordinates that the navigation stack can understand.

When all of this data is fed into the navigation stack, the robot can navigate from waypoint to waypoint while avoiding obstacles and automatically adjusting its speed. The challenge inherent in this is combining the sensor data and tuning the navigation stack's settings to give a balance of processing speed and accuracy (by adjusting the grid size of the costmap, making it 2D vs 3D, etc), as well as adjusting the acceleration, speed, and radius parameters of the robot to allow it to go from point to point as quickly as possible while not coming into contact with any obstacles.

4.6 Software Innovation

This is our first attempt at this contest and we started from scratch, with no expertise in ROS or vision algorithms. Nonetheless, we believe that the following will prove to be innovative about our software design when it is fully implemented (perhaps not until next year): 1. Our use of ROS as a platform. 2. Our vision algorithms to detect white lines and barrels. 3. Our use of a single-camera algorithm that operates over time to detect distances of objects. 4. (Eventually, but probably not this year): Our use of multiple cameras to increase reliability of the robot’s understanding of the world. 5. (Eventually, but probably not this year): Our use of speed-control algorithms to trade off speed versus the uncertainty of the sensor data.

5 Performance analysis / testing

As of this writing, we have done very little real-world testing; we have lots to do in the forthcoming weeks! We plan the following tests:

- ***Test speed and agility*** by running the robot via remote and autonomous control at slow speeds (when obstacles are close by) and fast speeds (when obstacles are far away). We expect our slow speed to be about 1 mph and our fast speed to be about 2 mph.
- ***Test ramp climbing ability*** by running the robot on the course we have set up (small natural hills) and on a bicycle ramp (10 degree incline). We expect that the robot can climb inclines up to 45 degrees.
- ***Test reaction times*** by running the robot on the course. We expect that we can obtain sensor data every 50 milliseconds and react to it in real time.
- ***Test battery life*** by running the robot through the entire outdoor course that we have prepared. We expect about 3 hours of run-time under contest conditions.
- ***Test distance at which obstacles are detected*** on our course. We expect to identify the white lines to a distance of 30 feet. We expect to identify the barrels at a distance of about 30 feet and to detect their distance accurately when they are within 20 feet.
- ***Test how the vehicle reacts to complex obstacles*** such as switchbacks, dead ends, etc. by running on the course. We are unsure how well the ROS navigation will do.
- ***Test accuracy of arrival at navigation waypoints*** by running on the course. We expect to be able to arrive to given waypoints within 1 meter.

At the oral presentation at IGVC we will supply comparison of predictions with actual data.

6 Conclusion

This is our first year in IGVC. We are still implementing ideas. We will update this report at the oral presentation at IGVC. We think that these will prove to be winning design decisions:

- Buying the Husky A200 as our robot base. This allows us to focus on our interests – software and sensors – instead of grappling with mechanical issues.

- Using ROS as our platform. The learning curve is steep, but we believe that ROS will let us focus on applying known algorithms in novel ways, instead of re-inventing the wheel.
- Our vision algorithms, especially our single-camera algorithm for distances over time.

7 References

[1] "IGVC Design Reports, 2008, Princeton University," 2008. [Online]. Available: http://www.igvc.org/design/reports/dr218.pdf. [Accessed 8 May 2012].

[2] "Robot Operating System (ROS)," [Online]. Available: http://www.ros.org/wiki/. [Accessed 8 May 2012].

[3] W. Garage, "Willow Garage home page," [Online]. Available: http://www.willowgarage.com/. [Accessed 8 May 2012].

[4] "Husky A200 Unmanned Ground Vehicle," [Online]. Available: http://www.clearpathrobotics.com/husky. [Accessed 8 May 2012].

[5] "Clearpath Robotics," [Online]. Available: http://www.clearpathrobotics.com. [Accessed 8 May 2012].

[6] "Agile Software Development Processes," [Online]. Available: http://www.agile-process.org/. [Accessed 8 May 2012].

[7] K. Schwaber and M. Beedle, Agile Software Development with Scrum, Prentice-Hall, 2001.

[8] "80/20 T-Slotted Aluminum Tube Framing," [Online]. Available: http://www.8020.net/T-Slot-1.asp. [Accessed 8 May 2012].

[9] "Logitech HD Pro Webcam C920," [Online]. Available: http://www.logitech.com/en-us/webcam-communications/webcams/devices/hd-pro-webcam-c920. [Accessed 8 May 2012].

[10] "BU-353-S4 Weather-proof USB CPS Receiver by GlobalSat," [Online]. Available: http://www.usglobalsat.com/p-688-bu-353-s4.aspx#images/product/large/688.jpg.

[11] "ROS Concepts Page," [Online]. Available: http://www.ros.org/wiki/ROS/Concepts. [Accessed 8 May 2012].

[12] "Open CV Vision Library, Wiki," [Online]. Available: http://opencv.willowgarage.com/wiki/. [Accessed 8 May 2012].

[13] "Histogram of Oriented Gradients (HOG)," [Online]. Available: http://en.wikipedia.org/wiki/Histogram_of_oriented_gradients. [Accessed 8 May 2012].

[14] "ROS Dynamic Window Approach (DWA) Planner," [Online]. Available: http://www.ros.org/wiki/dwa_local_planner.

[15] "Scrum Services from Icon ATG: diagram adapted from "Agile Software Development with Scrum" by Schwaber and Beedle," [Online]. Available: http://www.iconatg.com/services/process/scrum.php. [Accessed 8 May 2012].