RoboGoat 2013

United States Naval Academy, Systems Engineering Department

By Midshipman:

Cory Oberst (SR)

Keegan Wetzel (SR)

Jason Case (SR)

Kelsey Learned (SR)

Adviser Statement: I certify that the design and engineering of the vehicle by the current student team has been significant and was appropriate for a capstone senior design course.

Adviser: Assoc. Prof. Joel M. Esposito ___________________________

New For 2013 Entry:

U-turn detection and correction algorithm

Flag detection and path planning

New Laser Range Finder

Android tablet interface

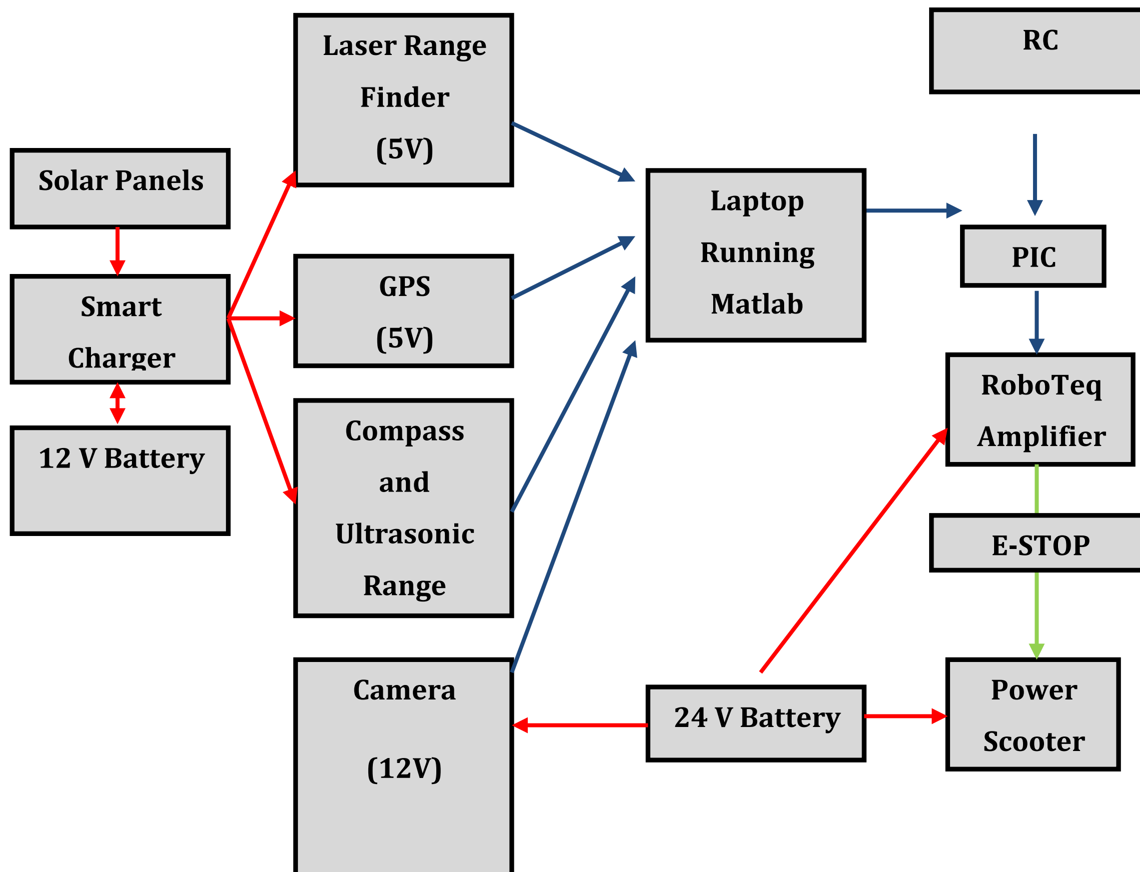
"Leash" manual control system

Arduino based navigation system

Overview

The Robo-Goat is designed around four principles:

1. Maximize off the shelf hardware use.
2. All processing done by one laptop running Matlab
3. A system whose state and world view are easily visualized and controlled by the developers, can be debugged more efficiently.
4. Mapping is not necessary to complete the competition tasks and introduces unnecessary complexity.



Innovation: We believe these are the most innovative features of the RoboGoat.

- Frame Design: As compared with our previous entries, this version is much closer to ideal. The robot has a nearly minimal 2' 4" by 3' 2" footprint and maximal height of 6' -- making vision and obstacle avoidance easier.
- Solar Power: it uses solar power to charge a battery while running the onboard electronics, including the laptop, eliminating battery life as one potential limiting factor when testing.
- Robust Vision System: in our experience, this is the most challenging component of the competition. Our system uses a combination of alternate color spaces, automatic background detection and a new shadow compensation algorithm to adapt to challenges arising from variable lighting conditions.

- User Interfaces: We subscribe to the adage: if you can't see what the system is thinking, you will not be able to debug or improve it. To that end we have implemented a variety of user interfaces: a robot centered world view, a remote control model that can be switched to on the fly, an integrated power system display and a camera threshold selection tool.
- Maximize Off the Shelf Components: When possible we try to focus on the planning and perception algorithms rather than building custom hardware.
- Runs Entirely in Matlab: We exploit Matlab's extensive library of image processing routines, statistics toolbox, GUI and visualization tools. Programming in Matlab enables rapid prototyping of code, and makes visualization easy. Despite Matlab's reputation for being slow, with proper coding technique and a new laptop we are updating at 10 Hz. We believe we are the only entry running entirely in Matlab.
- Android Tablet Interface: Instead of a traditional RC controller this year we have opted for an Android controller using a GUI that interfaces via a Bluetooth serial port. Using the built in accelerometer of the tablet we are able to give drive commands to the vehicle. Also this enables us to remotely initialize our serial ports instead of having to be physically at the laptop.

Design Decision Process

Characteristic	Option 1	Option 2	Option 3	Option 4
Vision Hardware	Auto Iris Camera	USB Camera	Stereo Camera System	-
Vision thresholds	Fixed	Global Adaptive	Locally Adaptive	-
Obstacle Avoidance	Ultra-Sonic Sensors	25 Hz Laser Range Finders	10 Hz Laser Range Finder	Camera System
U-turn Correction	Non-existent	GPS Path Tracking	Camera Path Tracking	
User Interface	Purely on Laptop	Android Tablet	Remote Control	Wii Remote
Flags	Detect	Detect and Path Planning	Ignore	

Background: The RoboGoat capstone project is a legacy project that began in 2009. Every year the vehicle has competed in the IGVC (Intelligent Ground Vehicle Competition). In 2009 the Goat placed 20th, in 2010 and 2011 it placed 10th, and in 2012 we placed 3rd overall and 2nd in the navigation challenge. This year, we hope to remain in

the top 3 and hopefully win the competition. Last year, a large issue we ran into was the likeliness of a u-turn in sharp corners. In the previous competition we wasted half of our runs because in one direction the vehicle always made a u-turn in the beginning portion of the course. Also we found that having to crouch to get to our laptop and initialize all of our sensors and ports was not ideal. Last year we also were able to reach the flags after time had run out but we did not have any flag detection algorithms. This year we have designed a system so that if we reach them again we will be able to navigate the flags. After identifying these weaknesses, our team developed the following morphological chart to help identify the solutions. **The hi-lighted sections of this chart represent the key portions of our design as well as the major additions made to our vehicle this year.**

Android Interface

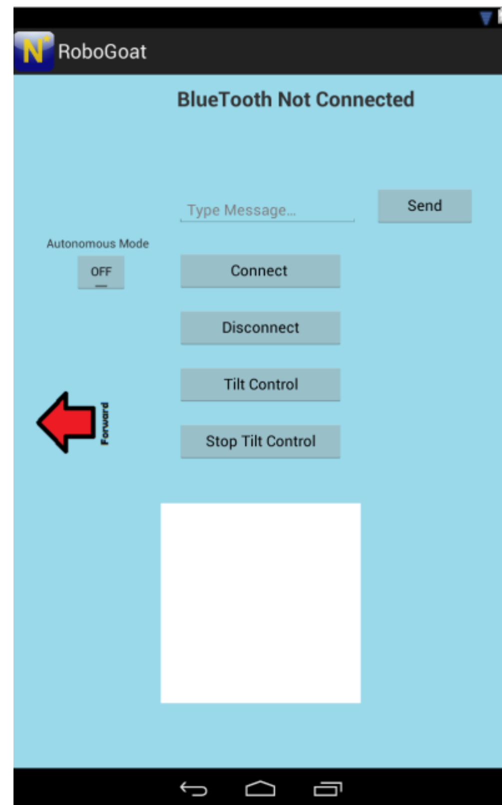
For this year's edition of RoboGoat, we decided to update and improve the user interface. We developed two methods for user interaction with our vehicle. One method utilizes the vehicle's computer vision system to manually drive the vehicle using a colored leash, another uses an Android tablet to control the vehicle wirelessly through Bluetooth.

To interface with the Android Nexus tablet we developed an Android application which runs a simple GUI and transmits and receives data through Bluetooth's serial port profile. The application minimizes the need for user interaction with RoboGoat onboard laptop computer. All the user has to do is run one Matlab program on the laptop and from the user can perform all necessary operations from the tablet. The GUI has four important capabilities. The first is that it can set up Bluetooth communication to the onboard laptop. When the user runs the main program on the onboard laptop a serial port is opened through the laptop's Bluetooth adapter. When the user presses the connect button on the GUI (see figure) the Android application connects to this serial port and enables serial communication. The second capability is that the tablet can be used as a remote control to drive the vehicle manually. We do this by reading the tablet's built in accelerometer sensors and streaming the raw data to the laptop computer. Once the laptop receives the data it is scaled and passed through a simple moving average filter to filter out and high frequency noise. The data is then sent to the vehicles Roboteq board to drive the motors. The third capability is the GUI's ability to receive data from the laptop. When running in autonomous mode the GUI is able to receive live images from the vehicle's vision system as well as other data such as heading, waypoint distance, and state and error information to be displayed as strings for the user. The fourth capability is the ability to send string commands to the laptop. This is particularly useful for initializing sensors (serial ports) in Matlab as well as controlling the vehicle's state. For example, in the current set up the user is able to run and stop our autonomous code by simply toggling a switch button on the Android GUI.

The tablet is particularly useful for vehicle testing because it allows the user to minimize time spent crouched to work on the vehicle's laptop (see figure) thus allowing for more comfortable and efficient testing.

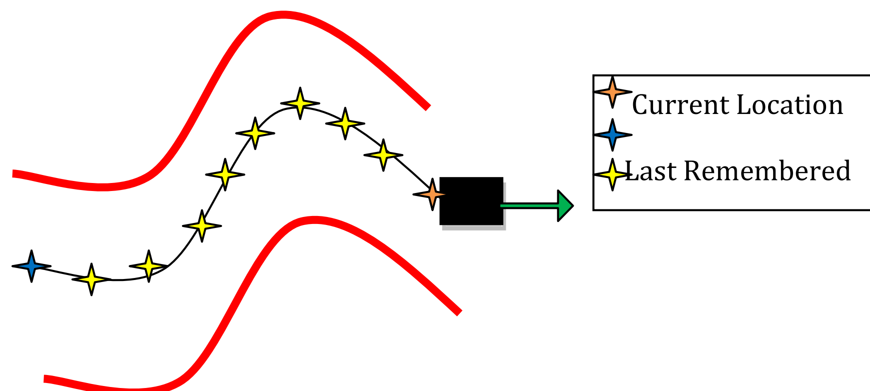
Before

After



U-turn Detection and Correction

In 2012 we ran in to a major problem involving the vehicle making a complete u-turn at the beginning of the course and going in the wrong direction until we had to stop it. Our vehicle had no way to remember where it had been so it never knew it was going the wrong way. In previous years we had experimented with logging all our previous GPS points but we found that this slowed down the computer and used too much memory. This year we added a u-turn detection and correction algorithm using the GPS. The idea behind this is that the vehicle will remember its locations for the last 10-20 seconds therefore creating a tail of GPS points as shown below.



In order to get the tail of GPS points we start out with an empty array of a fixed length and begin logging our GPS points. Each time we get a new point the old points shift right and the new point is placed at the beginning of the array. This allows the vehicle to drag a tail of its old locations behind it without bogging down the memory of the computer. The basic idea behind detecting a u-turn is that if the current location and last remembered GPS point are within the distance of half the lane (we used 2 meters) then the vehicle has made a u-turn. However we found it to be slightly more complicated than this. If the vehicle was standing still or had moved too slow then the algorithm would detect a u-turn because the vehicle had not been able to move outside of the u-turn detection distance. To fix this we found the length of the tail by summing the distance between every point on the tail. This basically finds how far we have moved in the last 10-20 seconds based on the length of the tail. Then we only check if the first and last points are too close together when the tail exceeds a length minimum that is equal to 3 meters or just barely over the u-turn detection distance. If the tail is long enough and the first and last points are within 2 meters then the vehicle will realize it has made a u-turn at some point.

The next step in this algorithm is the correction portion. In order to do this we must know what direction we are heading. Therefore, once a u-turn has been made and realized by the vehicle the code stops and records the current heading from our compass. Then that data is sent to our drive controller and says that the desired heading is equal to the current direction ±180°. Then the vehicle will turn around and once it is on the correct heading will continue the obstacle course. Then the GPS tail array is cleared and begins to fill again to check for a u-turn.

Arduino Navigation Board

The navigation board previously used on the RoboGoat was built in house and featured a Rabbit 3000 microprocessor and hosted a wireless Xbee module, 3-axis accelerometer, 3-axis magnetometer, a GPS receiver, and numerous I/O ports (including analog, serial, and interrupts). We found that this board sufficed but when there was a problem with it we did not know how to troubleshoot it because of the lack of knowledge on our end. So this year we have opted to use an Arduino UNO board. The purpose of the board is to read the compass. We used this because the old navigation board had many unused features and this was a very simplistic idea. We also have the inputs of the UNO set up to read ultrasonic sensors for emergency stops for obstacles. The UNO sends the serial string to MATLAB where the actual computations are made.

The compass is calibrated by recording the raw numbers from the magnetometers while spinning the RoboGoat in a circle. The result when the x and y axis are graphed simultaneously is a circle. The values needed to zero the compass are the x and y coordinates of the center of the circle, taken by averaging all the x and y values separately. These numbers are then written into the MATLAB code that reads the data from the serial object through which the laptop and the Arduino UNO communicate. These numbers are added to the raw data so that heading can be accurately determined. If improperly calibrated, typical errors include all headings being in the same quadrant or headings appearing to skip a quadrant when the RoboGoat is rotated.

Flag detection and path planning

The goal of the our line-fitting code was to use the Hough Transform via the HoughLines MATLAB function in order to 'connect the dots' from each flag of a color to the next flag of the same color. HoughLines works by extracting line segments from particular bins based on the Hough Transform. Many parameters were experimentally tweaked to fit lines to our flags which were of reasonable usefullness when applied to driving the IGV autonomously. These parameters included rhores, the resolution in determining the distance of a line, thetares, the resolution for determining the angle of a line, FillGap, the maximum number of pixels allowed to fill a break in a line, and MinLength, the minimum number of pixels used to compose a line, among other parameters.

Once lines had been fit to keep the IGV within the flags, we began brainstorming methods to allow our vehicle to safely enter the flag lines which had been drawn. Many ideas were considered but the method pursued was that of essentially drawing a funnel from the flag nearest in distance to our camera. Using a funnel we believed that there was a high chance of successfully guiding our vehicle into the flag channel without running over any flags and without ending up on the wrong side of the channel. The angle of the funnel was computed by first finding the angle at which the flag line had been drawing; named flag_theta. For the red flags, from which we desired to remain on the left side when navigating the flag channel, the angle of our funnel line was computed using :

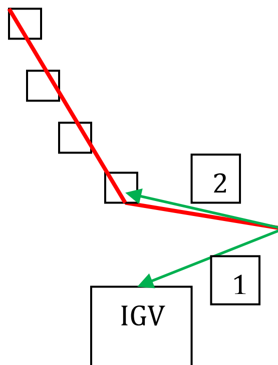
$$funnel_theta = flag_theta + .7 * pi$$

Likewise for the blue flags, from which we desired to remain on the right side when naviagting through the channel, the angle of our funnel line was computed using:

$$funnel_theta = flag_theta - .7 * pi$$

The funnel lines were then drawn by taking the cosine and sine of funnel_theta, multiplying this value by a constant to add magnitude (100 was a common value used), and adding the multiplied cosine and sine values to the respective x and y values for the end of the red or blue line nearest to the camera.

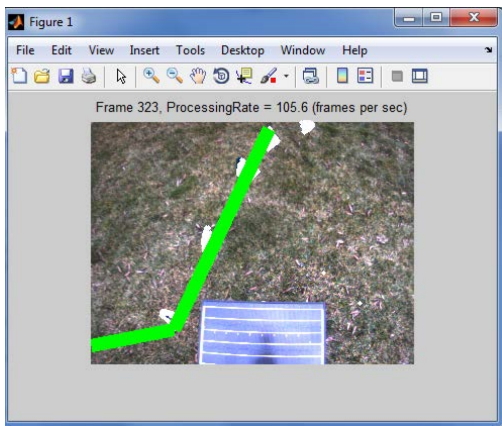
The final goal of our line-fitting code was to implement a script that could detect whether our IGV was on the correct or incorrect side of a certain color flag. Essentially, if we found ourselves on the right side of the red flags, or on the left side of the blue flags, that our vehicle could have some alert that it is driving an incorrect course. Our method for computing this information was to use the cross product between the vector comprised of the points from the end of the funnel line to the front of the IGV and the vector comprised of the points from the end of funnel line to the end of the flag line.

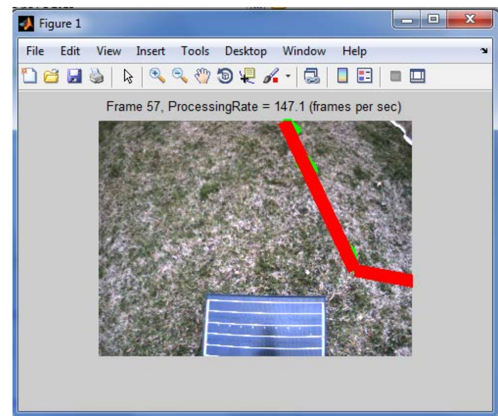


In the picture above the small squares represent red flags, the large square represents the IGV, the red lines represent the flag line and funnel lines drawn, and the green vectors represent the vectors of which the cross product is obtained as shown below:

$$CrossProductRed = 1x2$$

The scripts CrossProductRed and CrossProductBlue have been written to take the Cross product as explained above and display whether or not the IGV is on the correct or incorrect side of the flag channel based on which color flags it is detecting.





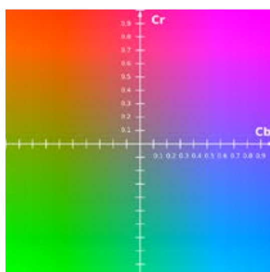
Visual Command Addition to the RoboGoat

The goal of the visual command feature of the RoboGoat was to utilize the Goat's camera and imaging processing in MATLAB to take commands from the orientation of an object. The Goat would be able to follow the object at a safe distance and speed by varying the angle it drove and the speed it went. In order to accomplish this, a striped leash was used. The camera picks up the stripes and then calculates a turn angle from the direction it is pointed and a speed based on how slack the rope is. Some imaging processing techniques used were binarizing the image, closing the objects detected, clearing any objects detected that touch the edge of the image, using regionprops to measure the major axis length of the stripes, and calculating Hough Lines of the detected objects. This creates a simple way to drive the RoboGoat around. It essentially is a less expensive remote control for the vehicle.

Lane Following System

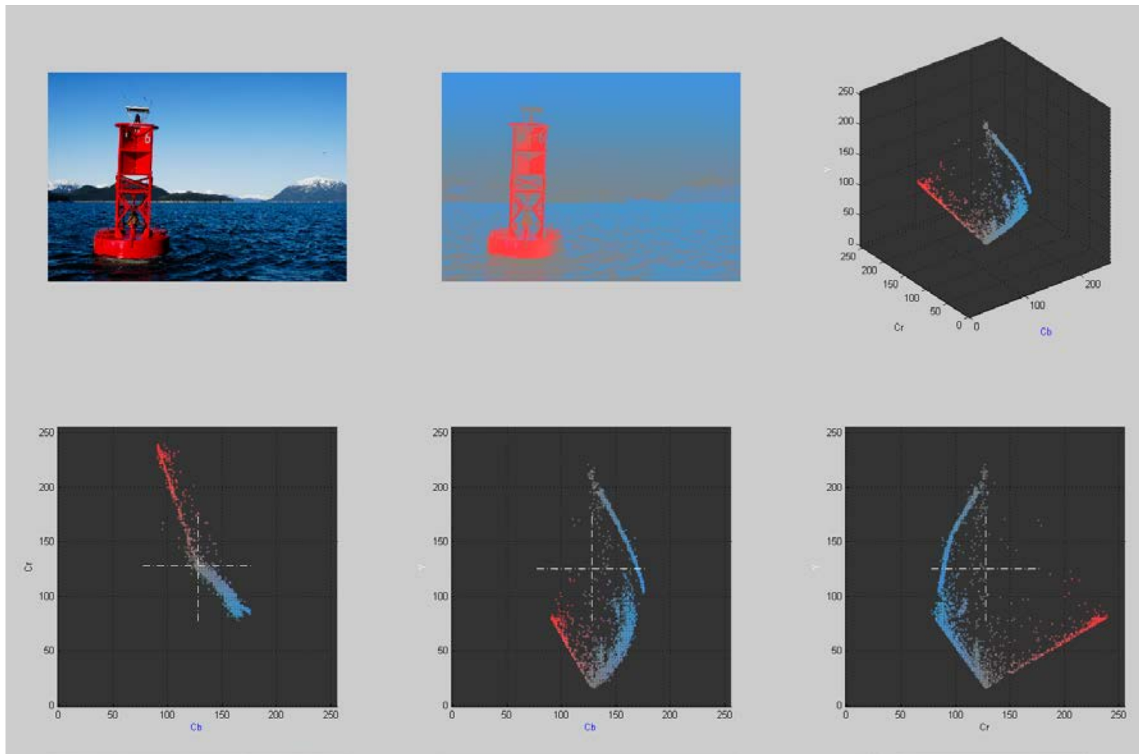
Hood, mount and filter, field of view: The camera is mounted underneath a hood to reduce glare from the sun overhead and uses a circular polarizing filter. It is mounted at about 70 inches above the ground, so that it has the largest view of the ground as possible. It is angled downwards, such that the bottom of the viewing window sees 6-inches from the front of the robot. This is to reduce the size of a blind spot directly in front of the robot. At this angle, the camera sees 24-inches wider than the robot on each side.

Why YCbCr: The color space that has been chosen for this project is YCbCr. The Y stands for brightness, while Cb and Cr refer to color values (below). If the user only looked at the Y values of an image, they would see a grayscale image with the black areas having a Y-value of 0 and the whitest areas having a Y-value of 1. This color space is handy for the RoboGoat because it distinguishes color from brightness. This means that the thresholds selected with Cb-Cr are more stable in differing lighting conditions, because Cb-Cr values refer to "pure" colors and are not affected by shades or shadows.

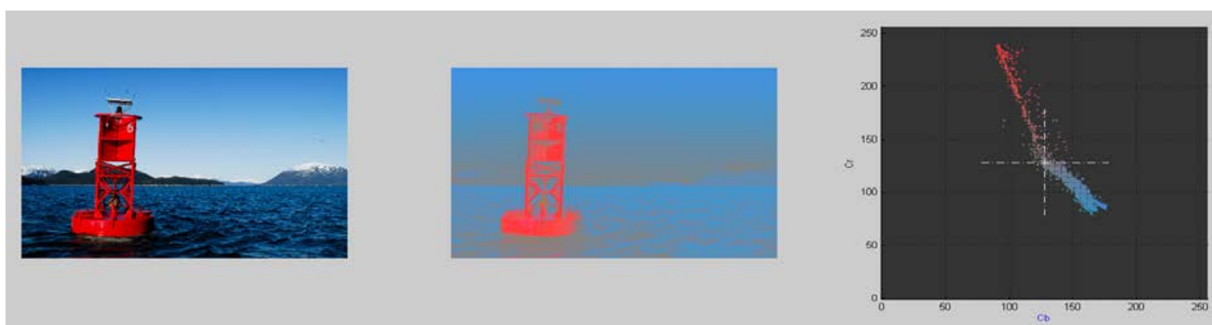


Cb-Cr color plane, Y set to 0.5

The 6-Plot Figure is shown below and is the most powerful source of image information in the program. The top three plots, moving left-to-right, display: the original image (top-left), the image shown in YCbCr with a uniform/normalized brightness level (top-middle), and a rotatable 3D scatter plot of the image's pixels within the YCbCr color space (top-right). The colors of the dots in this 3D scatter plot match the colors in the YCbCr image, so the user can see what part of the image they refer to. The bottom three plots, moving left-to-right, display the three 2D perspectives of the 3D scatter plot: Cb-Cr (bottom-left), Y-Cb (bottom-middle), and Y-Cr (bottom-right).

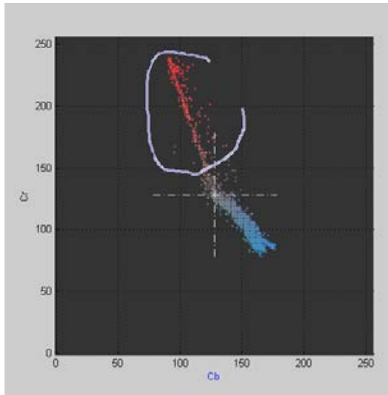


The three 2D graphs are very handy. In the example of using the picture of the red buoy, all three 2D plots show the dots for the red buoy from various angles of the 3D graph. The dots in first 2D graph (Cb-Cr), however, include <u>every value of Y (brightness)</u>. This means that if the red dots are selected from the Cb-Cr graph, every shade of the red buoy will be included. The other 2D graphs have Y in the vertical axis, so the red shade-ranges are spread vertically up and down the graph. But if the user wanted to <u>specifically</u> choose the lightest or darkest red pixels, they could choose the Y-Cb or Y-Cr graphs. Once the user chooses the 2D graph that they would like to use for threshold selection, the program will display a new window. This window displays three items: (left-to-right) the original image, the image shown in YCbCr with uniform/normalized brightness level, and the selected 2D graph.



Original Image *YCbCr view, uniform brightness* *Distribution of pixels in color space*

In the 2D graph, the user will freehand-select (draw) around the pixels they want to threshold. This is done by left-clicking in the graph, holding down the mouse button, and dragging the cursor around the desired dots. Figure 3 shows what the freehand selection looks like, in-progress:



The program reads the coordinate values of every selected pixel. In this example, the coordinates are Cb-Cr values (just like the x-axis and y-axis). The program then takes the maximum and minimum values from both axes - these four values become the thresholds.

Solving the "Barrel Problem": Certain objects on the IGVC course pose a problem for the robot's vision system. The robot is designed to use its camera to look at chalked/painted white lane lines on the ground, using them as a visual reference to stay within the lane. There are many obstacles to avoid along the route, the most common of which is an orange construction barrel. The figure below shows a picture of several such barrels on the IGVC course - the lane line is visible in this figure too.

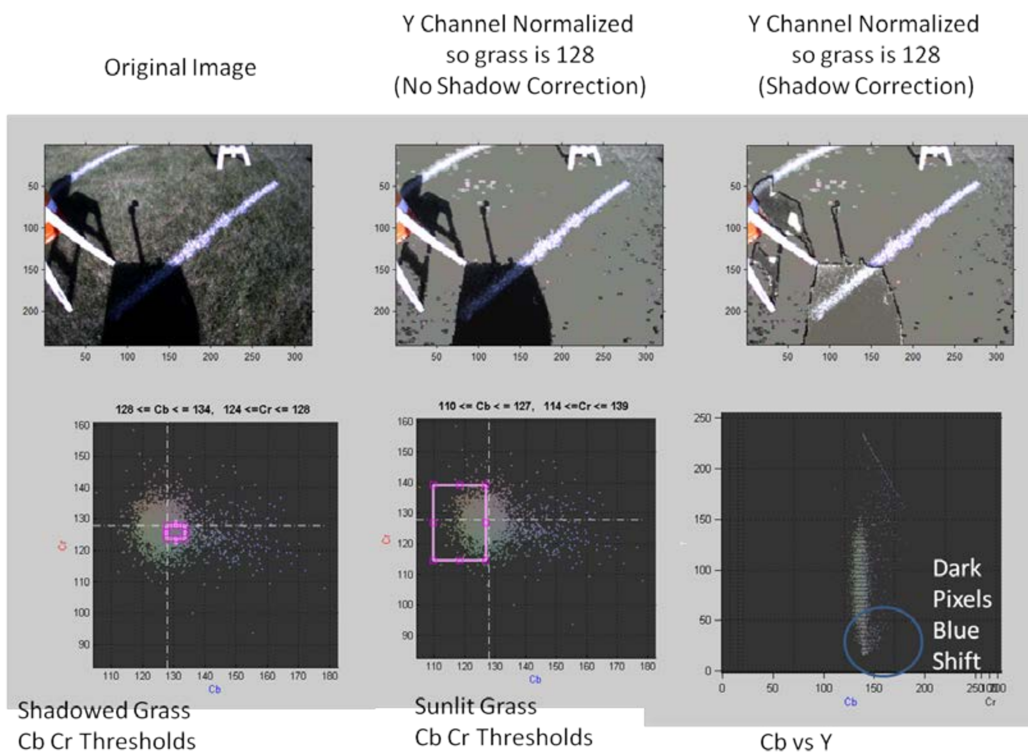
When pixels in each frame of the streaming camera fit the thresholded criteria of a "lane line" - when our robot sees what it believes to be a white lane line - it assigns a trend line to those pixels. This is a simple concept, the same process occurs in the brain when one sees a less-than-perfectly painted line on a field. However, the robot's line-fitting ability cannot distinguish the difference between white paint in the grass and the white stripes on the barrels. If they both fit the color-criteria for a lane line (white), then the program marks those pixels as "lane line" pixels. Then, when it calculates the trend line for the "lane line", the result is completely incorrect. The robot thinks that the lane line is pointed/angled in a certain direction, when any person looking at the field knows otherwise. Fortunately, there is a way around this problem through the use of structuring elements. A structuring element is a shape created around a pixel. For this particular task, we use a rectangular-shaped structuring element.

Orange pixels are dilated to overlap and conceal the white stripes

The result of this method is that the program is free to threshold and track the "actual" lane lines [in the grass], using the trend line calculations, since it is no longer confused by the white regions on the barrels. The following figure shows this dilation executed on the image of the course:

Figure 6

Calibration to ground plan with CalTech Toolbox: Using the CalTech Toolbox camera calibration software, we are able to determine the intrinsic and extrinsic parameters of the camera. This software allows the robot to know where the camera is, in reference to the ground. Since the camera is not stereoscopic, and without true depth perception, it calculates "depth" by assuming that all objects are flat on the ground.



Shadow compensation

Shadow Correction: Changing light conditions, both over time and even within the same image, are a constant challenge for outdoor vision. While the use of the YCbCr color space can help mitigate this, one defining feature of the while lines is that they have a higher Y component than the grass. We developed a new vision algorithm this year to address these issues.

1. Find all "green" pixels in image based on CbCr thresholds for grass.

2. Compute the average brightness (Y) of the grass. Rescale the image to make this brightness level the new median brightness (128) (Above figure top center)
3. Regions that are two standard deviations darker than this are considered shadows.
4. Apply a shadow correction technique inspired by Mark Ollis's 1995 Master's Thesis at Carnegie Mellon University. (Above figure Top Right)
   a. Rescale shadow regions to a median brightness of 128.
   b. Apply non-uniform color correction to compensate for sunlight vs. skylight illumination.
   c. Clean up halos and artifacts using morphology
5. Regions that are two standard deviations higher then background brightness, become candidate lane markings and are they subject to color, shape and size filters.

Obstacle Avoidance System

Laser: This year we added a newer, faster laser to the vehicle. Our new laser is the Hokuyo UBG-04 LX-F01. This new laser features a 0.38° resolution. The main upgrade with this laser is the scanning rate. The previous laser scanned 360° every 100 ms. The new laser scans 360° in 28 ms. Therefore we have decreased our refresh rate by nearly 4 times based solely on the laser allowing our program to run faster and more effectively. The laser measures phase difference and time of flight to determine the range to an object.

Our algorithm is patterned off of the Dynamic Window Obstacle Avoidance Method. After accounting for the robot's size and speed limitations, it finds the best "gap" or heading direction. "Best" is defined along the following criteria (1) closest to desired heading (specified by camera or GPS); (2) closest to current heading; and (3) maximum clearance.

The laser is capable of seeing up to 4 meters out; however, we set the maximum range that our navigation algorithm will recognize lower so we can navigate switchbacks. A switchback can be seen as three layers of obstacles. When the distance threshold of the laser is set properly, only one layer of obstacles can be seen at a time. This allows the RoboGoat to enter the first gap before trying to avoid the second layer of obstacles. If the distance threshold is set too high, the RoboGoat will see both the first and second layer at time, which will look like a solid wall, making the RoboGoat perform a u-turn.

The layering of the switchback obstacles is important because our algorithm that analyses the laser, camera, and ultrasonic data finds gaps in the obstacles. Lane data from the camera and returns from the ultrasonic sensors are combined with the laser data to form a complete picture of the obstacles around the RoboGoat. After finding the gaps, it compares the width of the gaps to the parameter that defines the width of the RoboGoat and determines if a gap is large enough. The code then finds the gap that is closest to the desired heading, which was defined using either GPS waypoints or lane information from the camera.

GPS Navigation System

GPS: The GPS receiver we use is a Trimble Ag 132. The receiver sensitivity is -185dBW and is accurate to within 1m using WAAS differential GPS. The receiver is attached to our laptop via a serial to USB adapter. The GPS receiver is mounted atop our mast to have a clear view of the satellites and is powered by the 12V battery through a 5V regulator. The receiver sends a NMEA (National Marine Electronics Association) sentence every .1 seconds to the serial object in MATLAB.

We display our position on a Google maps overlay written in Matlab. This visualization tool greatly improves our way points setting and debugging ability.

The RoboGoat does not use any form of mapping. In fact, the RoboGoat has no memory. In an attempt to achieve a 10Hz speed on our navigation loop and the reaction benefits that that brings, we decided to not map our environment. This, however, is making it difficult to keep ourselves from performing a u-turn when we encounter an obstacle or from crossing a line that has disappeared beyond our field of view. This was solved with the algoritm discussed earlier in this paper.

Drive System

Speed/Performance: The RoboGoat is hardware limited by its capabilities to 4.77 MPH. We also discovered that in order to maintain our minimum speed we have to, on average, send at least 6V to our motors.

Voltage (V)	RPM	MPH (10" wheel)
5.93	37	1.1
23.2	160.3	4.77

Roboteq: The RoboGoat is propelled by two 24V DC motors rated at 4.5A each. We control the motors with the Roboteq motor control board via commands from the laptop through a serial connection. We used the supplied RoboRun utility to measure the limits of our system by using a tachometer to measure our rotational velocity at varying voltage outputs. Since the motors are powered using two 12V lead acid batteries in series, we have an available voltage range of 24V. The RoboRun utility has a power setting that ranges from 0 to 127 and is proportional to the voltage range. This was the range used to test the speeds of the motors. The MATLAB function that sends commands to the RoboGoat, however, has translational and rotational inputs that range from 0 to 1. These are combined to form individual commands for each motor that correspond to the desired voltages. Forward and reverse changes are handled by changing which pin the Roboteq board uses to output a positive voltage to the motor with.

Wheel Chair: The chassis for the RoboGoat is a Sunfire Plus SP3C Power Mobility Chair. It weighs 82 lbs. The chair has a range of up to 20 miles, making it well suited to the competition. The chair is powered by two 12V U1 34 Ah batteries in series. The ground clearance on the chair is only 2.5 inches, meaning we can navigate small pot holes, but not large ones.

Safety Systems

Figure 8

Safety Light: A requirement for the IGVC is a safety light. The light must be solid when the vehicle is in ROV/manual mode and be flashing when it is in autonomous mode. The safety light we chose to use is a Federal Signal LP3TL-024R. We chose it for two reasons. First, it has an LED bulb that has a low current draw, .08A. The second reason we chose the light was its 4X weather rating, which means it is able to withstand dust, hose directed water, and corrosion. With a 100,000 hour life on the LED bulb, this light should remain useful to the project for many years.

Emergency Stop: The RoboGoat is paired with a 6CH Futaba transmitter and receiver. The Futaba transmitter manually controls the RoboGoat, has an Emergency Stop, and has the ability to switch the input of the Roboteq motor control board. A PIC12f675 is used to monitor CH5 which is the Emergency Stop channel. If tripped, the PIC disengages a relay that cuts power to a solenoid, which in turn cuts power to the motors running the RoboGoat. Another PIC12f675 monitors the other channels to manually control the RoboGoat and to change the input to the Roboteq board. The input for the Roboteq board can be set to the RS-232 input from the laptop running MATLAB or it can be set to receive input from the second PIC trough a MAX 323 which converts the signal from the transmitter to RS-232 levels.

Integration Testing

When considering our major systems, lane following, GPS, obstacle avoidance, our system is comparable to the 2012 design where we came in 2nd place. We believe our new additions are fully integrated with the old system and we can expect similar results. Specific testing points:

1. GPS Navigation: Our GPS did not change from the 2012 competition where we were able to hit all GPS points in the navigation course. We conclude that our GPS will have the same success this year.
2. Speed: The vehicle's max speed is 5 mph. However, at this time the autonomous navigation has only been tested at speeds up to 2 mph. Initial experiments with our laptop suggest an update rate of 10 Hz. We believe this will permit running the course at max speed.
3. Battery Life: At a recent test evolution the power systems were initially fully charged using AC power. Over the next 48 hours, we ran for about 4 hours using only the preexisting charge, and the energy contributed by the solar cells.
4. Obstacle Avoidance: Our obstacle avoidance algorithm is nearly flawless. It is perhaps the strongest feature of RoboGoat. At this time we intentionally use a detection distance of 1.5 meters, even though our hardware is capable of up to 4 meters.
5. Lane following: Over the course of 48 hours, we observe the vision system work nearly 100% of the time. More importantly we did not adjust the color thresholds despite the fact that the light conditions changed. This is extremely promising. Note it is very difficult for us to replicate the lane markings on campus because we cannot paint the grass

6. The rules state "…expect natural or artificial inclines with gradients not to exceed 15% and IGVC randomly placed obstacles along the course." We ae confident the robot can power itself up these inclines based on our inhouse ramp tests. More importantly we tipped the robot to its edge of stability and concluded that 15 degress does not pose a problem. With the new designed body in 2011-2012 the RobotGoat is capable of tipping up to 60 degrees without tipping over if it is not moving (see figure at right).
7. Flag combined with lane following: At this time we have not integrated the flag detection and path planning algoritm fully with the current lane following code. The difficulty of the funneling technique discussed earlier brought up issues of interfering with our lane following. Before the competition we plan to have this implemented and working in unison.

Cost Estimate

Part	Price
Chassis	$1,741
Weatherproofing	$80
Hokuyo UBG-04 LX-F01 Laser	$2,850
Roboteq Amp	$700
AgGPS 132	$1000
Compass	$40
Arduino UNO	$30
Emergency Light	$100
Solar Panel (2)	$280
Charge Controller	$60
Camera (1)	$670
Laptop and Software	$2,200
Google Nexus Tablet	$200
Total Price	**$9,951**