Team Seven Paper

Overall Strategy

The robot design centered around capturing red balls, elevating them to a height above the walls, and then opening a gate to release the balls into a field goal, in order to score the maximum number of points.

Mechanical Design and Sensors

Overall Frame and Powertrain

Roller

Our initial design for a ball collecting mechanism was some sort of paddle wheel, inspired by several of the teams' designs from last year. After getting a complete concept sketch down on paper for our entire ball collecting/lifting/dumping system, the team came to the conclusion that, in order for our ball lifting mechanism to work properly, our collection device would need to impart enough energy to the balls to reliably move them to the rear of our bot, where the entrance for the ball lifting device was to be positioned (in order to ease camera and sensor placement). We then decided that a paddle wheel or combine-type mechanism wouldn't be able to be spun fast enough to impart the necessary energy to the balls. It was then decided that a foam or rubber roller would be more feasable, as it allowed us to spin it much faster, without worrying about stalling the collector on a ball positioned in an inopportune spot (such as directly under the outer radius of a sweeping paddle). In looking for materials in lab, our first plan was to use a 3/4" dowel, with a section of gray foam pipe insulation glued onto the dowel in order to provide a higher-friction surface to grasp the ball. After searching through Mike's bag of motors, though, we were unable to come up with a viable solution for easily driving a 3/4" shaft, as would be required with the pipe insulation-and-dowel idea. On a trip to Home Depot, however, when looking for glue, we came across a foam paint roller which looked like it would fit our needs quite well, as it had a high friction surface, was the proper diameter, and fit on a 1/4" shaft, something we could easily drive. The final roller was made with this paint roller supported by a 1/4" aluminum shaft riding in two wood blocks attached to the support blocks for the front casters. A power seat motor from a Plymouth was used to drive the roller via a 1:1 chain drive. The motor was run at 6V.

Elevator

The design for the ball lifting mechanism was relatively unchanged from our original design. The idea was to use a rubber belt driven by two pulleys, attached to which would be wire or plastic paddles to push the balls up to a hopper through an aluminum channel. The only changes made to the concept were the following: First, the drive motor was

moved to the top pulley, in order to free up chassis space. Second, a separate support structure was made to support the pulley shafts, instead of having the upper pulley shaft supported by the hopper deck, which would then be, in turn, supported by support struts. This was done due to the relatively high forces placed on the upper pulley shaft by the belt tension. Third, our initial plan was to have a toothed timing belt, or a notched V-belt as the paddle mounting surface, but we switched to a section of 1/2" rubber fishtank air hose after discovering that the friction in the system caused by a notched V-belt was much, much too high for our motor to turn the shaft against. As was initially planned, the paddles were made out of steel wire, and the guide channel was fashioned out of thin sheet aluminum. The assembly included a threaded-rod-type belt tensioner, and was driven by a geared motor found in Mike's bag-O-motors.

Ball Gate

The ball release mechanism in the hopper was the simplist of our mechanical systems. It consisted of a wood flap zip-tied and hot glued to the output arm of a servo. Tape was used to bridge the gap between the hopper deck and door when the door was open. Cut up Shaw's cards were taped to the sides and front of the door to provide a guide for the balls as they rolled into the field goal, as well as provide an extension to the ramp, in order to allow our bot to dump the balls at a distance far enough away from the wall so that the IR sensor on the bot's front end would still get a valid reading. Bolts were taped to the inside of the ramp to agitate the balls, so that no two would get stuck against each other in trying to exit the hopper, thus preventing more points from being scored.

IR Scanning Head

Our two primary sensors were two short-range IR sensors mounted at 45 degrees to each other on a rotating head which was actuated by a servo mounted to the underside of the hopper deck. By havign two separate sensors, we were able to constantly be getting data from two side of the robot simultaneously, regardless of what angle the sensor head was at. The alleviated a previous problem, in which the robot was blind on one side when wall following, thus limiting the algorithm's robustness.

Wheel Encoders

The standard wheel encoders were used, with the 18-delineation encoder disks. Due to our inverted positioning of the stock motor mounts, however, the encoder circuit boards had to be moved 90 degrees around the motor output shaft, so that the encoders were now on the "front" side of the motor mounts, as opposed to the "bottom" (as they would be in our configuration), or the "top" (as they would be in the stock configuration).

Software Design

Vision

Mike began developing vision software during winter break. During that time, he wrote a

`V4LCap` class which captured images from any Video4Linux device, using the Java Media Framework to speak to the webcam. (Ultimately, the `V4LCap` class was used

only for initial testing and not during IAP.) He also found and decided to use ImageJ, a public-domain image processing library, for making it easier to handle different image sources. This allowed us to take advantage of many library routines already written and optimized in Java. Our final design has several classes that dealt with computer vision and image processing.

- The `CaptureThread` class is responsible for grabbing frames from the webcam as fast as possible. It contains two image buffers and captures to one while the other is being read and processed. It was important to have the call to maslab.camera.Camera.capture() be inside a thread because it was a blocking call and would otherwise prevent the rest of the program from doing anything interesting.
- The `VisionThread` class polls the `CaptureThread` and then performs all necessary processing of the image data.

- The `MachineVisionProcessorIJ` class is a holder for an ImageJ ColorProcessor and contains lots of our image processing functions. It also holds three byte arrays, `byte h], s[v[]` which contain hue/saturation/value data for every pixel in the image.
- The `HSVColor` class represents a color in HSV space. There were several pre-defined colors inside this class: white, yellow, green, black, red, blue, grey, violet, cyan. This allowed for an easy HSVColor.red.equals() mechanism to test for the presence of a certain color. There was also an HSVColor.toColor() function that returned a standard Java AWT Color object representing this color.
- The `Contour` class dealt with int[] arrays that contained one value for each column of the image. These contours were initialized to -1 in all elements, and then functions would operate on an image and identify the top of the blue line (not too useful), the bottom of the blue line (much more useful), and the break between floor and wall.

As noted above, the `VisionThread` controls the per-frame processing of images. When a new frame is taken, its order of operations is as follows:

- Downsample to 32x24 (from capture resolution of 160x120)
- Run `MachineVisionProcessorIJ.basicColorFilter1()`, which classifies each pixel as one of the basic colors. Having this code working over break was very useful. It uses a series of if statements to determine the color of the pixel. First, white is detected by a low saturation (less than 70 of 255) and high value (above whiteMinValue, which is auto-calibrated at run-time to better distinguish between wall and carpet). Black is detected by low value (< 80). Of pixels that remain, those that are considered not brightly colored enough (s<64 or v<90) will be

turned grey. Any remaining pixels are sorted by hue. There are then two extra comparisons that make extra demands on pixels deemed red or yellow, particularly requiring a higher value or saturation. One extra comparison is then made that turns any whiteish-blue (carpet) into grey.
- Identify the Contour of the bottom of the blue line. This searches for blue -> other transitions down a column.
- Do blue-line filtering, filling in the image with blue, for any points above the contour just determined.
- Find a floor-wall contour, hopefully because the carpet will be mostly grey and the wall will be mostly white.
- Runs `MachineVisionProcessorIJ.findRedBall1()`, which determines navigation toward a ball using the lowest pixel of red found and calculates a turn direction.
- Runs `MachineVisionProcessorIJ.findGoal1()`, which determines if a goal is visible and which way to steer toward it.
- Publishes every 10th frame to the BotClient.

Odometry

The robot keeps track of its current location from its start, where X,Y, and Theta are all 0. The bot knows how far it has travelled by averaging the changes in the readings from the left and right encoders. The angle of the bot, theta, is determined by calculating the quotient of the difference between the change in the right encoder and the change in the left encoder and the distance between each encoder. It then uses to trigonometry to get the absolute location, with the starting point as reference.

Mapping

The robot draws sensor data onto two maps of the Mapper class to store information about its surroundings.

- First, there is a global map, where the robot position is moved around the map with its absolute X,Y, and theta. In the global map, both vision data to calculate the distance to a wall by its apparent height, knowing its actual height, and IR distance data are used to determine the location of "wall" points. The global map has 10 cm resolution.
- A local map stores data only from the IR sensors and is always centered around the bot. As the bot moves, local map data that would be off the map is discarded. The rational behind this is that we shouldn't use data from large distances away because odometry errors have accumulated since the bot moved from the distant location. The local map has 2 cm resolution.

Collision Avoidance

To avoid colliding into walls, the robot has a CollisionAvoidance class, which implements a series of "virtual sensors." These sensors will look around different

directions for "walls" in the local map to determine if an obstacle is in the robot's way before moving in that direction.

Mechanical Control

A few classes were written just to ease control of various robot actuators:

- The `BallGate` class controls the servo on the hopper. `BallGate.open()` opens the gate so that balls can flow, while =BallGate.close() closes it.
- The `Roller` class controls the OrcBoard's pin 3 as a digital out. This pin is connected to a 2N7000 which in turn drives the coil of a relay. When pin 3 is set high and the relay connection is closed, the roller motor is activated. We found that the OrcBoard would sometimes "lose" the pin mode. To deal with this, the `Roller` class actually contained a thread that frequently overwrote both the pin mode and the value of the corresponding pin.
- The `Elevator` class was very similar to the `Roller`. It uses the OrcBoard's pins 4 and 5 as digital outs to control Mike's magical motor controllers. Pin 5 controls whether or not the elevator is on, while pin 4 controls the direction. The `Elevator` class implements a thread that forces the elevator to go forward for 9 seconds and reverse for one, in order to reduce the likelihood of the elevator getting stuck. As does the `Roller` , the `Elevator` frequently re-writes its OrcBoard pin values.

Overall Control

The robot's central control class creates threads such as the IR/Servo SensorStalk, the Elevator controller, the CollisionAvoidance scanner, the Mapper, and the Odometry updater. These threads run as fast as possible so that their variables are as consistant with actual conditions as possible. Whenever the central control class switches into a state that requires movement, it will poll some combination of the threads' variables to see if it is possible to turn a certain direction, if the robot is getting too close to a wall, or if a goal is in sight.

Overall Performance

The robot performed its scanning and ball-capture abilities admirably, catching a total of 9 balls. Unfortunately, the computer crashed within the last fifteen seconds of the round and all powered sources stayed in their "on positions." The robot got stuck to a wall, as the function to determine stalls and to back up was disabled when the computer turned off. The robot was unable to find a goal within the alloted time to deposit the balls that it had captured.

Conclusions/Suggestions for future teams

- Goal docking is quite difficult. For a greater degree of success, start thinking of an implementation that allows for good goal searching/docking before you design

your bot and optimize your mechanical design for goal docking, not just for ball finding.

- Think twice before doing MASLab during IAP AND the following activities: Taking 2.670, participating in a rocket team engine creation contest, running IAP workweek, and teaching a class for SIPB in PHP. It does require a lot of time. Running out of term to implement exactly what you want, with machined parts and a well-tested finite state machine is quite distressing.
