Team Eleven Paper

Final Report

The purpose of the following Final Report for Maslab 2005 is to highlight our successes and failures while providing some insight into our design decisions.

Mechanical Design

Woody (our robot), wasn't actually Woody until his second revision. Before his death, and subsequent reincarnation as a wooden robot - Woody was created from Polycarbonate, sheet metal, and steel bolts.

Revision 1 The initial chassis was basically a working prototype modified as the team went along. The team decided that having a working robot at as long as possible was important for writing code that worked - so always having at least a drivable chassis was paramount throughout Maslab.

The ball pickup mechanism was something that was debated at length, and the team finally decided to have a a belt driven system consisting of one aluminum dowel connect to a drive motor (both the Maslab provided "high speed", and "high torque" motors were tested, but the "high torque" motor was less torque than a drive motor, and the high speed wasn't really that much faster).

The underbody of the robot was build around the ball pickup mechanism. it was intented to be a detachable piece from the ball pickup mechanism - so that the two could be tested independently. This goal was achieved - but possibly could have been avoided if we had chosen a less error prone ball pickup design that required less debugging from the start.

Revision 2 (Woody)

Rev 2 took much less time than Rev 1, and although it was originally intended to fix all of the problems with Rev 1, because of time constraints it mostly improved upon the speed and manuverability of Woody.

The second revision was also made out of wood and sheet metal, rather than the polycarbonate of the first revision. This allowed for a faster construction time (the lab was much better for working with wood than polycarbonate) but was less "cool" looking.

Control System

The overall control system structure for our robot was of particular importance to our team. Our initial aims were to minimize repitition of code, allow for complex behavior, and to have the simplest interfaces possible. In our design all field features of concern to the robot are represented abstractly as the same Java type: the Feature. All Features have an integer-based (power of 2) type (e.g. NEAREST_BALL=1, NEAREST_GOAL=2, BARCODE=4...), an angle relative to the robot at their moment of discovery, and a distance. This type of generalization made navigation very simple -- whether it was a ball, or a goal, or a barcode, our robot could use the same feedback loop to "drive for Feature". An equally great benefit is afforded by the use of integers to specify type. This allows Behaviors to specify which types of features they care about, simplifying their design and minimizing the work involved with extracting the Features for further analysis.

Features are created by FeatureProviders (each of which runs in its own thread) that sample and analyze data as fast and often as possible. The three FeatureProviders we used were ImageFeatureProvider, IRFeatureProvider, and CurrentSpikeFeatureProvider.

- ImageFeatureProvider creates Features that corresponded to field elements within the view of the camera (balls, goals, and barcodes).
- IRFeatureProvider creates Features corresponding to distances measured by the 4 80cm range-finders
- CurrentSpikeFeatureProvider creates a unique BERSERK feature, which suggested that the robot's motors were stuck, and berserk behavior was appropriate.

The Brain, as we so aptly named it, requests an updated Feature list from each registered FeatureProvider and places them in a hashmap of lists. This allows different feature providers to both provide the same types of features with no effect on later computation. The Brain then creates another hashmap of lists for the registered Behaviors, and populates it with the types of Features each Behavior includes in their integer feature-mask. Once this map is created, each Behavior then does some degree of computation on the Features (or lack thereof) and returns a suggested Action. These Actions each have a priority and an execution method associated with them. Once all the suggested Actions are gathered and sorted by priority, the highest ranking one is executed. Actions also provide some degree of state to their execution methods, allowing for emergent characteristics that would otherwise be difficult to accomplish with this type of constant re-election. It is this maintenance of state that allows the Action returned by the WanderBehavior to select a random direction and stick with it until losing an election to some other Action of higher priority.

While this abstraction took some time to get used to, its architecture made it very easy to build a very robust and complex system from a minimal set of rules. For example, when attempting to score a field goal we find that aligning and releasing the balls are two distinct yet highly interrelated tasks. We don't want to release them if we aren't aligned, but on the other hand we don't want to force a realignment when it isn't necessary. We therefore place all scoring-related Actions within the same Behavior, and use the maintenance of state to guarantee only valid tranisitions can occur. Now, the robot will only release the balls if it has just finished chasing or realigning with a goal. In this respect our system is analogous to a collection of finite-state-machines. Well actually, a collection of flexibly-ranked, mutating finite-state machines, any of whom have the potential to jump to the foreground, and all of whom can safely ignore the existence and quantity of the others.

Image Processing

One of the areas that that we especially concentrated efforts at the beginning was in testing and optimising image processing techniques. It was important to the goals of MASlab that our algorithms were both reliable at locating and identifying objects as well as fast, to allow for faster computation and thus faster movement of the robot itself.

MASlab assigns a unique colour to each component of the field in the tournament. The carpet is light blue, the walls are white with a blue strip at the top, goals are bordered on the left/top/bottom with yellow and all balls are red. This eliminates the need to use advanced processing techniques such as shape recognition and high-quality edge detection.

One possible algorithm is to convert every pixel value from the raw RGB values provided by the camera into HSV (hue, saturation, value) coordinates, and filter each pixel by its hue and apply algorithms to find large filled areas. The advantage to this approach is the ability to locate every possible instance of a particular object and decide upon the most likely candidate. For example, one could find all red distinct regions, mark them as balls, and then remove any that are too small (noise) or too large (possible temporary camera white balancing issues). While an implementation of this type of "complete" algorithm. in C/C++ or assembly would be fast enough to be feasible at the camera resolution of 160x120, and could be interfaced with Java using JNI, we took a less computationally intensive approach that did not require us to use faster languages.

A native function was already provided as part of the MASlab API to convert from RGB to HSV coordinates. Assuming this is fast, our Java approach to image processing involved subsampling the image from the bottom up, at different levels of subsampling at each row. Given the camera's field of view and the size of a ball, it is possible to determine the diameter of a ball on the screen, n, in pixels. When searching for a ball for the first time, the algorithm then subsamples by checking every n/2 pixels at every other row from the bottom up. Thus it is guaranteed to have at least 1 sampled pixel in the ball for each row sampled. Once a row returns a hit (i.e. a red pixel is found), the algorithm then proceeds to check if at least 4 of the neigboring pixels are close enough to red to regard the pixel as a legitamate find.

The algorithm then called a bounding function to find the smallest rectangle that enclose all pixels, starting from the found red pixel, that are close to red. This was done by creating an initial rectangle of 3x3 pixels surrounding the initial pixel, and then expanding the box upward, one pixel at a time, until close to no red pixels exist in the topmost row. This was then repeated for the left, right, and bottom edges. While a fill function could have been written, much as most graphics editors have as a tool for the user, this is much slower than simply finding a bounding box using the previously described method. The bounding box algorithm described is not generally perfect, although it should work with extremely low error if the object is a simple shape (such as a ball) and the image quality is good.

The advantages to such an algorithm are mainly in computational speed. Our algorithm was able to achieve almost 20 frames per second on the 733 MHz Eden board and Logitech Quickcam 3000 provided by MASlab to locate the nearest ball, goal, and identify one or two on the screen; the speed limitation actually mainly came from the camera capture speed! A second advantage is that one can always locate the closest ball or closest goal to the robot by starting at the bottom of the image using this method. Thus, the robot can pursue the ball without having to scan any other part of the image. We also experimented with tracking techniques by searching for the same ball in subsequent images by starting by the ball coordinates in the previous frame and spiraling outward in the search for red pixels; this was extremely successful at first but became problematic if the ball ever disappeared from view due to collisions or otherwise.

This algorithm also eliminated the need to do an exhaustive top-line filtering; by searching bottom-up one can simply terminate the search after enough top-wall-coloured pixels are seen (about 10 to 15 should be sufficient). Thus, the entire top wall is not searched, and with our camera mount there was never a way for any ball to be visible above the height of a top wall blue pixel. This also improved the algorithm speed.

Goals were located using a very similar procedure; however, due to the complex shape it was necessary to begin at the bottom as usual, then search left and right to bound the edges of the vertical portion reliably. Then it would search updward, left and right, and then downward, corresponding to the typical geometrical description of a goal if one were to follow the yellow pixels from the bottom of one verical segment.

Barcodes were located by searching for green pixels, first bounding left/right with green, and then up/down with black or green. Barcode reading was fairly straightforward from this point; it would also return -1 as the barcode's value if it was not confident of the pixels read or a red pixel was found nearby (corresponding to a ball that may be blocking part of the barcode). We also intended to ensure that there was a top wall above the barcode and floor below the barcode, but since our final code did not utilise barcodes, we did not finish this section. Our barcode-reading algorithm was nevertheless fairly reliable.

Summary and Final Competition

Good:

- Scored 7 possession points
- Arguably had the greatest coverage of the playing field (Explored the entire field).
- Won the Maslab Engineering Award

Bad:

- Ball pickup became stuck on the last two balls (could have scored more possesion points but 2 balls at once were caught in the belt drive).
- Robot did not find a goal and attempt to score until the last 7 seconds

On the contest day, Woody lived up to most of his expectations with the only dissapointment being in not finding a goal to score into until the last few seconds left in the game. Possible solutions could be

- After 2 minutes - do nothing but try to score. Go into ball ignoring mode (make the ball pickup priority very low, and wandering / wall-following priority very high)
- Have a simple laser sensor that keeps track of how many balls are in the pickup - and goes into the same "score-or-else mode" after a certain number of balls (E.G. 3 balls).

Something worth noting is that Woody seemed like a crowd favorite. With the Linux Festival Program based Speech Server working flawlessly, all of Woody's debugging output was translated into spoken words picked from a library generated by the team. This made Woody's actions both humorous to observe - and more understandable to the crowd. Throughout the software development, audio debugging output helped greatly while testing code - and did the same on competition day.