Good Wheel Hunting: UMass Lowell's Scavenger Hunt Robot System

**Robert Casey, Andrew Chanler, Munjal Desai, Brenden Keyes,
Philip Thoren, Michael Baker, and Holly A. Yanco**

Computer Science Department, University of Massachusetts Lowell
One University Avenue, Olsen Hall
Lowell, MA 01854
{rcasey, achanler, mdesai, bkeyes, pthoren, mbaker, holly}@cs.uml.edu

Abstract

This paper describes the UMass Lowell entry into the Scavenger Hunt Competition at the AAAI-2005 Robot Competition and Exhibition. The scavenger hunt entry was built on top of the system we have been developing for urban search and rescue (USAR) research. The system includes new behaviors and behavior sequencing, vision algorithms and sensor processing algorithms, all used to locate the objects in the scavenger hunt.

INTRODUCTION

The AAAI-2005 Scavenger Hunt required a unique combination of artificial intelligence, human-robot interaction (HRI) and computer vision techniques. The scavenger hunt was a judged competition where robots autonomously searched for a predefined checklist of objects in a dynamically changing area crowded with people. The objects ranged in difficulty from simple (e.g., a bright yellow beach ball) to complex (e.g., a multicolored soccer ball and a plush dinosaur doll). Many of the objects shared colors with objects in the environment as well as each other, making it more difficult to accurately identify the objects. To increase the difficulty of the task, teams could also sequence behaviors to find multiple objects in succession without human intervention.

Our scavenger hunt entry drew upon several research projects from our lab. The robot has been developed for research in remote operations such as urban search and rescue (USAR), covering topics such as human-robot interaction, robust robot control and real-time vision processing. The competition provided an opportunity to build these projects into a single system.

ROBOT HARDWARE

Our system's platform is an iRobot ATRV-Jr, a rather large, four-wheeled all-terrain research platform initially equipped with a full sonar ring (26 sonar sensors), a SICK laser rangefinder, pan/tilt/zoom camera, and a full Linux (kernel 2.2) box running on an Intel Pentium 3 processor. Since purchasing this robot over three years ago, we have heavily modified its hardware and software for our research projects.

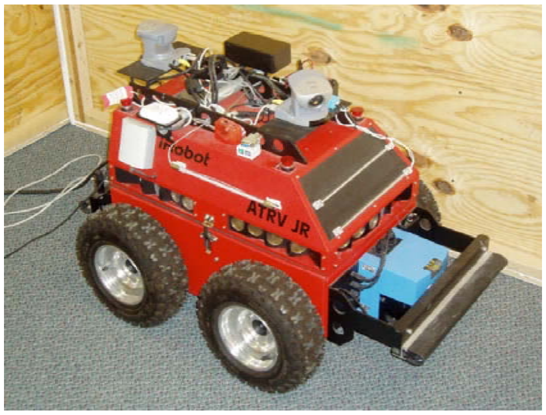

Figure 1: Customized iRobot ATRV-JR

One of our research projects is the development of effective human-robot interaction for remote robot operation. While studying a large number of interfaces developed for the AAAI and RoboCup USAR competitions, we noted that a large percentage of the robot hits in the environment were directly behind the robot [Yanco and Drury 2004]. The reason for this problem was clear; anything outside of the 180° camera pan range directly in front of the robot was essentially a blind spot. To alleviate this problem, we added a rear-facing pan/tilt/zoom camera to our robot. This camera (a Canon VCC4) is identical to the forward-facing camera that was already in place. In user tests using the two cameras, we noticed a significant reduction in the number of times users drove into obstacles behind them.

The next, and possibly most significant change we made to the system itself was to fully upgrade the system's on-board computer. While studying previous competitions, the need for more advanced vision processing became evident.

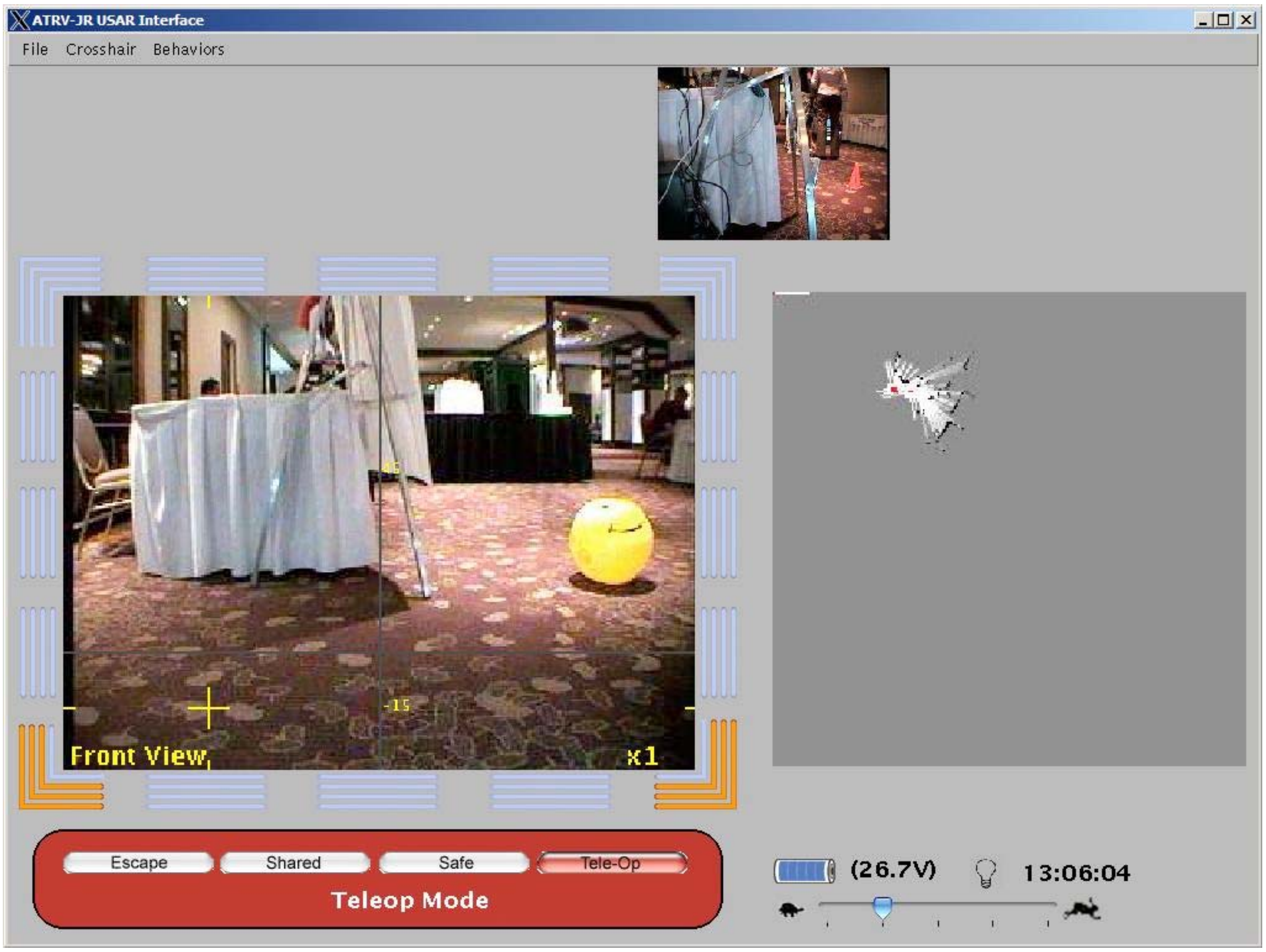

Figure 2: The interface developed for remote robot operations, described fully in [Baker et al. 2004]. This interface was modified for use in the scavenger hunt competition.

To alleviate the computation requirements of these advanced algorithms, we replaced the Pentium 3 system with a Pentium 4 system with one gigabyte of RAM. For this upgrade, the robot required a new power supply and updating of all of its software, including iRobot's Mobility software.

INTERFACE OVERVIEW

The interface used in the competition was developed in our lab, initially for USAR applications [Baker et al. 2004]. Our design was influences by studying over twelve USAR robot systems over the past three years at AAAI and RoboCup competitions. The interface was designed to improve situation awareness. We have observed that the main area of focus for all users is the primary video screen to the exclusion of all other useful status information. As a result, many of the elements are displayed in or around the video window. For example, we overlay a crosshair to indicate the pan and tilt orientation of the primary video camera. We also include a video stream from the back camera that is designed to mimic the look and feel of a car's rear-view mirror. The ranging information is situated around the main video screen, so that users do not need to look far from their main focus to see if they are close to obstacles that may not be in the camera's view. A dynamically generated map is also placed directly to the right of the main video screen. For more details about the design of the interface, see [Baker et al. 2004].

The interface communicates wirelessly with the robot using the User Datagram Protocol (UDP). The robot is constantly broadcasting its status information, including ranging information, video feeds, and battery level. These updates are sent both on a regular basis as well as after receiving a command from the interface. The interface updates its displays only after acquiring new status data.

We do not use the Transmission Control Protocol (TCP) in this system for two main reasons. When using TCP, if communications are lost or messages are sent at a greater rate than the back end can handle, the commands will queue up. We do not want queuing because when communication resumes all those commands will be sent to the robot, which could cause it to behave erratically for a period of time until all the queued messages are dealt with. This type of operation could cause damage to the robot and environment, as well as put humans at risk. In these cases it is safer to drop packets rather than queue them up. Also, we do not use TCP because in a congested network or with a poor wireless signal the bandwidth will be throttled down to try to relieve some of the congestion. However, we always want the highest possible bandwidth for the interface/robot communications, which allows for more effective control.

For this competition, the only modification we made to the interface was the addition of a "Behaviors" menu, showing in figure 5. This menu is dynamically created from a list of behaviors sent by the robot when the interface is started. This menu included behaviors such as *Follow Green Trail*, *Track Soccer Ball*, and *Track Yellow Ball*. Using this menu, the user can choose to activate/deactivate as well as queue up any number of behaviors to be performed in sequence. The actual details of the behavior architecture are discussed below.

NAVIGATION

For the scavenger hunt, our robot was required to navigate safely in a conference hall setting in the presence of moving objects and/or people. Since the rules stipulated that AI techniques must be used in each entry, we developed autonomous navigation behaviors that could be used to move the robot safely around a semi-structured indoor environment independent of any higher-level task.

In our system, a mode is a collection of concurrent and/or sequenced behaviors that accomplish some high-level task. A behavior is a lower-level response to environmental stimuli such as obstacles and colors. This section describes the mode/behavior system we designed for safe autonomous robot navigation at the competition.

Navigation Behaviors

Each behavior has a weight that specifies its contribution to the overall behavior. That is, the actual behavior (drive command) executed by the system is a blending (weighted sum) of all active behaviors. All behaviors can be activated and deactivated dynamically as the task or situation dictates. It is also possible for a behavior to suppress or partially suppress one or more other behaviors. Using weights and suppression, it is possible to achieve very sophisticated behaviors using very simple building-block behaviors. Below, we describe the behaviors in our system.

Joystick Behavior. This behavior is the user's raw joystick command. The analog joystick command is turned into a drive command that varies in magnitude with how far the joystick is pushed or pulled.

Spin Behavior. This behavior causes the robot to spin or rotate in place. We created this behavior to search for an object, such as the yellow beach ball, 360 degrees around the robot. Since our cameras can not pan more than 180 degrees, we decided this behavior would be the quickest and simplest way to locate an object in wide open space.

Stuck Behavior. Since our robot has a full sonar ring, we can detect obstacles equally well on all sides of the robot. This means that our robot can travel just as safely backwards as it can forwards. We take advantage of this ability in our stuck behavior. If the robot is navigating autonomously, and it has not made progress—determined by how much it has moved in set period of time—then the robot starts driving in the opposite direction. This approach has proven successful because it is highly unlikely that the robot could become blocked on the front and rear sides simultaneously.

Open Space Behavior. Using the full sonar ring the open space behavior implements the idea of a potential field by treating individual sonar readings as vectors emanating from the robot and computing the vector sum of all sonar readings. The resultant vector gives the direction and speed of the robot, which will be toward the most open space. This behavior is not very useful by itself because cancellation of vectors (vector sum equals zero) is possible. There are many possible orientations of the robot with respect to obstacles in the environment that cause cancellation of vectors, which prevents the robot from moving. However, when combined with other behaviors, this can be a helpful behavior.

Forward Behavior. This is a very simple behavior that causes the robot to drive straight ahead. As explained in open space behavior, the potential field method turned out to be unsatisfactory for moving the robot around in a hallway environment, so we added this behavior.

Stop Forward Behavior. This is another simple behavior that completely suppresses forward behavior when there is something very close to the front of the robot. Without this protective behavior, the combined forward tendency of the other behaviors could cause the robot to hit something in front of the robot. This behavior prevents front hits very effectively.

Avoid Front Wall Behavior. We created this behavior to deal with the situation where the robot is moving in open space and encounters a barricade or wall on the front. The robot will turn left or right depending on the angle of the robot with respect to the wall. If the robot is facing more to the left, the robot will turn left. Once the robot has

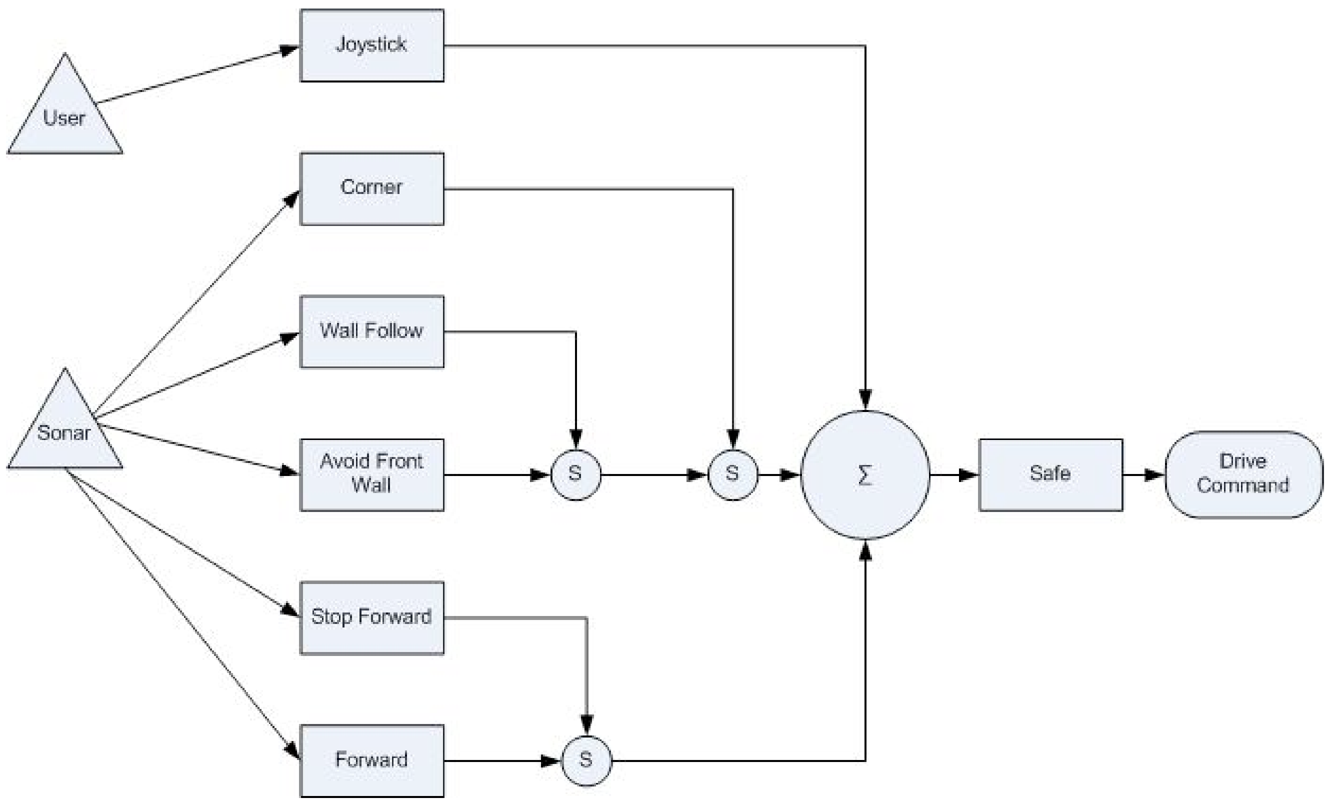

Figure.3: Shared mode achieves safe, autonomous navigation in a semi-structured indoor environment.

aligned itself parallel to the wall, the wall following behavior would kick in and take over.

Wall Follow Behavior. This behavior detects walls on either side of the robot and follows the closer wall in the case of a hallway. The robot maintains a specified distance parallel to the wall it is following by making steering corrections whenever it deviates from the specified distance. The robot makes steering corrections in proportion to the amount of deviation from the preferred distance. In this way, the corrections are generally very gentle and the robot maintains a fairly straight and constant path parallel to the wall.

Corner Behavior. This behavior detects when the robot is in a corner and guides the robot past the corner by turning the robot in the appropriate direction. Corner is used in conjunction with the wall follow behavior to achieve autonomous navigation in a hallway environment.

Safe Mode. Safe mode is really a speed governor mode in the current design of the system. Safe mode was created originally to allow the robot to protect itself when a user command would result in bumping into an obstacle. In safe mode, the robot detects nearby obstacles in the robot's path and stops the robot before a collision can occur. When the robot is navigating autonomously, however, stopping the robot would impede the robot's progress. Instead, we limit the robot's speed in proportion to its distance from the nearest obstacle in the direction of motion. So the robot slows, but continues to make gradual progress. The idea behind allowing the robot to make slow progress is that, eventually, some environmental feature (open space, a wall, a corner, etc.) will be detected and trigger a behavior that will move the robot away from the obstacles that are slowing it down.

Shared Mode: A Complete Mode/Behavior Example

This section describes a general navigation mode in our robot system: shared mode. Shared mode encompasses a true blending of robot and human control by combining user joystick commands with the other behaviors. The weight of the joystick behavior controls how much the user may override or influence the robot's autonomous control.

As shown in figure 3, shared mode takes user joystick commands and sonar readings as inputs. Each of the active behaviors that make up the shared mode can contribute a drive command, but one behavior can suppress one or more other behaviors depending on the desired overall

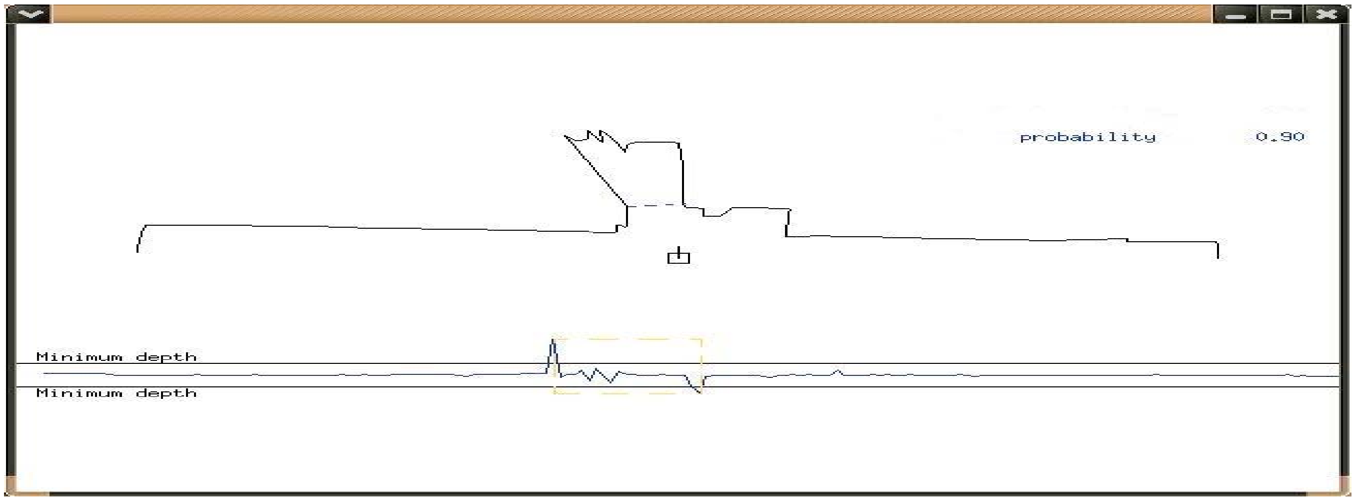

Figure 4: Demonstration of the door detection algorithm.

behavior. In the current example, the wall follow behavior can suppress the avoid front wall behavior, and the corner behavior can suppress wall follow and avoid front wall. The drive commands from the various behaviors are summed according to the weight of each behavior and the blended command is passed through a speed-limiting function in safe mode, which determines the command that actually moves the robot.

DOOR DETECTION

The door detection algorithm uses the laser ranging data to detect doors. The data from the laser range sensor is collected in the form of x,y coordinates rather than vectors to obstacles. The algorithm, on detecting an opening, calculates the probability that the opening is a door and indicates the location of the door relative to the robot. The algorithm is capable of detecting multiple doors at the same time.

A door is defined by three values: the minimum and maximum width, as well as the minimum depth of the door. The depth of a door can be defined as the space behind the door that must be free of objects. Thus, this algorithm can detect open doors only at this time.

The algorithm reads x,y values and records the coordinate value if the difference between the current and the last y coordinate is greater then the defined minimum depth. This is assumed to be the starting coordinate of the door, which we refer to as an upward spike. From this point on if another upward spike is observed then it automatically becomes the starting coordinate of the door, in other words the new upward spike.

While reading the values, if the difference between the current and the last y coordinate is less then the negative of the defined minimum depth and an upward spike has already been found, it is recorded as a downward spike. If the distance between the last recorded upward spike and the current downward spike satisfies the minimum and maximum width constraints, a door is detected.

Once a downward spike is found the algorithm resets the coordinates for the upward spike, and so the algorithm starts to look again for an upward spike. Once a door has been found, the algorithm calculates the probability of that being a door. The algorithm uses a simple but effective method to do so. The angle made by the perpendicular drawn from the center of the door towards origin and line segment from the mid-point of the door to the origin is found. Higher differences lead to lower probabilities for the detected door being an actual door.

Overall this is a fairly robust algorithm. During the tests performed in the hallway near our lab, doors were detected around 80% of the time. However, the algorithm performs poorly when there are many tables and chairs placed close to each other, as they were in the competition area. We are working on ways to improve this. One of the planned improvements is to make sure that there is a specified distance before and after the detected door with the same slope as that of the door, indicating walls on both sides.

In the figure 4, the small rectangle in the center of the screen is the robot and the vertical line indicates the forward direction. The lines around the robot indicate the environment surrounding the robot as seen by the laser range sensor. The dashed line seen in front of the robot is the door which has been detected. The two points where the minimum depth lines are crossed correspond to the start and the end of the door.

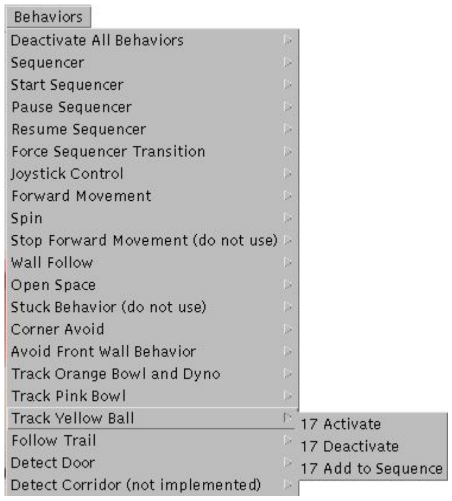

Figure 5: The behavior menu.

TASK SEQUENCER

Once we had the ability to complete various tasks in the scavenger hunt, we wanted a way to combine the tasks for the competition. Our solution, a behavior sequencer, allows us to queue up several tasks for the scavenger hunt. For example, we can queue up *Find Door*, *Follow Trail*, *Find Yellow Ball*, and *Find Pink Bowl*; the tasks will the be executed in that order.

Our interface supplies the functionality to start, stop, and pause the sequencer or even force a transition to the next task. The interface also allows us to manipulate the sequencer queue by adding, removing, and clearing tasks. The sequencer queue supports multiple instances of the same tasks, which allows us to run the same behavior multiple times in a row. With this we were able to follow a colored trail, find various colored objects (one at a time), and then follow the trail back home.

The sequencer gave us more flexibility during the judging of the system. We could ask the judges the order they would like to see our robot perform the tasks rather than having a hard coded predefined scavenger hunt, allowing the judges to challenge our system.

VISION SYSTEM

Our entry used a vision system called Phission, developed in our lab. Phission is a toolkit to assemble any number of multi-threaded subsystems for continuous vision data processing. There are modules for capturing, processing, and displaying video data. There is also a system module that provides a simplified interface for starting, pausing, and stopping all the threads that each module encapsulates. Any number of modules may exist within a system module and the only limitation lies within the capabilities of the hardware.

Using a C/C++ API, the output of one module is connected to the input of another module. A capture module output will be connected to the input of a processing pipeline module or a display module. The processing module also can be connected to a display module. The display module is mainly used for debugging and development of the processing system because of the resources required to display video data. Generally, the display will be disabled during a competition or live run when seeing the video is not necessary.

Phission is capable of processing video at very high frame rates because of the way in which the modules pass video frames to each other. While the pipeline thread is processing a video frame, the other threads continue to capture and display video frames. The capture module always has a copy of the most recent, fully captured video frame. When the processing thread is ready to process another image, it can retrieve the newest frame without having to wait for a capture cycle to complete. A display connected to the output of a processing thread will also be able to copy the most recent fully processed image as soon as it's ready. Phission was designed in this way to transfer data asynchronously in a passive manner. A source module does not push data to a destination module. In Phission, any given module is not likely to wait as long for the output from another module as compared to a non-threaded serially designed system.

The Phission pipeline module is a simple one input to one output processing thread. Filters are added into the pipeline and they use a workspace image as input and output. The workspace image is managed by the pipeline module. Its purpose is to remove the need for a filter to copy an input image and copy to an output image every time the filter is run. All filters have a common high level interface by inheriting from a filter class.

While Phission was built with video processing as its main data type, the system is extensible to any type of data through a parent object that provides for the live/continuous updating of a data object. The blob and the histogram filter both make use of that live object interface to store and propagate histogram and blob information. A control thread, usually the main loop of a program, can connect to a blob data object that is output from a specific blob filter and update the object when new data is available. In the case where new data is not available, the control loop can pass over the processing of the blob data until new data is available.

Phission is available as a library that can be built on Linux or Cygwin/Windows and linked with any C/C++ program. For languages such as Java or Python, the Phission API

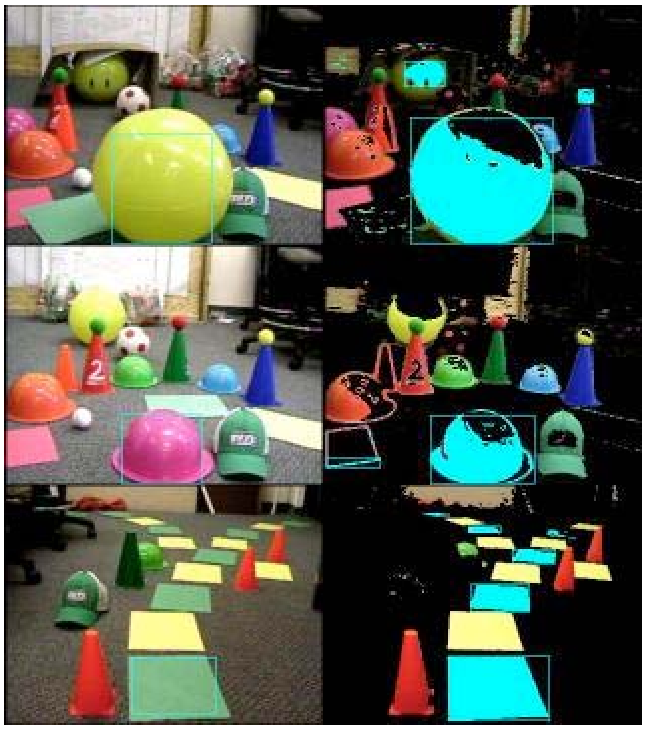
Figure 6: (a) Histogramming yellow ball and (b) HSV Threshold and blobbing yellow ball. (c) Histogramming the pink Bowl and (d) HSV Threshold and Blobbing of the pink bowl. (e) Histogramming a sheet of green paper on the floor and (f) HSV Threshold and blobbing of the green paper laid in a trail.

and facilities can be included through a loadable module that is installed on supported systems.

OBJECT RECOGNITION AND TRACKING

Most of the competition tasks involved finding various colored objects. We created a generic behavior capable of tracking a colored object and driving up to it. Specific object tracking behaviors (e.g., *Find Yellow Ball*, *Follow Green Trail*, etc.) are created with a set of saved color matching thresholds which are input to the generic object tracking behavior. The *Follow Green Trail* behavior is derived from the *Track Object* behavior to allow for a minor difference in control functionality. Where the *Track Object* behavior will stop when the object is directly in front of the robot, the *Follow Green Trail* behavior will continue driving while it sees another green marker that is large enough and below the horizon.

In order to achieve better vision processing performance, the capture module is started up when the robot back end program is loaded. A reference to the capture module is passed to each *Track Object* behavior upon creation. Whether the robot is running a behavior or not, the capture module will always be running to provide the very latest image immediately when the behavior is activated.

Training and segmentation on video is done using the HSV (Hue Saturation Value) color space. The HSV color space is better because it is less sensitive to lighting changes and colors are more easily matched. Matching a color in HSV space proves easier because similar colors (for example, all yellow objects) are located in a single linear range within the Hue field. In addition, removing bright and dark objects can be done more easily given HSV color data. This eliminates much of the video data that is useless.

The trained values for an object are found by using all the same hardware equipment and pre-processing filters (Gaussian, HSV conversion) and then running a histogram over an area of the image that contains the object. The histogram information output is an HSV color value and an upper and lower threshold value. A program is used to histogram on a live video stream as opposed to single frames loaded from files. In this manner, one can see how the blobbing differs from frame to frame and it also allows the placement of other objects within the field of view to determine the potential for accidentally matching other objects. The same program can take in an HSV threshold parameter file to permit live configuration of the HSV threshold value to remove much of the unwanted surrounding scenery. Examples of the images seen during training are shown in figure 6. The images on the left are the histogram input images. The images on the right are the images that are first HSV thresholded and then blobbed on the histogram color value outputs.

The object matching algorithm is rather simple at a high level. First the video frame is blurred with a 3x3 Gaussian filter to reduce image noise and inconsistency. Next, the image is converted to the HSV color space. After conversion, the HSV values are thresholded according to their Saturation and Value bytes to remove very bright or very dark pixels. Finally, the blob filter is run to segment HSV pixels that match the given trained values.

After all the processing of the video data, the location of the objects within the video frame can be retrieved and acted upon. Using an assumed horizon located at the middle of the image, the tracking behaviors will align on largest blob below the horizon until it is in the center of the camera's view by returning rotation motor values and no translational motor values. When the image is within the center of the image, the behaviors will return translational values to drive towards the object. The normal behavior is to stop when the object is reached, however, the follow trail behavior will continue to drive as long as there is a large blob that matches the trail object.

Since all this happens in conjunction with many other tasks and sensor processing, it is possible to poll on whether there is new blob information available. If there is no new blob information available, the previous rotational and translational values are returned and are not updated until new information is ready. This prevents the blocking of more important avoid behaviors and other general control routines. The vision subsystem aggregates the video data into a slower updating data set (blob coordinates and size information) which takes time to generate. Waiting on this data can adversely affect the response of other robot behaviors in the control system.

RESULTS

Our system was able to locate several of the scavenger hunt items, including the yellow beach ball, the pink bowl, the two-colored soccer ball and a trail of green paper. Work on distinguishing the pink and orange dinosaur from other pink and orange items was not completed in time for the competition.

The action sequencing developed at the competition allowed for the system to demonstrate all of its finding behaviors in an order specified by the judges.

The robot performed its tasks well, despite a last minute charging circuit failure, resulting in discharged batteries just a half hour before the time set for judging. Fortunately, we were able to bypass the circuit to allow the batteries to get some charge before the judges arrived.

Our system was awarded two technical awards: one for its object recognition capabilities and another for the usability of the control interface.

ACKNOWLEDGEMENTS

This work was supported in part by NSF IIS-0308186, NSF IIS-0415224 and NIST 70NANB3H1116. Travel support was provided by the sponsors of the AAAI Robot Competition and Competition.

REFERENCES

Michael Baker, Robert Casey, Brenden Keyes and Holly A. Yanco (2004). “Improved Interfaces for Human-Robot Interaction in Urban Search and Rescue.” *Proceedings of the IEEE Conference on Systems, Man and Cybernetics*, The Hague, Netherlands, October.

Holly A. Yanco and Jill Drury (2004). “‘Where Am I?’ Acquiring Situation Awareness Using a Remote Robot Platform.” *Proceedings of the IEEE Conference on Systems, Man and Cybernetics*, The Hague, Netherlands, October.