Multi-Touch Multi-Robot Interface

David Millman
millman@cs.stanford.edu

Abstract

The overarching theme of this project is to explore ways for a single user to control multiple robots using a multi-touch interface. In particular, how does increasing the number of robots change both the interaction methods and the control methods? This project aims to the answer the question of "What types of problems occur when controlling real robots, and how do you solve them?"

1 Introduction

Much like cell phones have become an everyday item in the last decade, robots are increasingly coming into mainstream use. Despite there being no standard interface for controlling a single robot, we are exploring the idea of how a single person can control multiple robots.

Imagine driving a car – your hands on the wheel. Now imagine driving two cars at the same time, with two steering wheels, two gas pedals, and two brakes. This is a difficult task. Much like driving two cars at the same time is tough; trying to drive two robots would be just as hard. Now what happens if you have three, four, or ten robots? It gets even worse if you consider that the robots may each have their own arms, too.

To remedy this single-operator-multiple-robots control problem, we must give the robots some level of autonomy. Once you decide that a robot will have some self-control, the next question is "how much?" Current thinking suggests that the level of autonomy should be able to slide from tele-operated (precisely remote controlled) to completely autonomous, e.g. "JoeBot, get me some coffee." (Baker & Yanco, 2004)

Multi-touch displays provide a natural way for one or more people to control a group of robots at higher levels of interactivity than a mouse. They have a combined interface and display, and are ideal for showing broad views of a scenario.

The Multi-touch Multi-robot project is an initial attempt at creating an integrated multi-robot solution. We expand on an existing early prototype from Stanford AI Lab.

2 System Infrastructure

Figure 1: Multi-touch Multi-robot Interface sample display

2.1 Previous Work

The Multi-touch Multi-robot Interface extends work done previously by people at the Stanford AI Lab (Ricciardi, 2009). The prototype could accept raw touch input and do simple processing on that input. It provided the foundation for displaying a map of the environment and the locations of the robots. There was also support for having a robot follow a user-drawn path.

2.2 Configuration Changes

The first step in advancing the state of the project was to upgrade the project to work with the latest version of ROS. It is important to leverage all of the fixes and functionality that have been and will

be added to ROS. Being tied to an older version would require implementing code that might already exist.

Launch files were added to the project to better compartmentalize setup and testing. Now, the ROS core (including map server), input/visualization, and robot controllers are all separate components. The overall system design is now as follows:

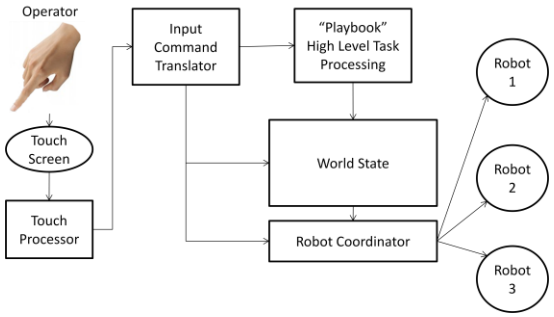

Figure 2: Multi-touch Multi-robot Interface Overview

2.3 Input Processing

The input system for the project has been updated to:

- Improve the responsiveness of the input
- Add more types of touch inputs
- Allow for complex input controls
- Allow for input controls that are dependent on the state of the interface

The input now follows a staged pipeline approach:

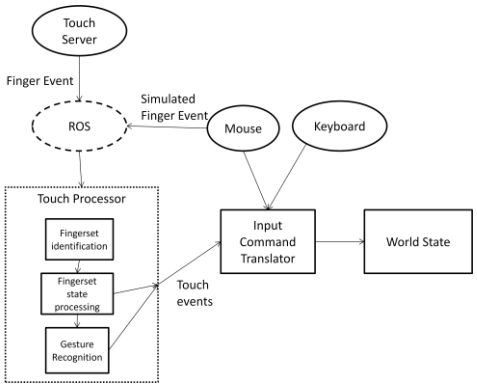

Figure 3: Input System Overview

Based on work done in user studies it is common for users to use multiple fingers when gesturing (Micire, Desai, Courtemanche, Tsui, & Yanco, 2009). For example, tapping on a touch screen can be used for selection. Without a mouse, it is natural to tap with one finger, or with a few fingers. To handle this behavior correctly, the touch processor has built-in support for multi-finger gestures (note that this is different from gesture recognition with multiple strokes).

Multi-finger gestures are handled by using a preprocessing step to join fingers into a group (a "fingerset") and then allow fingers to join or leave the fingerset. Conceptually, a fingerset most closely represents a few fingers on a person's hand.

Each fingerset emits events based on the state transitions:

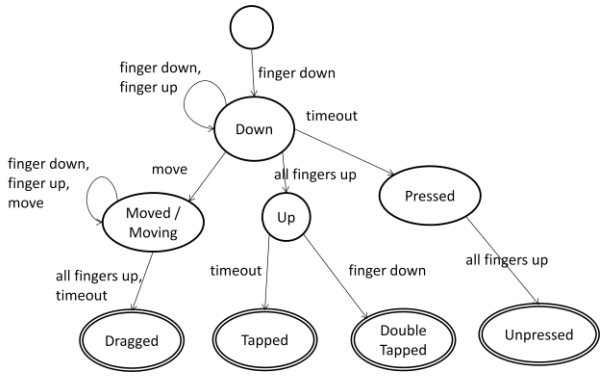

Figure 4: Touch Processing State Machine

The following assumptions are being made in with the current implementation:

- Once a finger is determined to be in a fingerset, it should not leave that set or join a different set. The thinking behind this is that a user will move all fingers in a gesture in the same manner.
- Association with a fingerset can be determined from the initial placement of the finger. In practice, this means that a finger identified near another set of fingers (based on some distance threshold) will join the fingerset. A finger far from another set of fingers will not join the fingerset, even if it later moves closer (even within the distance threshold)
- Association with a fingerset can be determined by distance. Even for people with very large hands, we assume that when fingers are intended to be used as a set, the

distance between each finger is relatively small.

Notice in the fingerset diagram, that state transitions are mostly unaffected by single-finger changes.

To improve responsiveness of the touch interface, the touch processor emits events at several states, not just terminal states. For example, as soon as a finger is pressed, an event is emitted. At the point of the finger press, it's impossible to know if the user intends to press and hold or press and release (i.e. tap). But, if the user presses in a location where the interface would respond the same to a press and hold or a tap, the interface can respond as soon as the press is detected instead of waiting for the entire tap (press and release).

Gesture recognition is handled as a final stage. After a fingerset terminates with a drag event, the dragged path is used as an input to the gesture recognizer. By the time the gesture recognizer processes the event, the touch processor will have already emitted "press" and "move" events corresponding to the dragged path. In the case where the gesture was the intended input, the other events will be ignored. And in the case where the other events were the intended input, there is no need to wait for gesture recognizer.

For testing purposes, the mouse can simulate a single-finger input. Pressing the mouse left button represents a finger press, and releasing the button represents releasing the finger.

2.4 Robot Control

The robot controller has been expanded to incorporate the idea of waypoints. Waypoints give a method for allowing a user to control the robots, but still have the robots act autonomously.

The implementation for waypoints is a plug-in for the ROS move_base module. The waypoints are treated as intermediate goals, with the last waypoint being the final goal. The ROS Navfn module is given a costmap and is used to plan a path between each pair of waypoints.

The granularity of waypoints can be used to adjust the desired autonomy of the robot. For example, giving only a single waypoint (i.e. goal) will cause the planner to find a best-cost path from the current location of the robot to the goal – a high autonomy situation. At the other extreme, inputting many waypoints, at the scale of one per each cell in the map, would effectively cause the planner to follow your exact path.

The current implementation of the robot controller supports 3 waypoint-following behaviors:

- Standard: The robot follows the planned path to the first waypoint. Then it follows the path through each waypoint until it reaches the goal. Then it stops.
- Repeat: Once the robot reaches the goal, it finds a path to the first waypoint and repeats. This can be used for having a robot move in a loop.
- Repeat in reverse: The robot follows the planned path through each waypoint and to the goal. Once the robot reaches the goal, it follows the same path in reverse. This can be used for having a robot patrol up and down a hallway.

3 Continuous Area Sweeping

High level tasking for robots can be thought of in terms of having a "playbook". That is, a pre-defined set of rules that a team of robots should follow. In this vein, we look at implementing Continuous Area Sweeping. Previous work (Ahmadi & Stone, 2006) has used the assumption that the robots will divide the map into "regions of responsibility". Since this will eventually be used in conjunction with human operators, this constraint is relaxed for more flexibility.

In Continuous Area Sweeping, the goal is to find a path through an environment such each area is "touched" (this might be touched in the literal sense if you were physically sweeping, or e.g. areas that we have observed when surveying). Continuous Area Sweeping assumes a dynamic environment, so the best path through an environment may be constantly changing.

Continuous Area Sweeping has connections to several complex behaviors including surveillance, exploration, search & rescue, and cleaning. We use "area patrolling" as a reference case for implementation. There is a team of robots that must organize themselves to continually inspect each area of a map.

Our initial implementation makes some simplifying assumptions regarding the problem:

- The robot team is homogenous – all robots are identical
- Each robot has a 360 degree field of view, i.e. can see in all directions
- The map can be discretized into a small set of cells
- The robot can move up, down, left, or right, and the movement is deterministic

3.1 Value Iteration

Patrolling is implemented using Value Iteration.

The reward value is based on the time since a location was visited and the other visible cells from that location. More precisely, we use the sum of the difference in last visit times:

$$R(s) = \sum_{v} now - LastVisit(v)$$

Where v is a cell visible from s.

For example, say we have a map and initialize all cells in the map to have the same last visit time. This would be analogous to a robot having just entered the map – no cells have been visited yet. Then, we place a robot at cell (0,0). The reward function would be updated based on the robot's line of sight (Figure 5).

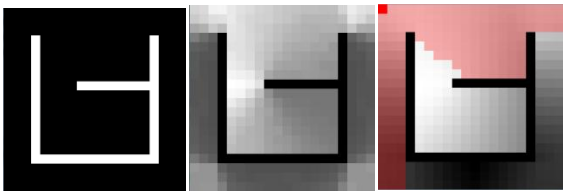

Figure 5: "Cave" map obstacles, reward function, value function (with robot line of sight). Ligher color indicates areas of higher value.

Running the algorithm on a series of maps produces intuitive behaviors. For the square corridor map the robots correctly find an optimal configuration. They can see all cells, and thus stop moving. In the cave map, one robot will patrol the bottom of the map, while the other patrols the top and occasionally peeks inside the cave.

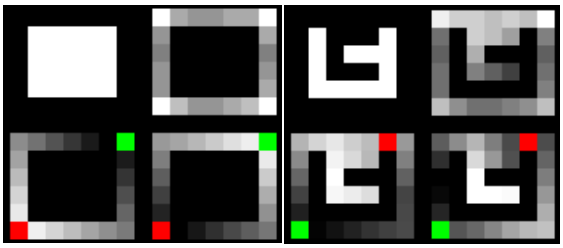

Figure 6: The top row shows the obstacle map and initial reward function for the "Square" and "Cave" maps. The bottom row shows the value function for the red and green robot respectively. Since the value function is 4 dimensional, the display shows all states for the red/green given a fixed position for the other robot.

3.2 Fitted Value Iteration

When adding more robots to patrol a map, the computational complexity increases dramatically. In our case, each additional robot increases the state space by two dimensions (x and y coordinate). Even in very small maps, this causes a problem, In light of this, we explored the use of Fitted Value Iteration (Ng, 2009) to sample the state space.

In evaluating Fitted Value Iteration, we visually inspected the result of using a number of different terrain features (Figure 7). Since the goal of Fitted Value Iteration is to approximate the value function, we used the discrete value iteration result of the cave map as a reference. An infinitely sampled Fitted Value Iteration function should match exactly. Due to the non-linearity of the data, our best results came from using a Locally Weighted Linear Regression function to fit the parameters. The only features ultimately used were the x and y coordinates.

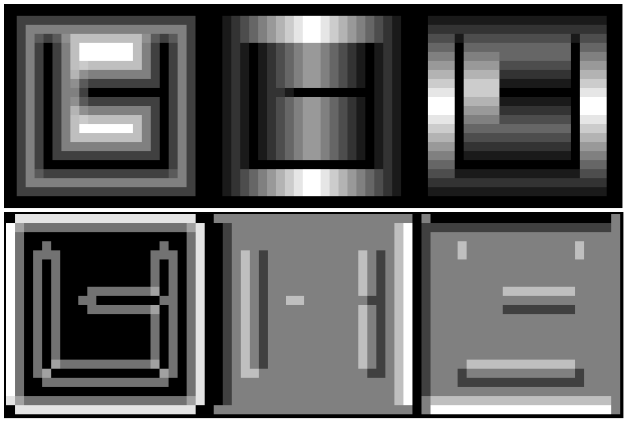

Figure 7: "Cave" map feature functons (from left to right) – distance transform, X-distance transform, Y-distance transform, gradient, X-Gradient, Y-Gradient

We found it was particularly important that obstacle cells were always included in the sample set for Locally Weighted Linear Regression. Otherwise, there is the very undesirable effect of having functions that would take the robot through walls (Figure 8). This is because the walls are a definite source of discontinuity in the value function. In theory, the value function could be discontinuous anywhere, based on the last visits between cells. In practice though, the value function is near-continuous in all areas except obstacles. This is because robots trying to sweep an area are constantly moving around, leaving a trail of continuity.

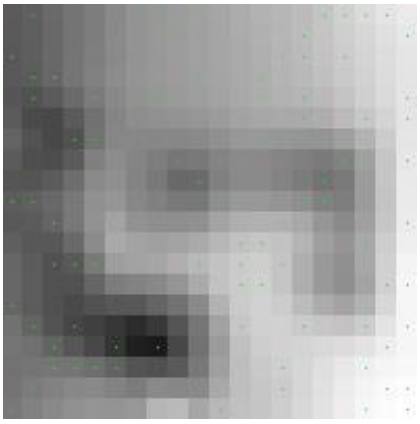

Figure 8: "Cave Map" value function approximation. Green dots indicate sampled points.

4 Conclusion

We presented the current state of the Multi-touch Multi-robot interface along with an explanation its core modules. In addition, we showed initial work towards incorporating Continuous Area Sweeping as a high level feature.

Acknowledgements

Thanks to Morgan Quigley for the fruitful discussion about the project, and for assistance with ROS.

References

Ahmadi, M., & Stone, P. (2006). A Multi-Robot System for Continuous Area Sweeping Tasks. *IEEE International Conference on Robotics and Automation*, (pp. 1724–1729).

Baker, M., & Yanco, H. A. (2004). Autonomy Mode Suggestions for Improving Human-Robot Interaction. *IEEE Conference on Systems, Man, and Cybernetics.*

Micire, M., Desai, M., Courtemanche, A., Tsui, K. M., & Yanco, H. A. (2009). *Analysis of Natural Gestures for Controlling Robot Teams on Multi-touch Tabletop Surfaces.* Retrieved 2009, from http://robotics.cs.uml.edu/fileadmin/content/publications/2009/2009-06-16-TableTop2009.pdf

Ng, A. Y. (2009). *http://www.stanford.edu/class/cs229/.* Retrieved 2009, from http://www.stanford.edu/class/cs229/notes/cs229-notes12.pdf

Quigley, M., Gerkey, B., Conley, K., Faust, J., Foote, T., Leibs, J., et al. (2009). ROS: an open-source Robot Operating System. *Open-Source Software workshop at the International Conference on Robotics and Automation.*

Ricciardi, T. (2009). *Multitouch Nav*. Retrieved 2009, from ROS: http://www.ros.org/browse/details.php?name=multitouch_nav