CSC353 Robotics Seminar Final Project
Physical iCreate in a Virtual World

Janet Guo and Yang Li

December 18, 2010

Abstract

In this paper, we introduce our project of navigating a physical iRobot Create (iCreate) through a virtual world. We developed a vision-based technique for localization of the robot and used a client-server model for motion control. In particular, the client-server model includes a set of protocols to remotely control basic iCreate motions. We successfully implemented a simple prototype that allows users to choose, by clicking in the graphical interface, a position for the robot to autonomously navigate to.

1 Introduction

Imagine walking into a museum and there, projected on the floor, a maze or a series of colorful shapes interspersed throughout the space. And as you walk forward, you notice robots–real, physical, 3D robots–navigating within the space as if the projected obstacles were real. As you watch, you notice that the obstacles are moving about and yet the robots always seem constantly aware of the changing environment. Puzzled, you scan your surroundings and see people gathered at stations around the perimeter of the projection. Curious, you walk to one of the stations and see a miniature of the projection space on a touch-screen. A small child has her finger on a green triangle and is dragging it across the screen, which causes the actual green triangle on the floor to move as well. She then takes her finger away and the obstacle stops moving. Through all this, the robots managed to avoid the changing terrain caused by the small child.

The vision described above is currently feasible, but there are many issues that need addressing. Some high-level questions: How does the robot know its position? How is the robot aware of its surrounding, i.e. knowing its position relative to its obstacles? How does the robot know where to go? In order to answer the above questions, there needs to be a meshed hardware and software framework in which to explore the answers. For instance, Figure 1 shows a possible for the framework with just one existing robot.

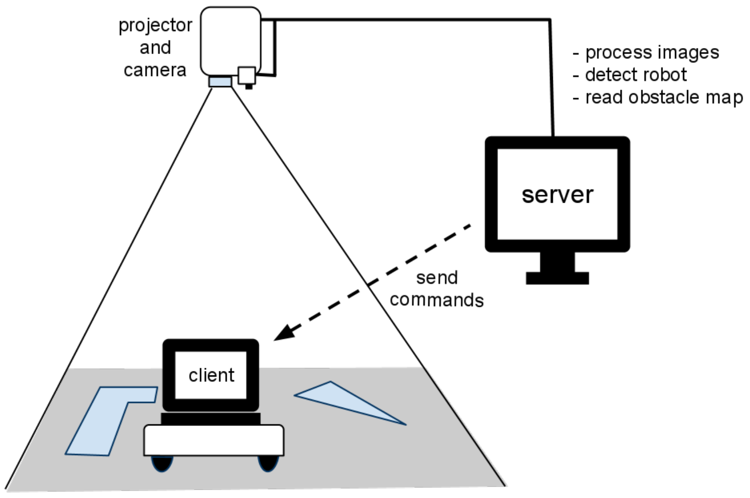

Figure 1: Basic setup of the senario

A projector as well as a camera is mounted on the ceiling and positioned to face the floor, giving a top-down view of the room. The camera keeps track of the mobile robot within its view. The server computer grabs the camera images and processes them to determine the location of the robot. On the server, a user may input obstacles. Using both obstacle and robot information, the server than computes the motion plan which includes obstacle avoidance. Commands are then sent to the client computer which induces the robot to follow the motion plan.

Our project attempts to establish a simplified version of the above framework, answering the most basic question:How does the robot know its position? We designed a simplified scenario for setting up the framework which includes:

- a single robot (the iCreate)
- assumption that there are no virtual or physical obstacles in the environment
- simple, straight-path inverse kinematics
- a GUI interface with only one control: target placement by mouse-click

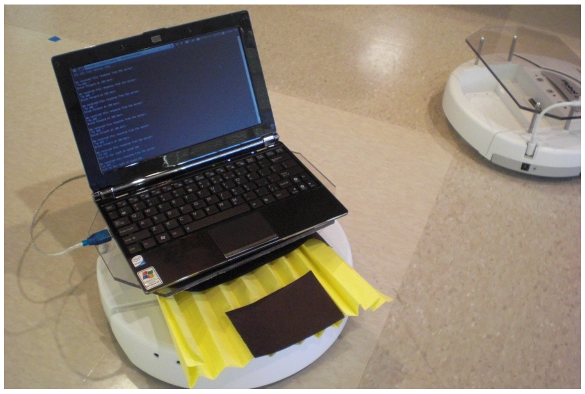

Figure 2: Physical robot: iRobot Create

In the following sections, we will discuss how we implemented the framework, some of the challenges we faced, and possible future improvements and goals which can added to the project.

2 Method and Implementation

There are three main things we need to setup the framework: localization, communication, and motion planning. For localization, the server uses the images captured from the overhead camera and, using color detection, determines the position and orientation of the robot. For communication, we implemented a client-server network model where the client process, running onboard the netbook of the iCreate, controls the actual robot while the server dictates the motion plan to the client. For motion control, we explored both reactive and non-reactive control schemes. It turns out reactive controls worked better in our situation.

2.1 Vision-based Localization

Localization is achieved by giving the robot a specific color and processing the images from a camera to detect the specific color. For our project, we set the update rate of the camera to 30 frames per second. The server computer then constantly processes the images from the camera to determine the location of the robot. For our color detection algorithm, we used the Hue Saturation Brightness (HSB) color model instead of RGB because slightly different shades of the same color can have wildly different values in RGB. As for a specific color, we decided to track black because it can be determined by brightness value alone. To detect other colors, we would have needed to use hue and saturation values as well. The algorithm to detect a specific color based only on brightness is as follows:

```
For every new frame:
  grab image from camera
  extract brightness values into an array of image pixels
  For each pixel in the brightness value array:
      If the pixel is within the threshold value previously determined:
          mark the pixel as white
      If the pixel is outside the threshold value:
          mark the pixel as black
Find the contour of the white pixels
```

The threshold value essentially determines the range of values which we consider black. Depending on the lighting in a specific environment, the threshold value may need to be calibrated. In our specific implementation, in Ford Hall room 341, we found 55 (indicating a range of 0 to 55) as the optimal threshold value for the classroom. Upon finding a contour (blob), we could determine the x- and y-coordinates of the counters center as well as the width and height of the contour. Using this technique, we set the program to look for the two largest black blobs in view, and assume that the larger of the two is the body of the robot. The smaller blob is used for orientation. So on the robot itself, we use the black netbook as the larger of the two black blobs and placed a smaller black rectangle (we fondly call it the antenna) in front of the netbook. With this setup, we represent the orientation of the robot as a vector from the center of the larger blob (netbook) to the smaller blob (antenna). See 2.1 for an aerial pictorial representation of the setup.

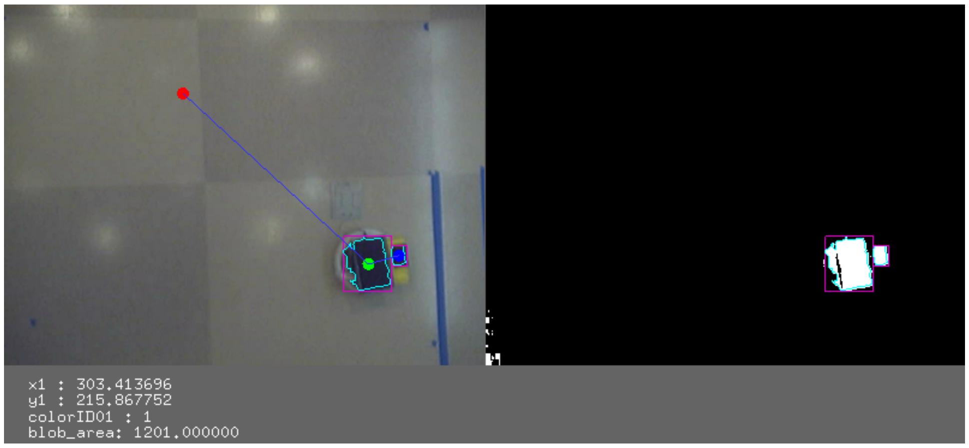
```
x1 : 303.413696
y1 : 215.867752
colorID01 : 1
blob_area: 1201.000000
```

Figure 3: Screenshot of the GUI interface (Left panel: original video, Right panel: black-and-white video for finding contours

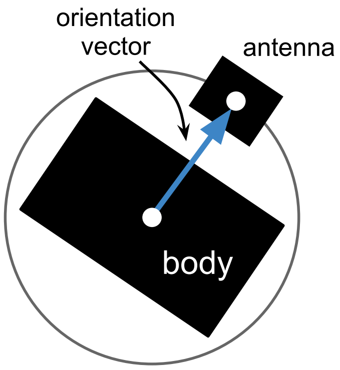

Figure 4: Determining the orientation of iCreate

In short, the vision algorithm produces the position and orientation of the robot. We implemented color detection using openFrameworks [1], an open source C++ library for simple and intuitive multimedia programming.

2.2 Client-server Communication

A client-server model maintains communication between one centralized server process and multiple client processes. Since we are using a single iCreate, only one client is connected to the server at any given time. Such a communication model is implemented in most operating systems using sockets, which are mechanisms for exchanging data between two processes until one endpoint closes the connection. These processes can be on the same machine, but in our case, the client and server machines are independent and connected via a wireless LAN (local area network). Given the dynamic IP address of the server, the client can locate the server machine and initiate a connection with the server. We chose TCP (Transmission Control Protocol) as the communication protocol because of the reliability it provides. We also used a blocking socket so that the client process on

[1]openFrameworks official website: http://www.openframeworks.cc/

the iCreate waits until it reads a message from the socket. The communication steps are shown as follows:

Server process	**Client process**
Establishes a listening socket at a port (e.g. 30000) and waits for connections from clients ...	
	Creates a client socket and attempts to connect to the specific port (e.g. 30000) of the server.
Accepts the client's connection	
	Waits for commands from the server...
Sends a command (e.g. `FW 300`) to client and waits for acknowledgement...	
	Receives the command, replies with an acknowledgement message (e.g. `received`).
Sends another command and waits....	
...	...
Sends "close" command to client	
	Closes the connection
Listening on socket...	

Table 1: Socket communication between server and client

The server program sends commands using the following command protocols that we devised, listed in Table 2.

command name	description
CLOSE	close client connection
FW *speed*	drive forward at given speed (mm/s)
STOP	stop driving
TRAN *distance speed*	translate this distance (mm) at this speed (mm/s)
LT *speed*	turn left at this speed (mm/s)
RT *speed*	turn right at this speed (mm/s)
LR *angle*	rotate left by this angle (degrees)
RR *angle*	rotate right by this angle (degrees)

Table 2: Command protocol for remote controlling iCreate

For instance, a command `TRAN 400 200 LR 90` can be interpreted as "drive the robot forward for 400 mm at speed 200 mm/s, then rotate clockwise 90 degrees". We first implemented the basic client and server model using the Berkley Sockets API, which is the most commonly used C++ socket library for the Unix system. We adapted the ClientSocket and ServerSocket wrapper classes written by Rob Tougher[2] to provide a higher abstraction for the client-socket model in our

[2]Rob Tougher, Linux Socket Programming in C++, http://tldp.org/LDP/LG/issue74/tougher.html

application. To integrate the TCP server within openFrameworks, we used the ofNetwork addon library instead of the simple ServerSocket wrapper class.

2.3 Motion Control

The objective of motion control in this application is for the iCreate to perform a simple inverse kinematics toward a target determined by a mouse-click in our GUI. To do so, once the position, orientation, and width of the robot (see 2.1) as well as the position of the target location is known, the program on the server side determines several variables:

1. DIST: the distance between the target and the iCreate
2. ANG: the angle between the iCreates orientation vector (extending from the center of the robot body to its antenna)
3. RADIUS: half the width of the bounding box of the robot body.

Using built-in functions in openFrameworks, we can easily calculate the above values (see Figure 2.3). A global constant ANG_ERR is predefined to be the margin of deviation from 0 allowed for the angle. We found 20 be an optimal value in our experiements, When ANG_ERR is small, iCreate tends to oscillate between turning left and turning right while driving straight; On the other hand, when ANG_ERR is large, iCreate drive in straight path, but may require several turns to reach the target threshold. From our experiment, we found 20 to be an optimal value such as iCreate only makes one turn during its motion towards the target, under good lighting condition.

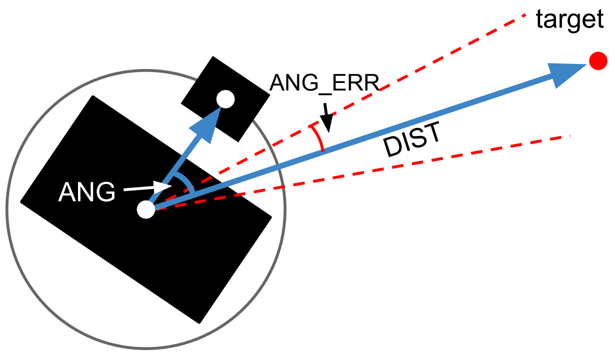

Figure 5: Compute the deviation of current position and orientation from target position and orientation

Next, we plan out the motion using one of two schemes: non-reactive and reactive.The descriptive pseudo code for both schemes are shown as follows:

Non-Reactive scheme

```
If DIST is greater than RADIUS:
    If ANG is less than negative ANG_ERR or greater than ANG_ERR:
        rotate left for ANG degrees or right for ANG degrees respectively
    Else:
        move forward DIST
```

Left and right rotation is implemented by the commands `LR angle` and `RR angle` respectively. `TRAN distance speed` makes the iCreate translate the given distance forward. No stop command is required.

```
  Reactive scheme
If DIST is greater than RADIUS:
    If ANG is less than negative ANG_ERR or greater than ANG_ERR:
        rotate left or right respectively
    Else:
        move forward
Else:
    stop
```

Left and right rotation is implemented by the commands `LT speed` and `RT speed` respectively. FW speed makes the iCreate go forward while `STOP` makes it stop. We tested both methods. The reactive scheme proved the most useful. See Section 3.2 for our reasoning.

3 Challenges

3.1 Vision Challenges

Using a vision-based technique for localization proved more difficult than we had anticipated. For example, the non-linear relationship between RGB values made real-world color detection near impossible. A material might be a uniform red color. However, because of real-world conditions such as lighting, shadows, reflectance, etc., a camera wont see only one shade of red. Instead, the red image pixels captured will fall in a range of red values. Unfortunately in RGB, two very similar shades can have wildly different values, making it very difficult to simply give a numeric range which represents a certain color. As a result, we turned to the HSB (Hue Saturation and Brightness) color model which proved much easier to work with.

Another challenge that arose, and continually remains a problem, is the programs extreme sensitivity to lighting conditions. If the lighting conditions were two dim or too bright, blobs were either wrongly detected–i.e. detecting things like shadows instead of the robot–or not detected at all. Figure 4 illustrates the lighting dependency by comparing the contour detection in our classroom with the normal lights in Figure 6(a) and with added spotlights just above our camera in Figure 6(b) . As you can see, the difference is remarkable. If we had tried to use the settings in Figure 6(a), vision localization not have worked. On the other hand, the setting in Figure 6(b) worked perfectly.

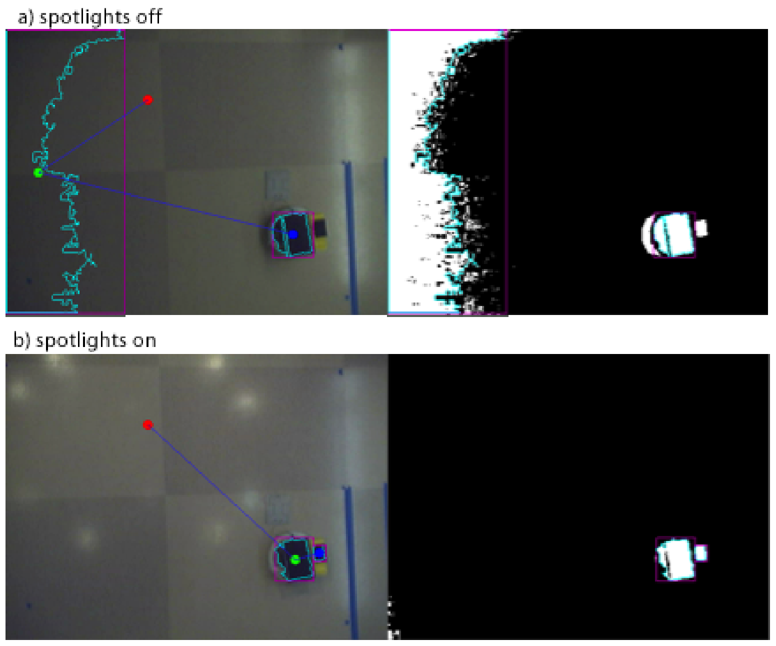

Figure 6: Effect of lighting condition on color detection

In order help assuage the lighting dependency problem, we implemented a color threshold variable that users can calibrate to their working environment. However, despite even the best calibration, light bouncing off a moving robot will vary, causing what we call camera jitter, meaning the shape of the contour detected jumps slightly from frame to frame. As a result, the center of the contours constantly jump around as well, even if the robot is in one place. This causes the robot to oscillate unnecessarily.

To reduce oscillations caused by jitter, we compared the most updated center position coordinates to the coordinates from the frame before. If the difference between the two sets of coordinates was less than the margin of error (the values stored in the variables ANG_MARGIN, for the angle, and DIST_MARGIN, for the distance), we keep the older coordinate and throw out the new one. If the difference is significant enough to indicate actual movement of the robot (meaning it is greater than ANG_MARGIN or DIST_MARGIN), update to the newer coordinate. This method turned out to work quite well; the robot no longer oscillated because of jitter (our values for ANG_MARGIN and DIST_MARGIN were 3 and 15 respectively).

3.2 Disadvantages of non-reactive motion control

The non-reactive motion control scheme is extremely difficult to implement in practice because of the inaccuracy of distance and angle measurements using a camera. There are several sources of errors.

First, mapping the on-screen distance in pixels to the physical distance that iCreate travels in milimeters is inaccurate. In order to find the pixel to milimeter ratio, we attempted multiple times to measure, within the GUI, the number of pixels between the starting and final position of the iCreate when it executes the command TRAN 500 (move forward for 500 mm). However, because of camera distortions (it is hard to ensure that the camera is pointed straight down) and the unexpected curvature of the iCreates path, the ratio we computed turned out to be unreliable.

Second, even if the ratio were correct, since the centers of the body and of the antenna are constantly changing due imperfect lighting, it is very likely that the angle and distance offsets computed were not reasonable, causing the iCreate to overshoot its target and re-perform navigation which often does not reach the target either. Since in this scheme, the iCreate API functions we needed, `translate(distance,speed)` and `rotate(angle,speed)`, were non-preemptive, i.e. would not allow interruption while in the middle of an action, we were unable tell the robot to stop when it reached a target. To work around this, we used reactive motion control, implemented by functions such as `forward(speed)` and `stop()`, which return the control back to the user program immediately after they have been called.

4 Future Goals

Using this project as a base, we propose the following list of possible future goals, in order of increasing difficulty:

1. Setup floor projection
2. Make iCreate follow a sequence of straight-line paths
3. Add collision detection with physical obstacles using bump sensors
4. Add virtual obstacles to the environment, find the appropriate collision avoidance algorithm to make iCreate navigate around obstacles autonomously
5. Add in multiple robots (Note: the currently implemented network model only supports one client robot. To support multiple clients, we may consider rewriting the client program as an openFrameworks module with the ofNetwork addon.

5 Conclusion

In the project, we developed a framework that allows a mobile robot to autonomously navigate towards a single target. Our main contributions are localizing the robot through computer vision and establishing wireless communication between the robot and a remote computer. This framework is a starting step on which future projects involving robots, augmented reality, and interactive art can be built.