

# Exploring with LEGO Robots

Daniel Limbrick

Emily Sherrill

Graduate Student Advisor: Daniel Balasubramanian

August 4, 2006

## Abstract

In this paper, we describe the results of our SIPHER summer project, which involved using a Lego Mindstorm robot to discover the layout of a maze and communicate this information to a PC, which acted as a controller. We show the process we used to simulate communications and interaction from the computer to the robot and vice-versa. In addition, we explain the algorithms of the simulator, as well as the serial-interface circuit, which is used in the real implementation.

## 1 Introduction

Our project can be considered a scaled-down version of using unmanned vehicles for the use of uncharted territory, such as the Mars Rover. The basic idea is the following: we start with a Lego Mindstorm robotics kit, which we will use to build a robot that can explore a maze that with an unknown layout. As the robot explores, we want to be able to communicate information about the layout of the maze back to a base station, which in our case is a PC. We also want to use the base station as a controller, meaning that the PC determines the exploration sequence based on its current view of the maze. Once the maze has been completely explored, we want to be able to display the layout of the maze on the PC, and allow the user to select a point in the maze, and have the PC transmit the control sequence that will the robot drive to that location.

Our initial tasks included learning how to move the robot in a straight line, completing semi-accurate turns and rotations, and developing algorithms for maze mapping and traversing to a point. After doing this, we were familiar with the abilities and the limitations of the robot and our algorithms. We then worked on

the communication interface so that the robot and PC could communicate wirelessly, after which point we began to integrate all of these pieces together to accomplish the overall task.

## 2 Materials

### 2.1 Lego Mindstorm Robot

The robot created for this project includes the following parts:

- Two rotation sensors
- One touch sensor
- Two motors
- Two 4.96 x 2.8 cm back wheels
- Two 4.2 x 1.0 cm front wheels
- One Lego Mindstorms RCX programmable brick

### 2.2 Surface of Travel

The testing of robot movements performs with greater accuracy on poster board, which has texture that does not allow the wheels to slide or stick. Thus, the surface of our maze has been covered with poster board to allow for a greater probability of successful processing of the maze by the robot.

The maze was created from a wooden base using plastic walls and pegs to create the walls. Because of the robot's width, the maze was designed into squares of 36cm x 36cm. One maze block, referred to in this report, is said to be the 36cm length of a square.

### 2.3 Programming Language

All code downloaded onto the RCX 2.0 brick was written and compiled for leJOS (Lego Java Operating System), a replacement firmware for the Lego Mindstorms RCX brick. leJOS is a Java Virtual Machine that fits within the 32kb on the RCX. Programs written for leJOS are written in a condensed version of Java that contains additional leJOS classes. The control applications on the computer side were written using the latest version of Java.

We required the ability to read/write information to/from the serial port of the

PC. To do this, we wrote the low-level communication routines in the C programming language, and used the Java Native Interface (JNI) to import this functionality into our Java program.

## **3 Mid-Summer Summary**

### **3.1 Challenge**

Accomplish specific tasks to discover the abilities and the limitations of the robot and wireless communication.

### **3.2 Process**

After accomplishing our first goal of programming the robot to move in a straight line, we began work on completing accurate turns and rotations with the robot. After discovering that the accuracy necessary for the maze to be mapped and traversed correctly could not be easily reached, we moved on to developing algorithms for mapping the maze based on the exploration of the robot and for traversing the maze to a point selected by the user. Finally, we developed code to read and write from a serial port, thus allowing the future capability of Bluetooth connectivity.

### **3.3 Problems**

A variety of problems arose while working on accomplishing the above tasks. The first major problem that occurred was a result of the robot's original design, in which the front wheels were independent of the motors. Thus, the front wheels would slip during the forward movement of the robot. Once this problem was corrected, a second problem arose while attempting to make the robot perform precise turns and rotations. The front and back wheels stick and slip on the surface of the poster board, although not always at the same time or consistently during the testing. However, if the wheels are removed and the robot is allowed to perform the program on its gears alone, the movements of the robot are completed with near perfect accuracy.

## 4 Maze Mapping and Go-to-Point Algorithms

### 4.1 Challenge

Develop algorithms to control the robot as it explores the maze and as it travels to a user-specified point in the maze.

### 4.2 Algorithm Solution

A maze class was created in Java to contain all the methods and various variables associated with the actual physical maze. This class holds the dimensions of the maze with a coordinate system for each block of the maze. The blocks are assigned a number which represents what each block is in the maze: A zero represents an undiscovered area of the maze, a one represents a wall, and a two represents a previously traveled/discovered area of the maze. The dimensions of the maze are specified by the user, as well as the current location of the robot in the maze. Once the dimensions are obtained, the actual two-dimension integer array representing the maze is created with two extra rows and columns, which act as a buffer area. The buffer area is automatically considered to be walls.

Once the starting position is obtained, the maze mapping algorithm will start the robot on its exploration of the maze. As the robot reaches a block in the maze, it will perform an explore algorithm which uses the robot's touch sensor to determine where walls are located in the maze. The touch sensor value is relayed back to the computer where it is stored, a true will represent the location of a wall or false otherwise. The computer's algorithm is designed to send the robot toward the next zero location in the maze. From that location, the robot will execute the explore algorithm again. If more than one zero location exists around a block, the maze mapping algorithm decides on the movement of the robot using the following order of favoritism where the first direction is favored over the following: forward, right, left. However, if no surrounding position is a zero, the algorithm will determine if there any other reachable undiscovered areas remaining in the maze. If the computer locates open areas remaining in the maze, it will send the robot back along a previously visited route until it reaches the correct location. This algorithm will continue until all reachable zeros in the maze layout have been visited. If the robot should be surrounded by walls or block from a portion of the maze by walls, the remaining unreachable portions of the maze will be set as walls by default in the array representation.

The go-to-point algorithm begins upon the completion of the maze mapping algorithm. A MazeFrame object is used to track and display the progress of the robot as it maps the maze. Once the go-to-point algorithm becomes active, we

begin listening for mouse clicks. This allows the user to click on any portion of the maze representation being displayed on the screen. The selected coordinates are converted to fit the maze dimensions, and then sent to the go-to-point algorithm. If the user has selected an invalid location, such as a wall, then algorithm is not allowed to begin. The user is informed about the invalid selection and asked to select again. The go-to-point algorithm performs a depth-first search with marking to obtain the path for a robot from its current location to the point the user clicked. The correct path is stored in a stack as directional string commands. Before the sequence of commands is sent to the robot, we highlight the path the robot will take. While the robot is completing its traversal, the user clicking on portions of the maze will not affect the program, as we do not listen for mouse clicks during this time. Once the robot has reached the user-selected point, the user is informed the process has completed and asked to select another point.

### 4.3 Problems

Various problems arose during the testing and running of both algorithms. At first, several logical errors existed in the code, but as those errors were located and corrected, some design flaws were discovered. Originally, the user was able to click on points in the MazeFrame as the maze mapping algorithm was processing. Once that algorithm completed and the second one started, the areas that were clicked during the maze mapping algorithm were used as the points for the go-to-point algorithm. The algorithm would move the robot to each point that was previously clicked in the order that the points had be selected. The correction was to create a method that could be called to activate the MouseListener at the end of the maze mapping algorithm rather than activating it in the constructor of the MazeFrame.

A second problem that arose was an issue with when the go-to-point algorithm would not recognize that the robot was trapped in a block or an area and unable to reach the remainder of the maze. Our first correction for this problem was to alter the logic of the method `reachableZeros`, which determines if a zero can be reached. Basically, the method checks in all four directions, and if each direction is a wall, then the zero is unreachable. We added the logic to consider a zero surround by either a wall or a two as unreachable. This correction worked for our trapped cases but did not for the complete maze, because it would not reverse back to a location where it could follow the direction of a different zero. So, the `reachableZeros` was restored to its original logic, and a second method was created, `reachableFromTwo`. This method allows the computer to locate a previously visited block from where a zero can be reached. Thus, the computer is able to determine when it is trapped and when it should follow a previous path to locate a zero.

## 5 PC/Robot Communication Simulation

### 5.1 Challenge

Simulate the interaction and communication between the computer and the robot during their processing of the maze in order to locate errors that may exist in our algorithms.

### 5.2 Algorithm Solution

Simulating the PC/Robot communication allows us to test our algorithms in an ideal environment, with reliable communication and a robot making accurate turns and sensor readings. Using two different computers, one acting as the robot and the other acting as the computer, two Bluetooth wireless adapters were used to connect the PCs in a similar fashion to what the actual wireless connection would be. Next, a fake maze was designed in the robot simulator. Once the robot simulator has completed the maze exploration, the computer simulator's maze should match the fake maze. After testing the two algorithms considerably, the previous statement proves to be correct: At the complete of the mapping process, the robot and computer's mazes match.

Originally, the movement commands were designed to be string commands, but after discovering that robot-to-computer communications can possibly cause data to be altered or lost, the commands were changed to byte operation commands. Once this change was completed, the algorithms were tested on a variety of mazes: a 10x10, a maze where the robot starts in a position surrounded by walls, and a maze where the robot is block by walls from the lower half of the maze. With the addition of the `reachableFromZero` method, the simulation performed correctly for each case.

### 5.3 Problems

During the simulations, only three problems occurred of minor concern. The first problem was an accidental discovery. As the simulation is running, the command window is present along with the `MazeFrame` depiction of the maze. The command window displays the messages received from the robot simulator as it is returning wall locations and performing movements. If, while the program is running, the user scrolls up in the command window, it will result in a back up of the communications port. This results in messages sent from the robot to the computer being missed or skipped. If the program misses the command that states that the robot has completed its process, then it will become an infinite loop as a result of waiting

for the complete message to be sent before it continues to the next process.

The two algorithms, maze mapping and go-to-point, each operate with their own DLL. As a result, the second major problem that occurred was in dealing with the communication port control. In our DLL, we did not include a method to release control over the communication port. Because of this, the go-to-point algorithm would not be allowed access to the communication port to send its information to the robot simulator, resulting in a hot stop error. To correct this, we attempt to use only one DLL and have it referenced as a static in the maze mapping algorithm class. However, this too resulted in a hot spot error, so a close method was created. The close method causes the current DLL to release its hold of the communication port for the following to have access to it.

The third issue was discovered on the robot-side simulator, again by accident. To run the simulator, the robot side is started first, and then the computer side is started. All of the processing occurs on the computer side, with only the movement and explore methods running on the robot side. If, a command window is already open and running the robot side simulator and a second one is open to run the simulator as well, the two windows will communicate with one another using incorrect commands, and thus interfering with the true simulator.

## **6 Circuit**

### **6.1 Challenge**

Connect a circuit that communicates with both the Bluetooth device used by the computer and the RCX Lego brick. Test its ability to transfer information from the RCX Lego brick to the computer and vice versa.

### **6.2 Algorithm Solution**

The typical RCX Lego brick communicates to an infrared tower through infrared light emitting diodes and an infrared receiver. To expand on its communication capabilities, we soldered wires onto the LEDs and receiver and connected these wires to an external circuit. The circuit, custom-built by our graduate student mentor, converts the signal from the wires to serial communication. With this serial port, we are able to connect a Bluetooth Brainbox, identical to the device that we use for the Bluetooth connectivity on the computer.

We tested the Bluetooth communication between the PC and the RCX brick by running separate Java programs on the RCX brick that sent and received information. The program that received data from the computer simply contained a

command to print the value of the received data onto the LCD screen. Simultaneously, the computer ran a program that sent bytes to the RCX brick via Bluetooth communication. To send information from the RCX brick, we wrote a program to display the entire ASCII table and ran it on the brick. We viewed the results in the hyper terminal of the computer. In order to get accurate results, we had to calibrate the potentiometer in the serial communication circuit in order to regulate the differences in voltage in the circuit as compared to the RCX brick.

### 6.3 Problems

Sending information to the RCX brick worked successfully. However, receiving information was not as reliable. We could fine-tune the potentiometer to get perfect results for receiving information from the RCX brick, but this calibration had to be performed quite frequently. The required resistance is determined by the voltage level of the incoming communication signal from the RCX, which is in turn affected by the battery level of the RCX (which is constantly decreasing). However, the comparison signal (whose voltage level is determined by the potentiometer) is also being powered by a battery source that is constantly draining. These minor fluctuations in voltage levels are enough to cause unreliable communication. Using power supplies instead of batteries could alleviate this problem, but given that the robot needs to move across a large area, the size would make it impractical. Also, during the use of the circuit, the diodes located on the front of the RCX brick sometimes burned out, but we were unable to reproduce this problem consistently, and thus could not determine the precise conditions that triggered the excess current.

## 7 Conclusions

We successfully adapted the RCX brick for Bluetooth communication, and proved that under certain conditions, it is capable of sending and receiving with accuracy. We were also able to simulate the robot exploring, mapping, and finding destinations in the maze. However, we were not able to receive information from the RCX brick with accuracy in the real demonstration because of the difficulty in filtering out the excess noise generated by the use of the motors. Had time permitted, we would have liked to address this issue, either by building a hardware filter, or accounting for the noise in our communication protocol.