

RoombaWriter

Chalk-Drawing Robot

EECS 149 Embedded Systems

Professor: Edward Lee

GSI: Shanna-Shaye Forbes

Mentor: Igor Paprotny

Students: David Burban, Skot Croshere, Dan Lynch, Andre Zeumault

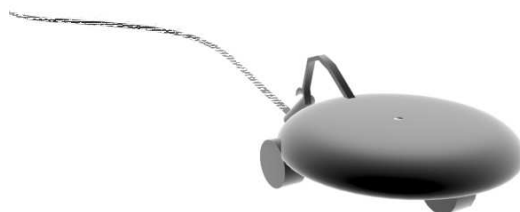
May 14th, 2010

Table of Contents

Abstract	iii
1 Introduction	1
2 Algorithms and Models	1
2.1 Simulation	1
2.2 Configuration Space	2
2.2.1 Line Segments	2
2.2.2 Arbitrary Curves	4
2.2.3 Circles	5
3 Implementation	6
3.1 Protocol and State Machine	6
3.1.1 Packet Structure	6
3.1.2 Internal Program Text	7
3.1.3 States	7
4 Software	9
4.1 Processing	9
4.2 Keil μ Vision IDE/Debugger/Simulator	9
4.3 LabWindows	10
4.4 screen	10
4.5 Vim3D	10
5 Hardware	10
5.1 Linear Actuator	10
5.2 H-Bridge Circuit	11
6 Challenges	12
6.1 Iterations	13
7 Results	13
8 Division of Labor	15
9 Concepts	15
10 Feedback	16

RoombaWriter

Chalk-Drawing Robot



Abstract

This report discusses the algorithms and models associated with building a robot with the capability of plotting 2D drawings. Our team has built a robot that renders drawings based on commands sent from an interactive user interface, or from commands generated after parsing a conventional vector graphics file format. Our idea is to take an ordinary iRobot, and retrofit it to draw images with chalk or a pen. The paths will be generated via external software that generates commands based on user input and sends commands to the robot over bluetooth. We think of it as a free-form printer.



RoombaWriter

Chalk-Drawing Robot

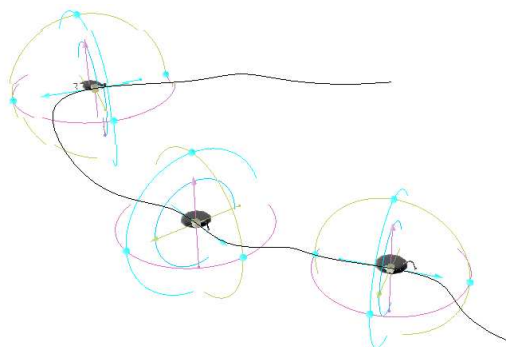
1 Introduction

The project's goal is to build a plotting robot. Along the way, we discovered through simulation that generating the path that a robot must traverse for a given drawing is not obvious. Many interesting geometric and algorithmic problems have been solved, and have been applied to this project. To abstract these algorithms, we have built a paired system connected through a bluetooth network that provides a graphical user interface. This interface allows users to simply draw line segments on the screen as in any conventional drawing program, and then sends a series of commands as a byte stream to the robot. The robot parses these commands until an end of transmission character is received, and then begins plotting. The mechanism for plotting involves a linear actuator that raises a metal hinge attached to a piece of chalk or a pen, which allows full control for where lines are needed.

2 Algorithms and Models

2.1 Simulation

Before any implementation, our team incorporated what we learned about simulation in previous EECS 149 labs. We utilized software to find implementation problems before writing any code or building any hardware. We chose to use software written by one of our group members, because the program was written in C++ and we were able to extend it to perform complex functions like differentiation or projecting vectors, with the ability to write out data as a file in a conventional manner.



The first simulation.

The first test we performed involved drawing a path on the screen, and simulating the robot traversing along this path. We noticed right away that the pen attached to the back of the iRobot was not following

the path. This led us to solve an interesting problem involving configuration space.

2.2 Configuration Space

Over the first stages of development, numerous hours were dedicated to solving mathematical problems involving the robot's plotting ability. These relationships required trigonometry, geometry, calculus and linear algebra, and a great deal of thought. The following three sections will discuss methods that we came up with for transforming data and drawings into velocities and commands.



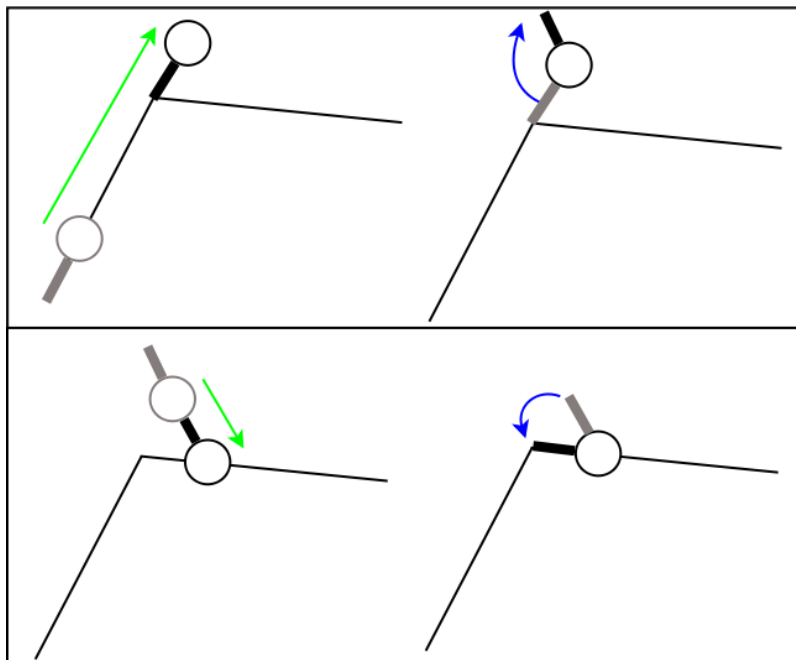
2.2.1 Line Segments

The configuration space of the robot involves the possible ranges of movement and positions it can perform. To draw a single line segment requires two rotations and two driving movements. This is due to the fact that a change of basis is required from the coordinate frame of the robot with its center point as an origin to the center of the pen or chalk.

The algorithm for this change of basis when drawing line segments for a set of segments L is as follows:

- 1: **for all** i such that $0 \leq i < L.size() - 1$ **do**
- 2: $D_1 \leftarrow L[i].length$
- 3: $R_1 \leftarrow$ angle required to rotate to required starting point on $L[i + 1]$
- 4: $D_2 \leftarrow$ distance between current position and required starting point on $L[i + 1]$
- 5: $R_2 \leftarrow$ angle between current orientation and $L[i + 1]$
- 6: place the pen down
- 7: drive D_1 units
- 8: pick the pen up
- 9: rotate R_1 units
- 10: drive D_2 units
- 11: rotate R_2 units
- 12: **if** $i + 2 = L.size()$ **then**
- 13: drive $L[i + 1].length$
- 14: **end if**
- 15: **end for**

As you can see, this requires that we look at every pair of line segments. The figure below demonstrates this algorithm for two line segments, and assumes that the robot begins in the correct starting position:



The Plotting Algorithm

2.2.2 Arbitrary Curves

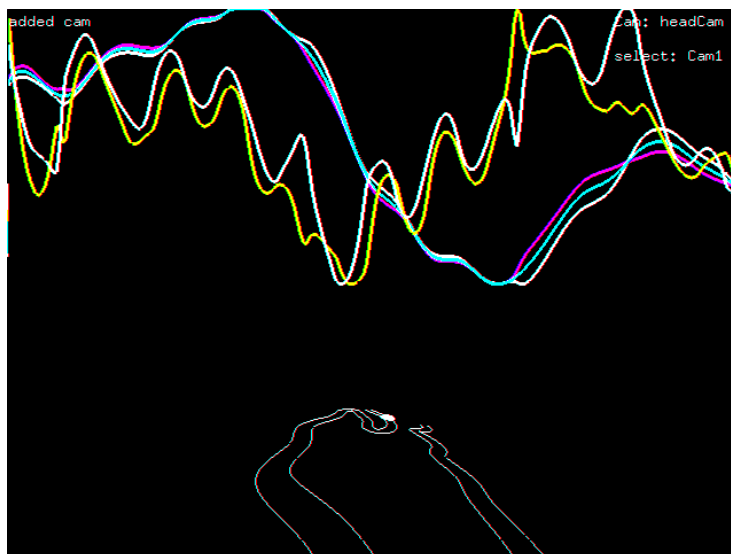
Another problem arises when you have an arbitrary path of equally spaced segments and want to determine the wheel speeds that will produce such movement. The first method we derived and implemented within our simulation. The idea is that you can approximate an arbitrary set of curves as small line segments. First you simulate the robot along an arbitrary path, then differentiate the position for x and z coordinates (of the simulated wheels positions) with respect to time. The velocities are represented as the global velocity in x and global velocity in z . To find the relative velocities, you can then project these velocities as a vector $\langle x, y, z \rangle$ onto the differentiated path, or orientation vector on the path that corresponds to that given velocity. The projection onto a vector v can be performed using a projection matrix, which is given by the vector v multiplied by its transpose v^T :

$$P = vv^T = \begin{bmatrix} v_x^2 & v_x v_y & v_x v_z \\ v_x v_y & v_y^2 & v_y v_z \\ v_x v_z & v_y v_z & v_z^2 \end{bmatrix}$$

After the projection is done, you now have the velocities relative to the iRobot's orientation. The entire process of computing wheel speeds from an arbitrary path P of points can be done as follows:

- 1: $D \leftarrow$ distance of wheels from center of robot
- 2: **for all** p such that $0 \leq i < P.size() - 1$ **do**
- 3: $\hat{f} \leftarrow \frac{P[p+1]-P[p]}{|P[p+1]-P[p]|}$
- 4: $\hat{u} \leftarrow \langle 0, 1, 0 \rangle$
- 5: $\hat{w} \leftarrow \frac{\hat{u} \times \hat{f}}{|\hat{u} \times \hat{f}|}$
- 6: $leftWheel[p] \leftarrow P[p] - D \cdot \hat{w}$
- 7: $rightWheel[p] \leftarrow P[p] + D \cdot \hat{w}$
- 8: $O[p] \leftarrow \frac{d}{dt}P[p]$ (set up orientation vectors)
- 9: $P_o \leftarrow O[p]O[p]^T$ (set up projection matrix)
- 10: $L[p] \leftarrow P_o \frac{d}{dt}leftWheel[p]$ (store calculated left wheel velocities)
- 11: $R[p] \leftarrow P_o \frac{d}{dt}rightWheel[p]$ (store calculated right wheel velocities)
- 12: **end for**

Below is a screen shot from our simulation after performing these calculations.

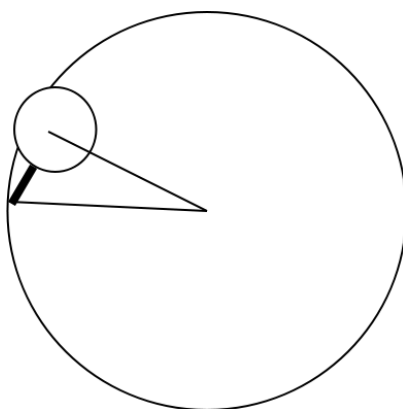


Simulation of the robots wheel velocities.

After this algorithm is finished, the arrays of vectors R and L will contain the velocities required to allow the robot to traverse the path given by a discrete sequence of points. Note that when we denote derivatives in the pseudo code, this would be done numerically in practice.

2.2.3 Circles

We also determined the math required to send a robot along a sector of a circle. Suppose the robot has radius R_r and the distance of the chalk from the center of the robot is ℓ . If we want the robot to drive a radius of R across an angle θ , then the robot will have travelled $R\theta$, but its wheels will have travelled $(R + R_r)\theta$ and $(R - R_r)\theta$. We can take the ratio of the distances travelled to determine the ratio of the velocities because the velocity is simply the distance per time. Thus, we have the relation, $v_{inner} = \frac{R - R_r}{R + R_r} v_{outer}$, where v_{inner} is the velocity of the wheel on the inside of the circle.



Drawing Circles.

These calculations are not right just yet. We need to account for ℓ . Given that we would like the robot to draw the radius R , we then need to calculate the radius upon which the robot will drive. This is a simple Pythagorean relationship, $R_d = \sqrt{R^2 - \ell^2}$. This gives us the final equation:

$$v_{inner} = \frac{\sqrt{R^2 - \ell^2} - R_r}{\sqrt{R^2 - \ell^2} + R_r} v_{outer}$$

3 Implementation

After diving into the theory, our team was ready to start implementation. Our implementation was established in four stages. The process included developing a protocol, a state machine, a user interface, and hardware design.

3.1 Protocol and State Machine

Our State machine and Protocol are almost one in the same. However, there is not a complete bijection between the two, in that we don't (in practical use) determine all states as packet types. However, all packet types are represented by a state. The idea is that the packet type determines a subset of the states and includes an implicit predicate that upon true allows the robot to enter another state, where it then seeks the next command.

3.1.1 Packet Structure

We defined an 8-byte packed structure that contains enough data to fully control the robot. Each command contains a type, an attribute, a left wheel velocity, a right wheel velocity, and a pen position. Below is the schema for the command packet:

name	type	attr	lVel	rVel	pen
size	1B	2B	2B	2B	1B

GCC allows you to specify a packed structure with the `__packed__` attribute. This disables any memory alignment or padding that would normally occur when defining structures in C. When receiving data from bluetooth as a continuous byte-stream of characters, we can just cast every sequence of 8 characters to the packed structure, allowing us to access data without bit shifting:

```
struct draw_cmd {
    uint8_t type;
    uint16_t attr;
    int16_t lVel;
    int16_t rVel;
    int8_t pen;
} __attribute__((packed));
```

However, we can only retrieve one character at a time from the universal asynchronous receiver/transmitter (UART) on the Luminary. Thus, the robot pulls one character at a time from the bluetooth port through the UART, where every 8 bytes represents a command, unless we receive a "@" character, which denotes the end of transmission.

3.1.2 Internal Program Text

The entire byte-stream is parsed into memory with a program counter starting at zero. As each command is executed, a test is checked against the *attr* field until it returns true. Once this happens, the state is changed to the UPDATE state where the program counter is incremented and variables are reinitialized.

3.1.3 States

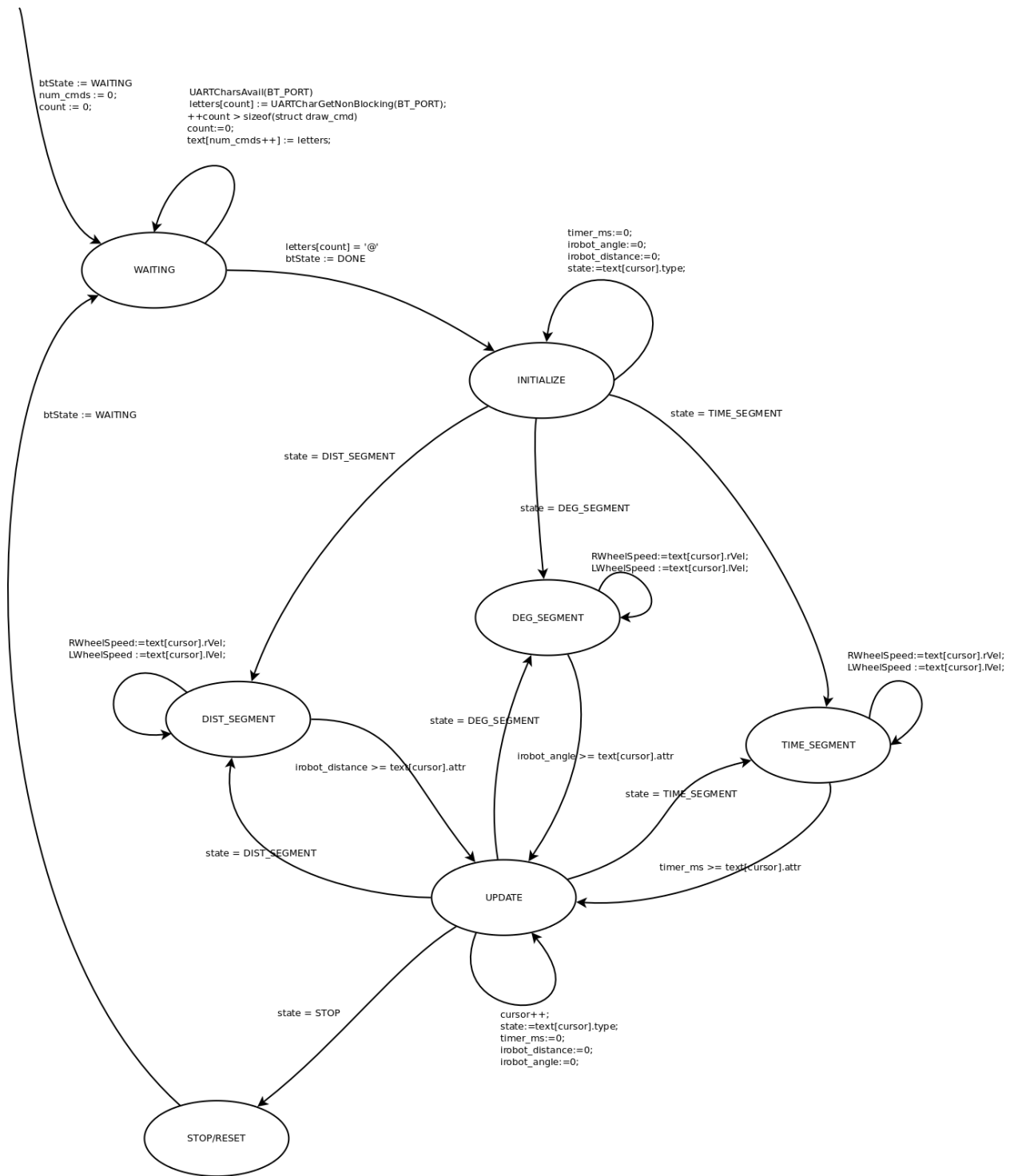
Our protocol and state machine are very integrated, so before fully discussing the meaning of the *type* and *attr* fields, it is important to understand the states. We have defined the following states:

```
INITIALIZE 0x00
DRIVE 0x01
RESET 0x02
UPDATE 0x03
STOP 0x04
TIME_SEGMENT 0x05
DIST_SEGMENT 0x06
DEG_SEGMENT 0x07
DELAY_SEGMENT 0x08
```

Notice that each state is represented as 1 byte, which is what the command packet's *type* field corresponds to. The *attr* field in the command packet has a new meaning depending on what the *type* is. The commands sent over bluetooth are all higher than 0x03, the others occur internally within the robot.

1. INITIALIZE
Initialize all values.
2. RESET
Start listening for commands again over the bluetooth.
3. UPDATE
Increment the program counter and read the next command according to the protocol.
4. STOP
Stop the robot.
5. TIME_SEGMENT
The *attr* field is interpreted as the time in milliseconds until the UPDATE state should be entered.
6. DIST_SEGMENT
The *attr* field is interpreted as a distance in millimeters. When this distance has been traversed, the iRobot enters the UPDATE state.
7. DEG_SEGMENT
The *attr* field is interpreted as an angle in degrees. When this angle has been rotated through, the iRobot enters the UPDATE state.

Below is the entire overview of the state machine including reading the bluetooth port.



The State Machine.

4.3 LabWindows

During the phase involving the robot arm, we built a custom arm controller using LabWindows. This applet had widgets that could control the many degrees of freedom that the robot arm had, and helped characterize the robot arm. Using these results, we were learned that we had to move to another design.

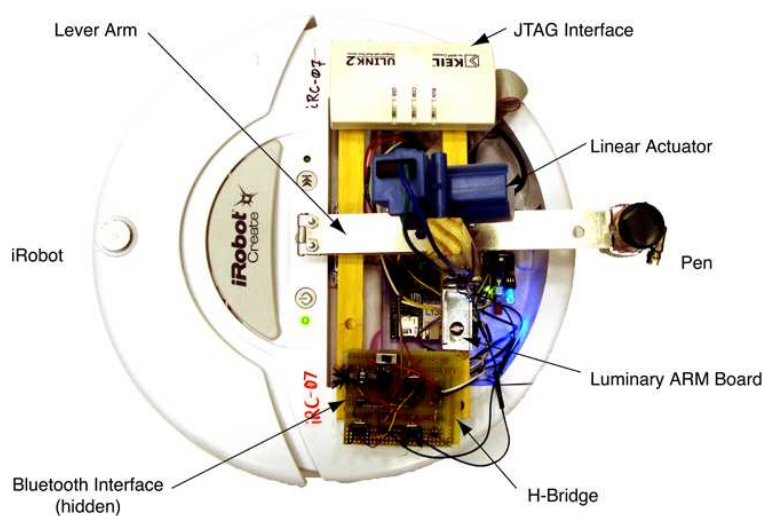
4.4 screen

When developing and debugging our serial protocol, we had to manually send bytes out to the serial interface. Screen provided the barebones minimum that we needed to send the data using a UNIX terminal.

4.5 Vim3D

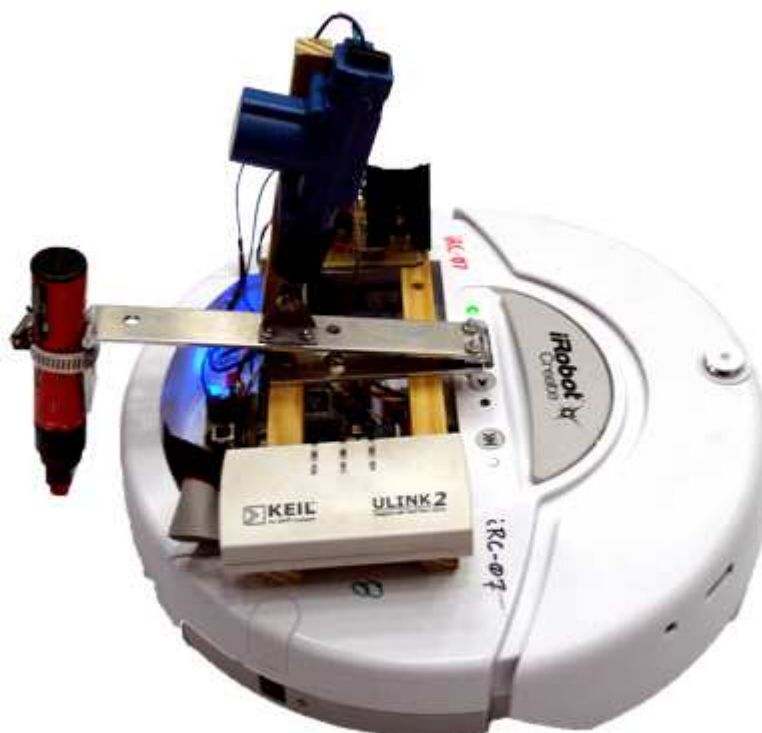
Vim3D provided a temporal and spatial simulation that helped give us insight into the geometric relations when plotting, which greatly helped with our algorithms. In addition it was a useful interface for testing the mathematical theories.

5 Hardware



5.1 Linear Actuator

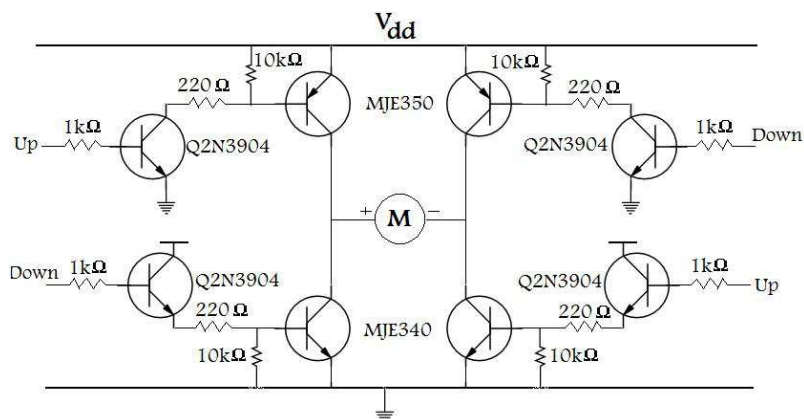
The robot could not just simply leave the pen in a single position. To move the pen up and down we used a linear actuator attached to a metal hinge. This hinge then pushed the pen to the floor or pulled it up.



The actuator we are using is very fast, and definitely gives us enough force to jam the chalk to the ground. The linear actuator takes a single voltage, and the polarity of the current determines the direction of the movement. In order to manage the polarities of the current, we connected two GPIO pins on the luminary to a custom H-Bridge circuit.

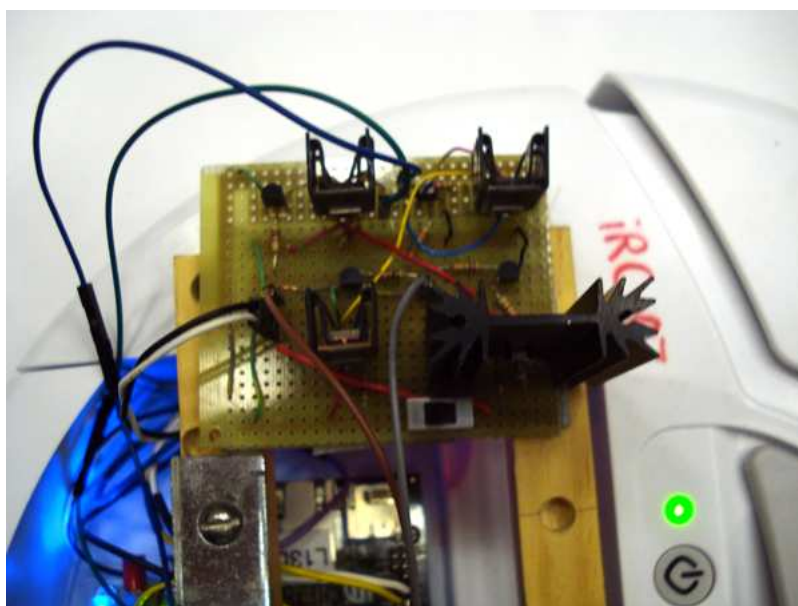
5.2 H-Bridge Circuit

Standard component in robotics that enables bi-directional control over high power and low power dc motors is an H-Bridge. Although H-bridge circuits can be implemented using BJT or MOSFET technology, for simplicity we chose a BJT based design. The circuit diagram is shown below:



Circuit Diagram for the H-Bridge.

The high power BJTs enable approximately 12 V at 1A to be delivered to the linear actuator. The smaller, low-power BJTs (Q2N3904) are used to drive the power transistors, boosting the limited output current from the Luminary's GPIO pins ($\approx 8\text{mA}$) into an acceptable 1A to be passed through the linear actuator. Interfacing external circuitry to a microcontroller such as the Luminary board requires care to be taken to ensure that the microcontroller is surge protected. The $1\text{k}\Omega$ resistors limit the current that can be drawn from the microcontroller to a safe limit that is below 1mA. The base current provided by the microcontroller pins is amplified by the first stage and that current provides the base current to the power transistors. The resulting collector current is amplified twice by two current gain parameters(beta) and therefore a high motor current is achieved via a small input microcontroller current.



The completed H-Bridge.

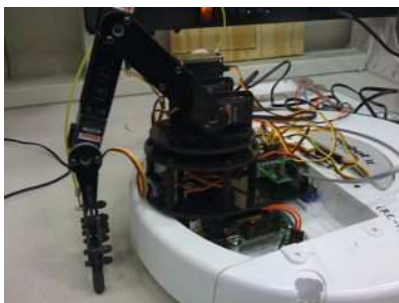
6 Challenges

After doing extensive testing, we have determined that the sensor values from the iRobot (angle turned, distance traveled) are unreliable. The robot apparently lacks actual sensors and relies on the commands given to the drive wheels. However, due to frictional differences between the two wheels, the outputs we received were not only incorrect, but also non-linear. We attempted to correct for the most common cases by multiplying the values with a constant, however this constant needs to be configured every time we launch the robot, as it does not seem to be consistent.

A huge problem we had involved storing the array of commands on the robot. We originally thought the data being sent was corrupt, as printing the data returned mostly garbage. It turns out the problem was in using the OLED display while receiving commands. The calls to the display were not restoring the stack. Once we stopped displaying updates on the screen, we were easily able to store the data into the statically

allocated array.

6.1 Iterations



The original robot arm.

We had significant problems designing a mechanism that would be able to move the chalk up and down. Our first design centered around the Lynx Robotic Arm. While this solution was able to move the chalk up and down, it had significant structural integrity problems, including twisting when the robot was making a turn. Besides that, the arm could not apply enough downward force to make the chalk draw, even when we were sending commands that would force it into the ground.

The next iteration of the chalk mechanism was to use a solenoid to lift the arm up. However, the solenoid that we had was underrated. The other disadvantage of the solenoid was that it needed a return spring to go back to the original position. Finally we arrived at the solution we were looking for—the linear actuator. The actuator gave us the power to keep enough force on the chalk and operated great after finishing the H-Bridge.

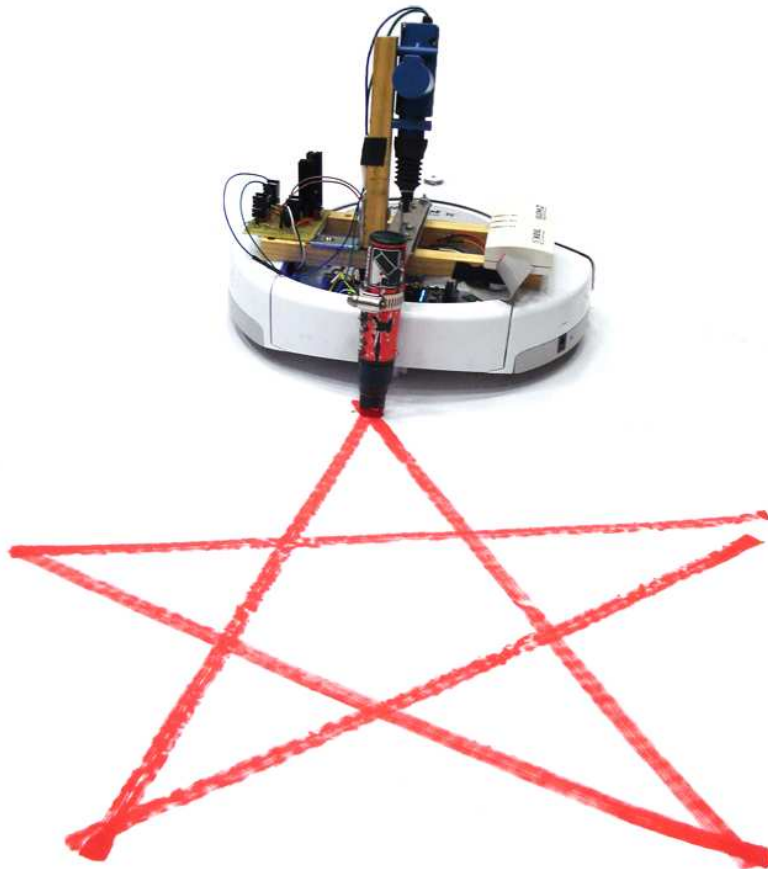
7 Results

We are able to draw basic shapes and have solved many interesting geometric, cyber-physical, and electronic problems while building this robot. On the topic of accuracy, I want to bring up some history. Art Dietrich and Norman Sanders built one of the first computer plotters in the early 1960's. Even a perfect mathematical description of a plane could not be plot accurately with inaccurate hardware. Accurate drawings didn't occur until they used a CAM machine to plot in two dimensions by replacing its head with a diamond scribe.

We had serious inaccuracies emanating from our sensors. The sensors that calculate rotation and distance, thus our tests and update states become offset. This error accumulates and thus drawings can become gradually more ambiguated.

However, in spite of these inaccuracies, we have been able to produce some interesting images and geometric shapes. For example, if you don't send a `STOP` command, the iRobot will repeat its state and you

can produce geometrically repeating shapes. There are also times when the iRobot's sensor are somewhat tame, and has allowed the roomba to generate images with some decent accuracy:



The star drawn by the RoombaWriter. You can view the video on You Tube here:

<http://www.youtube.com/watch?v=MqGHYeGNx5g>

In future iterations we would use an independent sensor for rotations, particularly a compass sensor, which would work very well. In addition, we would want to have better support for curves, ellipses, and spheres. We already worked out a great deal of the mathematics and theory behind the algorithms required to draw these shapes.

Improving the H-bridge design could potentially involve the addition of additional surge protection circuitry, i.e. diodes and/or bypass capacitors placed across the Vdd to ground rails to mitigate power supply noise. The primary enhancement that could be necessary for prolonged circuit use would be to redesign the H-bridge using a MOSFET implementation. Because BJTs are prone to the deleterious effects of thermal runaway, MOSFETs outperform BJTs in high power applications. The design of a MOSFET-based H-bridge is much more complicated than its BJT counterpart primarily due to the large threshold voltages presented by the high-power mosfets and the limited low output voltages of the microcontroller. Our compromise for simplicity meant that we had to endure thermal runaway effects by adding large heatsinks to the transistors.

8 Division of Labor

We divided the labor into three main categories: software, robot, and arm. The software involved bluetooth communication, SVG file parsing, the protocol, simulation, the state machine, and algorithms for plotting. The robot category was divided into power and data storage. The arm category had the actual physical mounting of the robot arm, current sense/feedback, up/down movements of the arm itself, serial communication, H-bridge, and Linear Actuator. Overall, we distributed the work very evenly among our group members.

Andre spearheaded the hardware development for our team. He spent a lot of time working out the math and modified a circuit schematic to have the proper voltage that our robot required. Andre then built an H-Bridge circuit for the linear actuator control from scratch. In addition, he built an interface for controlling the robot arm using LabWindows in our early testing stages. Andre played a major role in developing the serial protocol and designing electronic interfaces.

Dan played an active part on the software development of this project. He began by simulating the roomba traversing along paths with a model of the robot arm. In addition, he helped establish the plotting algorithms, and was able to help parametrize the robots movements using simulation software. Dan also played a major part in developing the state machine and protocol, and in debugging the bluetooth networking issues that came up along the way.

David brought a lot to the table from software to hardware. He was a big part of the code development on the iRobot, in particular, the data storage and bluetooth communication. In addition, David's work on the protocol development and debugging was valuable. David was solely responsible for the code interacting with GPIO pins and the H-Bridge that Andre designed. He also saved us during a major crisis when we overwrote the JTAG pins.

Skot spanned all areas of development. Skot's work ranged from coding the state machine and parsing SVG files, to helping solder circuit boards and connecting GPIO pins. Skot was a key player in the calibration stages of our development, spending many painstaking hours measuring error and applying the appropriate corrections to the code. Skot was also responsible for the Java development and user interface using Processing. Skot also played a key part in the development of the drawing and plotting algorithms.

9 Concepts

Many concepts from the course applied to our project. The iterative design process involving reduction in scope was very useful when we were spending too much time with the original robot arm. Had we not modified our scope, we may not have made as much progress. We decided it was important to just get robot to draw on paper by strapping the pen on the back with no up and down movement.

A major element of our project was the state machine and the models within it. The trace of different states is what strictly defines how the robot can draw an image. Hence, all byte-streams over our bluetooth network were simply traces that we could to define in our JAVA applet.

The entire process can also be represented as a hierarchical state machine. The two highest levels would be a communication state, and a plotting state.

Scheduling was a major element of our embedded code structure. Between polling the iRobot for sensor data, we needed to load commands from memory, update the screen with display information, and control the arm motions.

During the course of the project, it became important to be aware of which operations where atomic during code execution. This helped when debugging embedded system calls, in particular, when the displays calls to the OLED trashed our stack.

10 Feedback

The course was very well taught and the labs were well-designed. However, we feel that once the lab times became project work, time was not as well structured. If perhaps each week during lab (during the project phase of the course), every group brought in a prelab that outlines their milestones for the lab time, the labs would be more optimized. In addition, we thought that it may be good to have a project idea in the beginning of the semester. Then the labs could be extended or catered to the projects, and students could overcome obstacles earlier in the course.