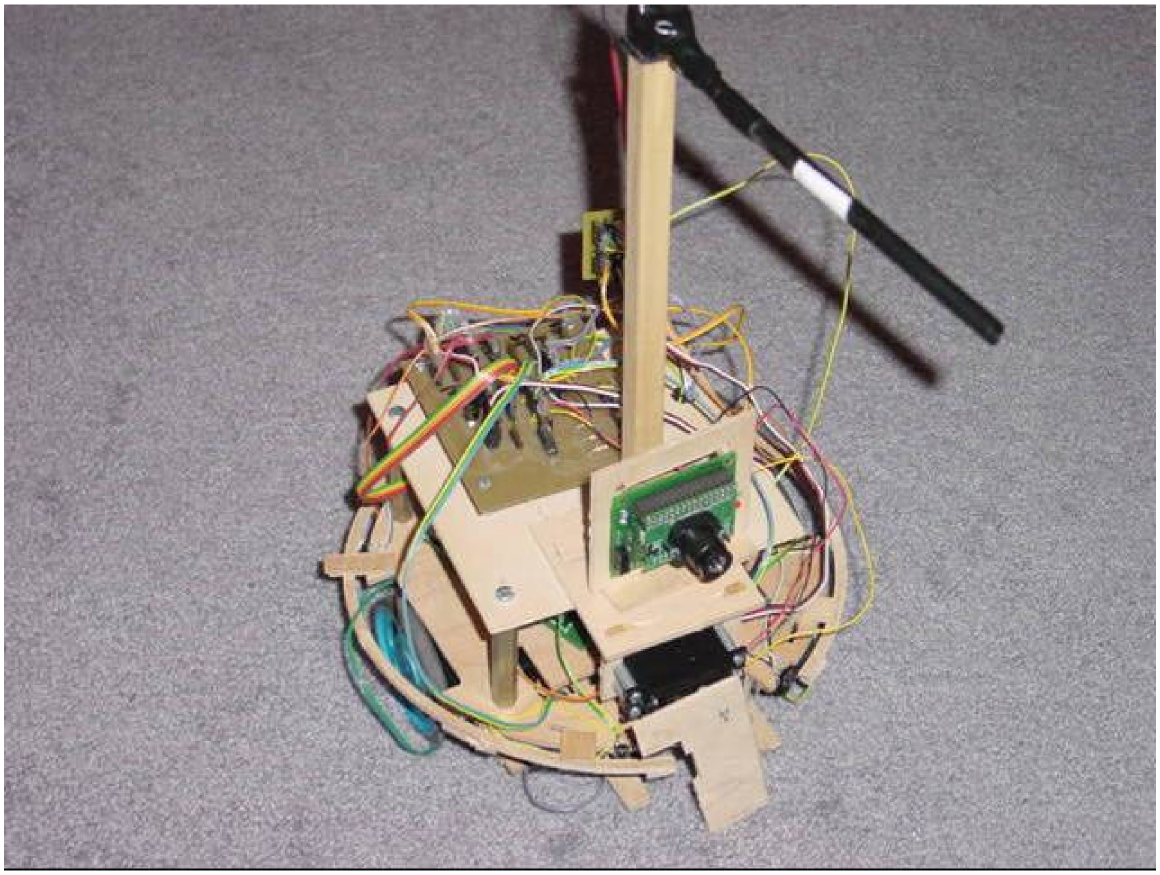
Scratchy: An Autonomous Pool-Playing Robot



James Larson
University of Florida
Department of Electrical and Computer Engineering
Intelligent Machines Design Laboratory
EEL5666 Fall 2002
Dr. A. A. Arroyo

Table of Contents

Abstract

Scratchy, the autonomous pool playing robot, using an array of sensors, actuators, and electronics, finds a pool ball and shoots it towards a pocket. It navigates a simulated pool table, finds the ball, captures it, and shoots it toward a beacon which emulates a pocket.

Executive Summary

Scratchy was designed to be able to find a pool ball on a table and shoot it towards a pocket. It uses the Atmel ATMega323 microcontroller, IR sensors, and a CMUcam to accomplish this goal. The CMUcam turned out to be the most difficult part to use as it was not as effective, nor as easy to use as it was advertised. However, the biggest hurdle for me in this project was my lack of working knowledge coming into this class. I am one of only a few undergraduates in this course and, to my knowledge, am the only one to have not taken Digital Design or Electronic Circuits, both of which would have been invaluable. However, the experience taken from IMDL should make future classes that much easier. I would encourage anyone with the stated prerequisites to take this class, but be forewarned that microprocessors can only take you so far.

Introduction

Getting computers to play games with humans is an interesting proposition. Computers playing games with humans through software has been around for decades. However, only at this point in technology has that intelligence been able to come out of the software realm and enter robotics.

Walk into the ECE computer lab at any given time and you will find several people playing pool – on Yahoo. Pool at the Reitz Union game room is a favorite for many students, especially me. I have decided to combine two of my greatest interests, pool and robotics, into a pool playing robot.

But building a robot that might potentially beat a human opponent is a somewhat scary thought for us arrogant pool players. As a creator, I would not want my creation to become better than me, thus deflating my ego. I have therefore created a pool playing robot that will find a ball and scratch every shot.

In this paper, I will discuss the integrated system of my robot and the platform, then go into further details of the sensors and actuators. This will be followed by the designed behaviors and the experiments which led to its success.

Integrated System

Scratchy uses an array of electronics and systems to interpret and utilize data. At the heart of this system is the AVR Development Board from Progressive Resources, LLC. The development board is based on an Atmel ATMega323 processor, which features 8 10-bit A-to-D channels; 2 8-bit timers and a 16-bit timer, each capable of output compare, input capture, and PWM; on-chip UART; and 32 I/O pins. This is a very powerful and easy to use system that I would recommend to anyone building a robot.

Complementing the Dev Board are several Protel boards which I designed and cut using the T-Tech machines in the lab. These were designed to help easy of connections by having the electronics localized, allowing distribution of power and ground and easing wire and connector construction. These worked well and served as great tutorials to learning Protel, but did not minimize wires as much as I expected.

Mobile Platform

Maintaining the idea of keeping systems simple, Scratchy is based on a simple, circular platform and a differential drive, allowing for the greatest mobility and simplest steering. Areas were originally included to house the batteries, servos, processor board, and additional circuitry. However, these areas became too crammed and many of these features were not realized in full at completion.

The main feature of the lower platform is the ball-catching area, which consists of a semicircular notch cut specifically to the specifications of a pool ball. I needed this area to trap the ball and to house the break beam.

The upper layers of the platform were areas added after the main platform was cut and most were not machined (exception being the CMUcam housing.) I needed these parts cut quickly and did not need anything glamorous, so shoddy workmanship was acceptable.

Sensors

In constructing the sensor suite, I used IR proximity detectors, an IR break beam, bump switches, a CMU cam, and a modulated IR beacon detector. Each of these will be discussed in detail.

For proximity detecting, I used Sharp GP2D12 IR proximity detectors obtained from Mark III. These feature an analog output and an advertised range of 10 to 80 cm. I found these very easy to use and interface, although the connectors were somewhat difficult to connect. I began my project using the GP2D120 IR sensors from Acroname which contained wires which would simply snap into place, making them much easier than the GP2D12's which took some pushing with a screwdriver to ensure connection. However, one of the GP2D120's broke, so I had to replace it as well as buy another. With these being twice the price of the GP2D12, I switched to the latter.

Interfacing either of these connectors proved to be simple. Each contains three pins for VCC, GND, and Vout. I connected the Vout pin to my A-to-D port and the VCC and GND pins and IR sensing was complete. I found this to be much easier than the suggested IR Sharp Can analog hack.

In order to achieve my tasks, I used three proximity detectors to sense the ball. Two GP2D12 sensors were placed outwards used mainly for obstacle avoidance. An additional GP2D120 was placed in the front center pointing straight forward. It should be noted that the GP2D120 has an advertised range of 4 to 30 cm. This will prove advantageous when creating behaviors.

A break beam circuit, as seen in Fig 1, has also been employed in Scratchy's sensor suite. This uses an IR emitter/detector pair obtained from Radio Shack. Using the circuits given in class, a simple break beam circuit was created utilizing a voltage divider circuit on the emitter side.

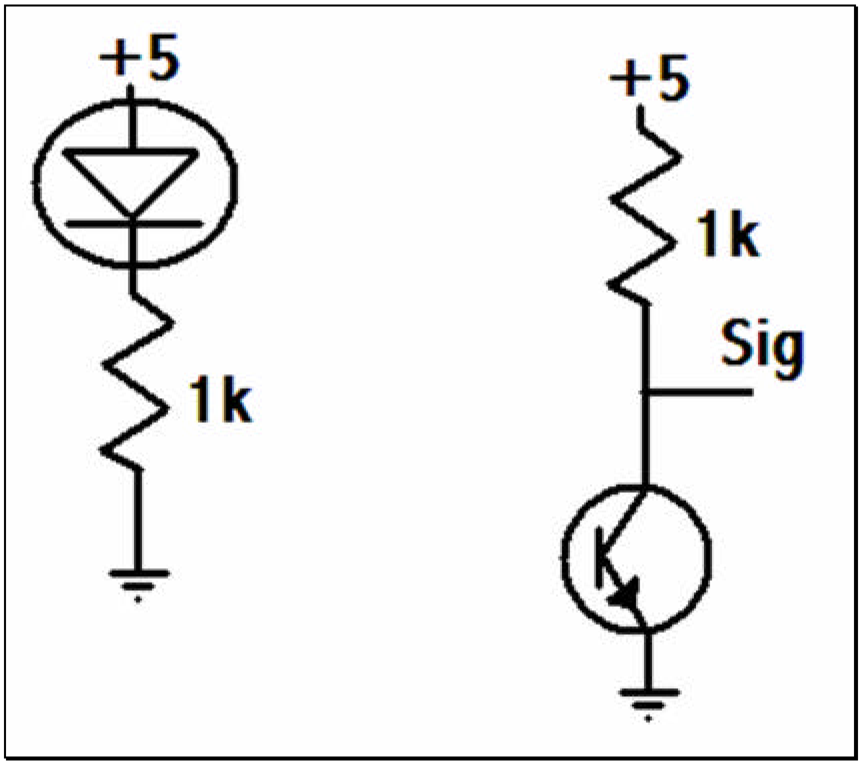


Figure 1 – IR break beam circuit

I found these electronics very easy to use. Using the circuit, I was able to obtain about 4.5 V when the beam is unbroken and 5V when broken. Although this seems close, the 5V reading is exactly 255 from the analog port, so anything less than 255 can be regarded as un unbroken beam. In tests, however, it was determined that between 250 and 255, part of the beam must be broken and in the final program, 250 was the minimum converted analog value.

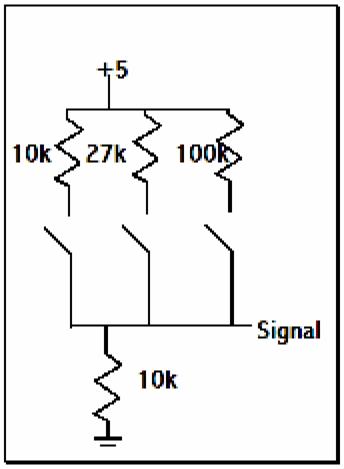


Figure 2 - Bump switch voltage divider network

For bump sensors, I used the button switches given in lab and used the voltage divider circuit given in class and shown in Fig 2. Although some of these voltages were stable enough to use in my final version of Scratchy, I found that the voltage for the back bump sensors was too close to the “unbumped” voltage, so they are unused.

Each of these bump networks were divided further to ensure that any place contacted on the bump skirt would register as a “hit.” Several bump switches were placed in parallel for each area (front left, front right, and back).

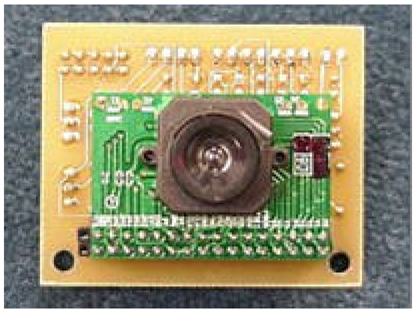


Figure 3 - CMUcam front view

The CMUcam, which I bought from Seattle Robotics, was created by Carnegie Melon University specifically for robotics. However, I have had numerous problems with it. It is advertised as being able to track an object, return the coordinates of the middle of that mass, and dump frames. It runs at 17 fps and can output via TTL or RS-232 connections.

The software that is included leaves much to be desired. The only working demonstration program is a java applet, which would lock my computer after 1 or 2 frame dumps. After installing this software on another computer, it worked better, but the 10 windows it opens is very annoying.

I encountered relatively few problems when interfacing the CMUcam with the computer, but once I tried to interface the CMUcam with my Atmel board, I couldn't seem to fix problems fast enough. I initially tried to go though the TTL connections, but they were flakey at best. After trying dozens of solutions to the dozens of problems I encountered, I eventually determined that the common ground plane between the Atmel board and the CMUcam was acting unusual. I found that I had to power my board up before powering my CMUcam or the two will not communicate.

To find the ball, the code in my robot waits the suggested 2 seconds at startup, turns on white-balance and auto-gain, then turns them off after 2 seconds. At that point, the average color in the center of the view is taken, and this color, plus or minus 10 RGB values, is tracked. I did not use polling mode, but instead I read the M packet as needed and expect the CMUcam to continually send data.

A modulated IR sensor is used to detect a beacon. I used a 38 kHz Everlight IR detector to accomplish this. I found that the 38 kHz frequency made debugging easier because VCR and T.V remotes also use this frequency. I began testing with an unhacked IR detector because I was unsure if this hack would work for the Everlight and was unsure if it was necessary for my case. I determined that the digital output would suffice for my needs, so it was kept for the final design. After a digital pin did not read the values I expected, I connected the digital output to an analog pin and this worked, with 5 V being no signal and a drop (depending on what pulse signals are sent) being beacon found.

Actuation

Two GWS ball bearing high-torque servos, hacked for continuous motion, were used for wheel drivers. These were very cheap servos which easily provided enough power and speed for my robot, but one of the plastic gear-heads broke which Amit was able to replace for me.

An additional unhacked servo was placed above the ball-capture area and serves as a gate which traps the ball while turning.

A solenoid is also employed to strike the ball. This is a S-17-85QH push-type solenoid from www.SolenoidCity.com. I purchased this as a cheap test solenoid to determine if I would have enough power to strike a ball. I initially thought that it would not provide enough power to hit the ball with any velocity, but that was using a power supply in IMDL which I believe did not provide much current. Using an 8-AA battery pack, it can strike a ball well enough to serve my purpose. It does not strike as hard as I was anticipating before starting this project, but as I stated, it was cheap and it performed its task.

I started with a circuit for an opto-isolated solenoid circuit found on the REU website. I found this circuit helpful, but I found it to be not completely helpful in my case. For a week, I believed that either the opto-isolator or mosfet were not working. Having not taken Electronic Circuits, I was unfamiliar with mosfets and transistors and their connections. I finally realized that I connected the source and the drain backwards. As well, in the circuit I found online, I believe it was for a continuous-type solenoid and my robot required a pulse-type solenoid. The original circuit had the resistor connected between the gate and the drain and my circuit has the resistor between the gate and source. The fix is seen in Fig 3.

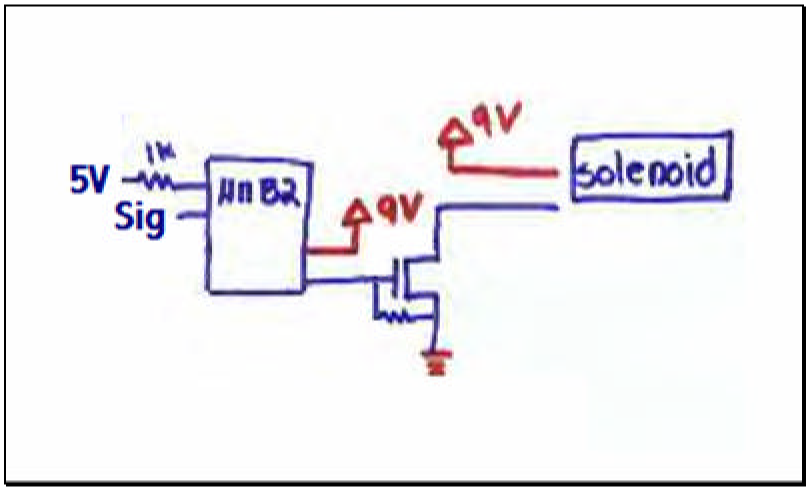


Figure 4 - Modified solenoid circuit

Behaviors

Throughout the operation of my robot, it goes through several stages of behaviors. Mostly, these are searching algorithms, rotating while looking for an object of some sort.

At the beginning of Scratchy's operation, he begins by searching for the ball using the CMUcam. During initialization, the CMUcam is set up to track a red object. Because the background used is bright green and well-lit, the red is very easy to track. Using the demo program which was included with the CMUcam, I found that red values on the ball varied between 90 and 200, green varied from 0 to 50, and blue varied from 0 to 60. Tracking these color ranges, I was able to filter out the green portions and track only the red ball.

In middle mass mode, the CMUcam returns the center of the object. When the returned "mx" value is within the center 10 pixels of the view, Scratchy begins moving forward slowly. As long as the ball is located in the center of the camera's view, Scratchy "crawls" forward. If the ball moves from the center of the view, he will rotate until it is located in the center again. This process continues until the ball is found in the bottom of the view.

When the ball is close enough in front of Scratchy, he will start relying on his IR sensors and break beam. The center IR sensor determines when the ball is very close and his movement is slowed. At this point, he moves forward until his break beam breaks, thus determining that the ball has moved into the catching area.

Once the ball is in the catching area, the gate is dropped and the search for the IR beacon begins. This is performed in much the same way as the search for the ball. Scratchy rotates at a “crawl” rate and continually reads the analog value connected to the Everlight IR detector. Once this value reflects a found beacon, Scratchy stops. The gate is then lifted and the solenoid is fired.

Experimental Layout

Most of the sensor and actuators on Scratchy did not require precise measurements, so large-scale experiments to find exact values were unnecessary. However, most did require some fine tuning, the extents of which will be described below.

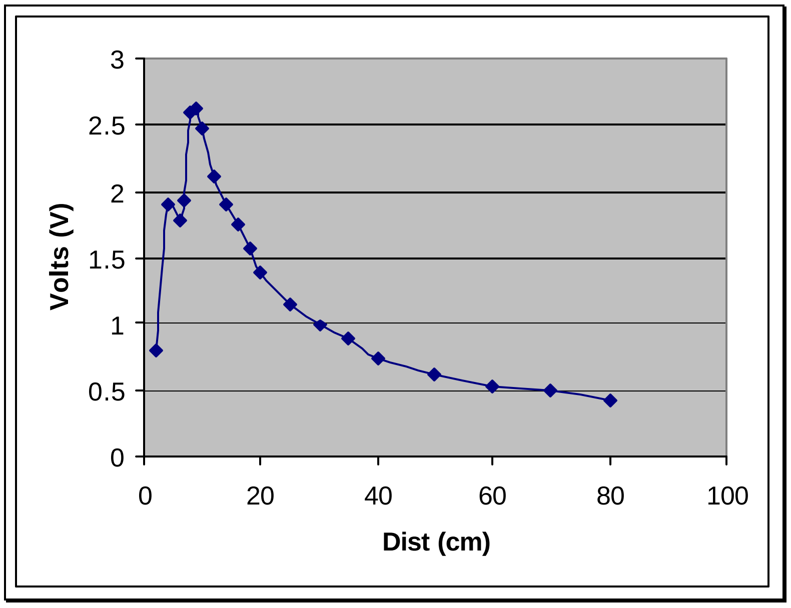


Figure 5 - Analog voltages of IR at varying distances

IR sensors were tested using the A-to-D port and LED's. A simple program was written to output the digital value of the IR to the onboard LED's. It was determined that a minimum value of 225 should be used to determine if an object was approaching. This value was chosen because it was close enough to guarantee that a collision was inevitable. Values larger than this could have allowed for an "overshoot" whereby the robot would move too fast for the A-to-D and the values would bypass the range between that minimum and 255.

The IR break beam was tested in much the same way as the IR range sensors. The output of the circuit was connected to the analog port and values were read at the different states, broken and unbroken. It was found that each were very steady, but the broken beam jumped to 255 when it was broken and much smaller when unbroken. I then added a 5-value padding and use any value greater than 250 as a broken beam. This was tested using the LED's by lighting the LED's when broken and keeping them unlit when unbroken and it yielded perfect results.

Bump sensors were initially tested using a multimeter and converting those voltage values to digital values for the A-to-D values. This, however, did not achieve reliable results. A program was written which lit the LED's when the A-to-D values reached around these voltages. The LED's did not light when they should have.

After that failure, trial and error yielded more desirable results. The test program was able to accurately predict which bump switch was pressed. However, as time went on, this system did not work as well as desired – another system was necessary.

Finally, a self-calibrating system was implemented. This system worked by waiting for the analog value to change. Once this value changed, a button must have been pressed and this value was recorded. The program then waited until the current value varied from the pressed value. This sequence continued for each set of bumps. However, the back bumps had values too close of values to the unpressed value, although I am unsure why.

Ultimately, I chose not to use the back bump switches since my robot never moves backwards.

Testing the IR beacon detector was another easy task. The hardware acted somewhat unusual at first when the digital port never went low when the Everlight was connected to it, but much better results occurred when connected to the analog port. Whenever the value was less than 250, the LED's were lit. I used a DVD player remote to emulate a 38 kHz signal. Due to a highly collimated detector (approximately 8 inches long), the detector was able to find the DVD remote within approximately 10 degrees, which was precise enough for my use.

The CMUcam took more tweaking than anything. Experiments were mostly frame dumps and tests on the computer. What these experiments did reveal, however, was that the CMUcam is very sensitive to IR interference and needs a strong source of light.

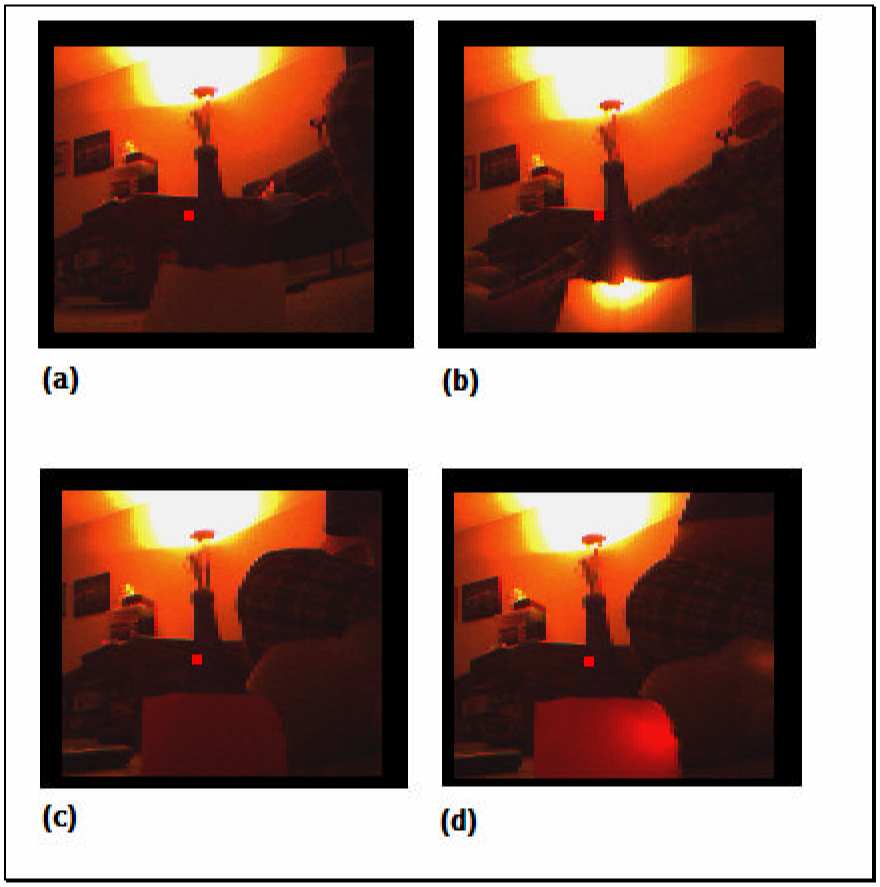


Figure 6 - White piece of paper (a,b) and red piece of paper (c,d) with and without light

It was eventually found that a well-lit green background with a red ball yielded a crisp picture which most easily allowed for color tracking. Sample color values were taken from frame dumps and color ranges for the ball were determined.

Conclusion

This was one of the most informative experiences of my life. After completing Microprocessor Applications at the end of Summer, I thought that IMDL would be mostly an applied Microprocessors class. I was quite wrong.

One of the main things I learned is that no matter how much reason you have to believe that something will work a certain way, there's no way to tell until you connect wires to it and find out. Nothing works exactly as stated and nothing is exact.

I also learned that no matter how much you try to plan for every scenario, another will come up that you didn't think of. Many of my hardware designs tried to take into account what would be added at a later time. Much of my software design eliminated as many bugs as possible. Each had issues come up that I could have never thought of.

The main thing I would change about Scratchy would be a complete redesign of the body. The lower area had no good place for batteries to stay, nor for the solenoid or its electronics. Many of the electronic boards could be better redesigned to eliminate duplication of parts and reduce overall size.

Overall, however, I was very satisfied with the outcome of this project. It was able to find a ball and shoot it into a pocket. For future work, if a better camera was used, shooting to make a ball (as opposed to the current "scratch" that this version of Scratchy shoots) could be possible and a very interesting addition. However, my current knowledge limits the amount I could expand this robot.

Appendix A: Code

```
#include <io.h>
#include <interrupt.h>
#include <sig-avr.h>
#include "\imdl\include\motor.h"
//#include "\imdl\include\uart.h"
#include "\imdl\include\ad.h"
#include "\imdl\include\bumpcal.h"

#ifndef IR_TOLERANCE
#define IR_TOLERANCE 3
#endif

#define STATE_FIND_BALL 1
#define STATE_WANDER 2
#define STATE_AVOID 3
#define STATE_CRAWL 4
#define STATE_GRAB 5
#define STATE_SEARCH 6
#define STATE_RELEASE 7
#define STATE_FIRE 8
#define STATE_RETREAT_LEFT 9
#define STATE_RETREAT_RIGHT 10


#define WALK_BACK_DELAY 4
#define WALK_TURN_DELAY 3
#define CRAWL_BACK_DELAY 3
#define CRAWL_TURN_DELAY 1


int irr,irl,irc,bb,state=STATE_WANDER;
int beaconval,beaconcal;
unsigned char packetType;

// M packet
unsigned char mx,my,x1,y1,x2,y2,pixels,conf;

unsigned char minx,miny,maxx,maxy;

void tempdelay(void);
void checkBump(void);
void grab(void);
void release(void);
void avoid(void);
void getADvals(void);
int ballFound(void);
void wander(void);
void retreatRight(int backDelay,int turnDelay);
void retreatLeft(int backDelay,int turnDelay);
void crawl(void);
void stop(void);
void search(void);
void fire(void);
void getMpacket(void);
void initCMU(void);




void tempdelay(void)
{
int i,j;

for (i=0; i<255;i++)
        {
        for (j=0;j<255;j++)
        {}

        for (j=0;j<250;j++)
        {}
        }
}

void getADvals(void)
{
        irl = getLeftIR();
        irr = getRightIR();
        irc = getMiddleIR();
        bb = getBBval();
        beaconval = getBeacon();
}

void avoid(void)
{
        PORTC = ~state;
        if ((irl > 225) && (state != STATE_CRAWL))
        {
                drive_right();

                tempdelay();
                tempdelay();
                tempdelay();
                tempdelay();
                drive_forward();
                tempdelay();
        }

        else if ((irr > 225) && (state != STATE_CRAWL))
        {
                drive_left();

                tempdelay();
                tempdelay();
                tempdelay();
                tempdelay();
                drive_forward();
                tempdelay();
        }
```

```
	else
		state=STATE_WANDER;
}

int ballFound(void)
{
	if ((irc > 225) && (state == STATE_WANDER))
	{
		return 1;
	}

	return 0;
}


void wander(void)
{
	state = STATE_WANDER;
	PORTC = ~state;
	while (state == STATE_WANDER)
	{
	PORTC = ~state;
		getADvals();
		avoid();

		if (ballFound())
			state = STATE_CRAWL;

		checkBump();
		if (state == STATE_RETREAT_LEFT)
		{
			retreatLeft(WALK_BACK_DELAY,WALK_TURN_DELAY);
			state = STATE_WANDER;
		}
		if (state == STATE_RETREAT_RIGHT)
		{
			retreatRight(WALK_BACK_DELAY,WALK_TURN_DELAY);
			state = STATE_WANDER;
		}

	}
}


void retreatRight(int backDelay,int turnDelay)
{
	int i;

	PORTC = ~state;
	saveSpeed();
	drive_backward();
	for(i=0;i<backDelay;i++)
		tempdelay();
	crawl_right();
	for(i=0;i<turnDelay;i++)
		tempdelay();
	restoreSpeed();
}


void retreatLeft(int backDelay,int turnDelay)
{
	int i;

	PORTC = ~state;
	saveSpeed();
	drive_backward();
	for(i=0;i<backDelay;i++)
		tempdelay();
	crawl_left();
	for(i=0;i<turnDelay;i++)
		tempdelay();
	restoreSpeed();
}


void checkBump(void)
{
// applies bump values to determine if turning is necessary
/*if (hitLeft())
	{
	state = STATE_RETREAT_RIGHT;
	}
if (hitRight())
	{
	state = STATE_RETREAT_LEFT;
	}*/
}



void crawl(void)
{
	PORTC = ~state;
	crawl_forward();
	state = STATE_CRAWL;

	// because of the bb mysteriously jumping high
	tempdelay();

	while (state == STATE_CRAWL)
	{
		PORTC = ~state;
		getBump();
		checkBump();
		if (state == STATE_RETREAT_LEFT)
		{
			retreatLeft(CRAWL_BACK_DELAY,CRAWL_TURN_DELAY);
			state = STATE_CRAWL;
		}
		if (state == STATE_RETREAT_RIGHT)
		{
```

```
                        retreatRight(CRAWL_BACK_DELAY,CRAWL_TURN_DELAY);
                        state = STATE_CRAWL;
                }
                bb = getBBval();
                if (bb > 250)
                {
                        state = STATE_GRAB;
                }
        }
}


void grab(void)
{
        gate_down();
        PORTC = ~state;
}

void stop(void)
{
        drive_servos_stop();
        drive_nowhere();
        PORTC = ~state;
}



void release(void)
{
        PORTC = ~state;
        gate_up();
}


void search(void)
{
        drive_servos_start();
        crawl_left();
        tempdelay();
        state = STATE_SEARCH;
        PORTC = ~state;
        while(state == STATE_SEARCH)
        {
//              PORTC = ~state;
                irr = getBeacon();
                PORTC = ~irr;
                if((irr < (beaconval - IR_TOLERANCE)) || (irr > (beaconval + IR_TOLERANCE)))
                        state = STATE_RELEASE;
        }
        drive_servos_stop();
        drive_nowhere();
}


void fire(void)
{
        int b;
        // clear bit 0 of PORT B, thus firing
        PORTC = ~state;
        b = PORTB;
        PORTB = b & 0xFE;
//      cbi(PORTB,0);
        tempdelay();
        tempdelay();
        tempdelay();
        tempdelay();
        tempdelay();
        tempdelay();
        tempdelay();
        tempdelay();
        tempdelay();
        tempdelay();
        tempdelay();
        tempdelay();
        tempdelay();
        tempdelay();
        // stop firing
        PORTB = b | 0x01;
//      sbi(PORTB,0);
}



void getMpacket(void)
{
        do{
                packetType = getChar();
        } while (packetType != 'M');
        flushChar();
        mx = getChar();
        flushChar();
        my = getChar();
        flushChar();
        x1 = getChar();
        flushChar();
        y1 = getChar();
        flushChar();
        x2 = getChar();
        flushChar();
        y2 = getChar();
        flushChar();
        pixels = getChar();
        flushChar();
        conf = getChar();
}


void initCMU(void)
{
        tempdelay();
        uart_send("L1 1\r",5);
        tempdelay();
```

```
		tempdelay();
		tempdelay();
		tempdelay();
		uart_send("L1 0\r",5);
		tempdelay();
		tempdelay();
		tempdelay();
		tempdelay();
		uart_send("L1 2\r",5);
		tempdelay();
		tempdelay();
		tempdelay();
		tempdelay();
//		uart_send("PM 1\r",5);
		tempdelay();
		tempdelay();
		tempdelay();
		tempdelay();
//		uart_send("SW 1 60 80 143\R",15);
		tempdelay();
		tempdelay();
		tempdelay();
		tempdelay();
		uart_send("RM 3\r",5);
		PORTC = 0x55;
		tempdelay();
		tempdelay();
		tempdelay();
		tempdelay();
		uart_send("TC 90 200 0 50 0 60\r",20);
		tempdelay();
		tempdelay();
		tempdelay();
		tempdelay();

}


void findBall(void)
{
		state = STATE_FIND_BALL;
		getMpacket();
		if ((mx < maxx) && (mx > minx))
		{
				state = STATE_WANDER;
		}
		else
		{
				crawl_left();
		}
		while(state == STATE_FIND_BALL)
		{
				getMpacket();
				if ((mx < maxx) && (mx > minx))
				{
						state = STATE_WANDER;
```

```
                }
        }
        drive_forward();
}


#define BUMP_TOLERANCE 7 //  value each way will correspond to a "hit"
#define NONE_TOLERANCE 5 //  tolerance for no bumps being pressed


int notouch,left,right,back;


void delay2(int delay_time) {
   do {
     int i=0;
     do {
     asm volatile("nop\n\t"
         ::);
     } while(--i);
   } while(--delay_time);
}

void delay_ms(int dt)
{
// roughly 3 delays per ms
        delay(dt);
        delay(dt);
        delay(dt);
}

void waitADval(int value)
{
// waits for the current IR range to cease
        int ad = value;

        while ((ad > (value - BUMP_TOLERANCE)) && (ad < (value + BUMP_TOLERANCE)))
        {
                ad = getBump();
        }
}


int hitRight(void)
{
        int bmp = getBump();
        if ((bmp > (right - BUMP_TOLERANCE)) && (bmp < (right + BUMP_TOLERANCE)))
                return 1;
        else
                return 0;
}

int hitLeft(void)
{
        int bmp = getBump();
        if ((bmp > (left - BUMP_TOLERANCE)) && (bmp < (left + BUMP_TOLERANCE)))
```

```
                return 1;
        else
                return 0;
}

int hitBack(void)
{
        int bmp = getBump();
        if ((bmp > (back - BUMP_TOLERANCE)) && (bmp < (back + BUMP_TOLERANCE)))
                return 1;
        else
                return 0;
}

void calibrateBumps(void)
{
        notouch = getBump();

        // turn on farthest right LED (wrt facing forward)
        PORTC = 0x7F;

        // wait for something to happen
        waitADval(notouch);
        right = getBump();

        PORTC = 0xFF;
        // light farthest left left LED
        PORTC = 0xFE;
        waitADval(right);
        waitADval(notouch);
        left = getBump();
        PORTC = 0xFF;
}



int main(void)
{

        outp(0xFF,DDRC);
        PORTC = 0x99;
        DDRA = 0x00;
        DDRB = 0x09; // mainly for solenoid out & gate

        initAD();
        PORTC = 0xFF;
        tempdelay();
        tempdelay();
        tempdelay();
        tempdelay();


        DDRD = 0x02;

        uart_init();
        sei();
```

```
	echoOff();

	// right, then left
	calibrateBumps();
	initCMU();
	PORTC = 0x0F;
	getMpacket();
	minx = mx - 5;
	maxx = mx + 5;
	miny = my - 5;
	maxy = my + 5;
	tempdelay();
	tempdelay();
	tempdelay();
	tempdelay();

	delay(100);
	PORTC = 0xFF;

	// calibration value
	beaconval=getBeacon();

	motor_init();
	PORTC = 0x33;

	while(1)
	{
		findBall();
		wander();
		crawl();
		stop();
		grab();
		delay(100);
		search();
		release();
		delay(100);
		fire();
	}
}
```