University of Florida
Department of Electrical Engineering

EEL 5666
Intelligent Machines Design Laboratory

Final Report

Autonomous Tile-Laying Robot

By: Alan Senior
December 9, 1996
senior@cimar.me.ufl.edu
(352) 338-2946

My autonomous robot, called TileBot, is designed to dispense square tiles according to an image stored in memory while avoiding obstacles. Stepper motors were used to provide accurate tracking and tile placement on a smooth surface.

Filename: tilebot.doc

TABLE OF CONTENTS

page

ABSTRACT

My autonomous robot, called TileBot, is designed to dispense square tiles according to an image stored in memory while avoiding obstacles. Stepper motors were used to provide accurate tracking and tile placement on a smooth surface. Behaviors include stopping in the presence of an obstacle until the obstacle can be cleared. The vehicle also stops if it runs out of tiles, allowing the user to re-fill the tiles and press the START button to resume processing. Obstacle avoidance, as well as tile detection, is performed using infrared sensors. Infrared emitting diodes are modulated at 40KHz, a frequency that can be detected by the detectors when it is reflected back to the robot by an obstacle. In addition to the infrared sensors, bump sensors are installed to detect obstacles that are not seen by the IR. Tiles are dispensed using a gravity-fed system in which approximately 800 tiles are stacked vertically. The tiles are dispensed one at a time as needed by the use of a servo under control of the microprocessor. The dispenser and power train are constructed entirely of aluminum which was custom designed and machined by the author to provide smooth, repeatable dispensing and accurate tracking.

EXECUTIVE SUMMARY

The goals in mind during the development of this robot were simple: 1) Autonomously dispense square tiles according to an image stored in memory while interacting with the environment through various sensors, actuators, and behaviors. 2) Design the power train and dispenser to provide smooth, repeatable dispensing and accurate tracking.

Three behaviors were developed for TileBot. The first and most important is tile dispensing. This is a simple behavior in which tiles are pushed onto the driving surface at pre-determined locations to form an image. The second behavior is obstacle avoidance. This behavior depends on feedback from six infrared emitter/detector pairs and six bump sensors mounted along the front and back of the vehicle. If TileBot encounters an obstacle, it will halt processing and wait for the obstacle to clear. Attempts to go around or otherwise avoid the obstacle would defeat the purpose of maintaining accurate parallel tracking for tile placement. The third behavior is associated with the tile dispenser. In the event that TileBot should run out of tiles, an infrared sensor mounted on the dispenser will detect the absence of tiles and halt processing until the user re-fills the dispenser and presses the START button. Once the dispenser is re-filled, TileBot will resume processing where it left off before the interruption.

TileBot consists of dual stepper motors mounted on a plexi-glass platform, with a tile dispenser mounted to one side of the vehicle. This allows TileBot to maneuver by counting the number of steps in a given direction, and dispense tiles when appropriate. The tracking accuracy of stepper motors far exceeds the capabilities of conventional motor/encoder combinations that are used on

most small robots of this type. The drawbacks are a much higher power consumption required by the stepper motors, and a greatly reduced driving torque (compared to conventional motors). TileBot will only run over smooth, level surfaces that are free of obstructions. This is not much of a constraint considering the intended function of TileBot.

INTRODUCTION

In this paper I will be describing some of the features and behaviors of TileBot. The purpose of this project was to create an autonomous mobile robot, capable of interacting with its environment through various sensors, actuators, and behaviors. TileBot, is designed to dispense fiberglass tiles according to an image stored in memory while avoiding obstacles. A robust combination of sensors and appropriate behaviors allow TileBot to perform its tasks.

TileBot consists of dual stepper motors mounted on a plexi-glass platform along with the power supply, microprocessors, motor controller, tile dispenser, and sensors. Plexi glass has proven to be lightweight and extremely easy to work with. The one drawback is its flexibility, which has presented minor problems that have been overcome by re-enforcing the platform. The dispensing system is mounted to the base of the robot. This system dispenses tiles at the request of TileBot's program.

This paper covers the complete organization of the system. The mobile platform is described, as well as the actuation of the stepper motors and tile dispenser. The sensors used for tile detection

and obstacle avoidance, as well as the behaviors associated with these sensors are explained. Several examples of TileBot's accomplishments are provided, along with possibilities for future work.

INTEGRATED SYSTEM

Figure 1 shows TileBot as it looks today. The vertical stack measures 12 inches high and holds approximately 800 fiberglass tiles. Notice that the dispenser is mounted to one side of the vehicle. This allows TileBot to dispense tiles outside the path of the wheels therefore eliminating the need for TileBot to roll over an area in which tiles have been placed. There are six infrared emitter/detector pairs which "look" out horizontally from the base of the robot.

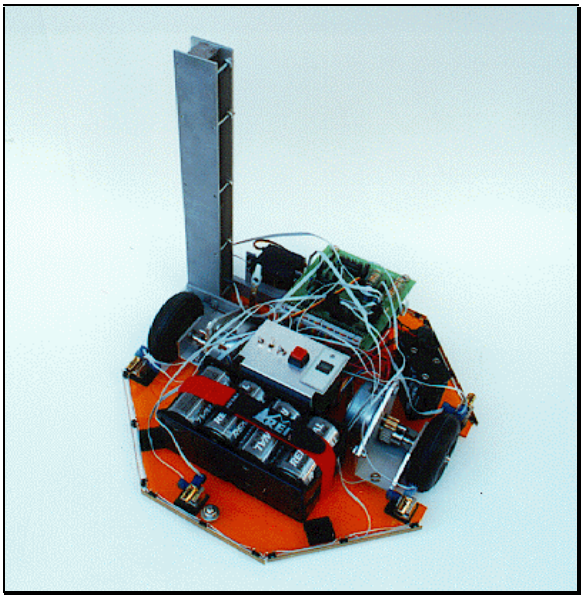

Figure 1: TileBot

Three "look" forward, while the other three "look" backward. A set of six bump sensors are also mounted on the forward and backward edges of the platform. There is a control panel with three switches, a reset button, and a START button. The function of each of the three switches is as follows: 1) Power, 2) Low Power Standby, 3) Boot/Run Mode

The low power standby switch allows TileBot to retain it's memory while expending a minimum amount of battery power. The boot/run mode switch is used during initialization of the

interactive C (IC) operating system (Mar94).

MOBILE PLATFORM

I wanted the platform to be clean-looking, straightforward, and lightweight. The platform is made of clear, amber-colored plexi-glass. It is octagonally shaped and measures 12 inches across. In the center of this base, stacked vertically on each other, the Motorola 68HC11 EVBU board and the Novasoft expansion boards are mounted. An 8-pack AA-Cell pack is mounted to the side of the microprocessor. This powers the electronics. Two 4-pack D-Cell packs are mounted to the rear of the vehicle. These power only the stepper motors and are isolated from the microprocessor power supply to reduce electrical noise generated by the motors. A servo is mounted on the dispenser to the left of the EVBU board. This servo is used to dispense tiles. The robot easily supports the motors, gear boxes, 16 batteries (8 D-Cells and 8 AA-Cells), dispenser, and microprocessor. Since it is designed to have a low profile, steel balls have been mounted in place of casters to hold the platform level on the two wheels. These will simply slide along with the robot on a smooth surface. Unfortunately, the combination of hardware and batteries does not make a very lightweight robot. This problem adds to the high power consumption of TileBot.

ACTUATION

Stepper Motors

Figure 2 shows a detail of one of the stepper motor/gear box combinations. The 12V stepper motors and gears are salvaged printer parts. The gear housing and stepped shaft to which the wheel is mounted were designed and machined by the builder. Ball bearings have been installed at either end of the drive shaft in order to reduce friction. This arrangement provides a 1:2 gear reduction from the motor to the wheels, reducing the 15 degree step angle of the motor to 7.5 degrees at the wheels. Using 3" wheels, this translates to a motion of 0.196" per step. The use of stepper motors has the advantage of allowing very accurate positioning of the robot, when compared to a DC motor/encoder arrangement. The disadvantages include added weight, greatly reduced torque (especially at high speeds), and high power consumption. In addition, without the use of an encoder there is no way to know whether the motor is actually rotating or not. On a smooth surface with charged batteries, this is not a problem. The direction and speed of rotation of a stepper motor is determined by the speed and order that the coils are energized. Fearing that constant calculation required to accomplish this may overburden the processor and cause erratic, unpredictable motion, I built the dual stepper motor controller pictured in figure 3:

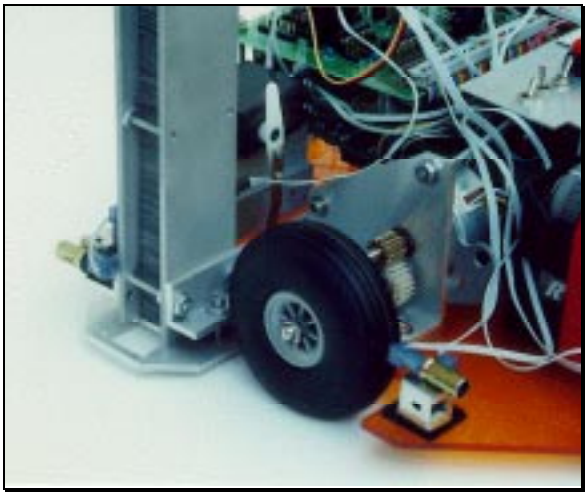

Figure 2: Gearbox/Dispenser Detail

This transfers the burden of synchronizing the motor coils to the hardware, thus greatly simplifying the task of programming. This controller also allows for separate motor and control circuit power supplies. The stepper motor controller circuit schematic is detailed in Figure 4:

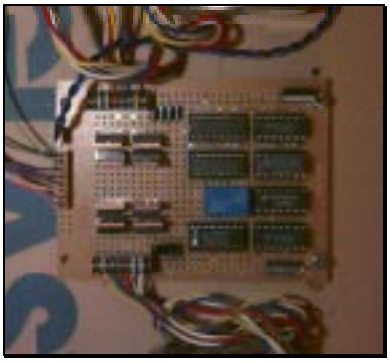

Figure 3: Motor Controller

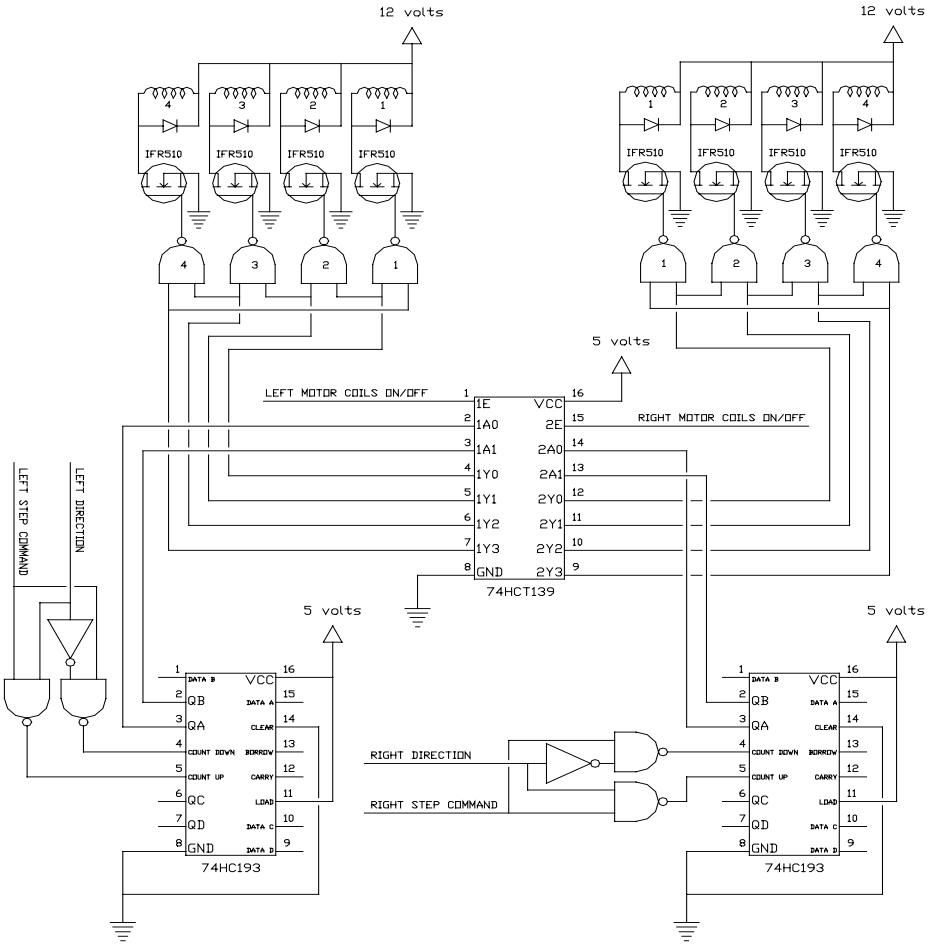

Figure 4: Dual Stepper Motor Controller Schematic

The functions and pin out descriptions of the 74HC193 counter were acquired from a National Semiconductor data sheet (Nat89). MOSFET properties were referenced in Getting Started in Electronics (Mim83). Notice that diodes protect the MOSFETS from overload. This protection is needed because the winding on a stator pole is an inductor, which tries to keep the same current flowing just after the MOSFET turns off. This produces high transient voltages across the MOSFET that can burn it out (Sar95). Table 1 illustrates the controller input bits:

Table 1: Stepper Motor Controller Input Bits

1: 0 to 1 Transition - Step Left Motor
2: Left Motor Move Forward/Reverse
3: Left Motor Coils On/Off
4: 0 to 1 Transition - Step Right Motor
5: Right Motor Move Forward/Reverse
6: Right Motor Coils On/Off

Dispensing System

The dispensing system was designed to dispense black tiles that measure 0.80in x 0.70in x 0.15in thick. Square tiles would have been ideal, but perfectly square tiles were unavailable. Figure 2 shows the bottom region of the dispensing system. The base of the robot holds the dispenser about 0.25in from the driving surface. Notice the square hole in the bottom plate of the dispenser. A tile that is pushed from the bottom of the stack by the servo enters this hole before it drops to the ground. This helps to assure that the tiles will drop uniformly with as little rotation as possible. Originally, a push-type solenoid was used to dispense tiles. This method was far too weak, worked inconsistently, and consumed much power. Therefore, a servo was used in conjunction with an infra-red (IR) sensor dispense tiles one at a time from a vertical stack

of approximately 800 tiles. The entire stack of tiles measures 12 inches high. The IR sensor assures that the dispenser contains tiles before a dispense occurs. The dispenser is constructed entirely of aluminum which was custom machined by the author to provide smooth, repeatable dispensing. The servo used to dispense tiles can be seen in Figure 2. This servo is connected to five AA sized cells that are on the platform. It is controlled by the EVBU via an Interactive C routine named SERVO.C. This routine allows the user to input a servo angle (in degrees) and commands the servo to that position by varying the timing of an electrical pulse sent to the servo. The servo is connected to a three-bar mechanism. This mechanism is detailed in Figure 5:

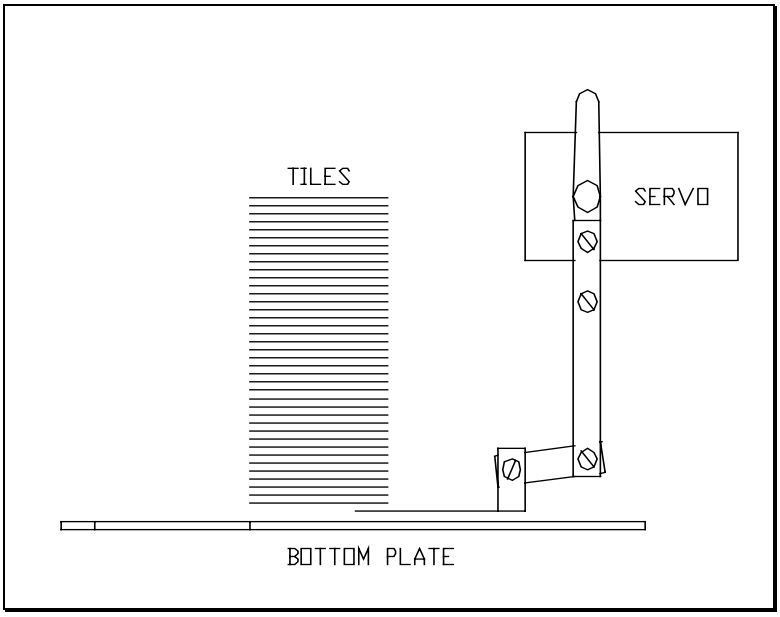

Figure 5: Three Bar Dispenser Mechanism

The length of the bar attached to the servo allows a large travel with a small change in servo angle. This feature helps to speed the dispensing process. In order to dispense a tile, the dispenser must first retract and allow the stack of tiles to drop down (Figure 6).

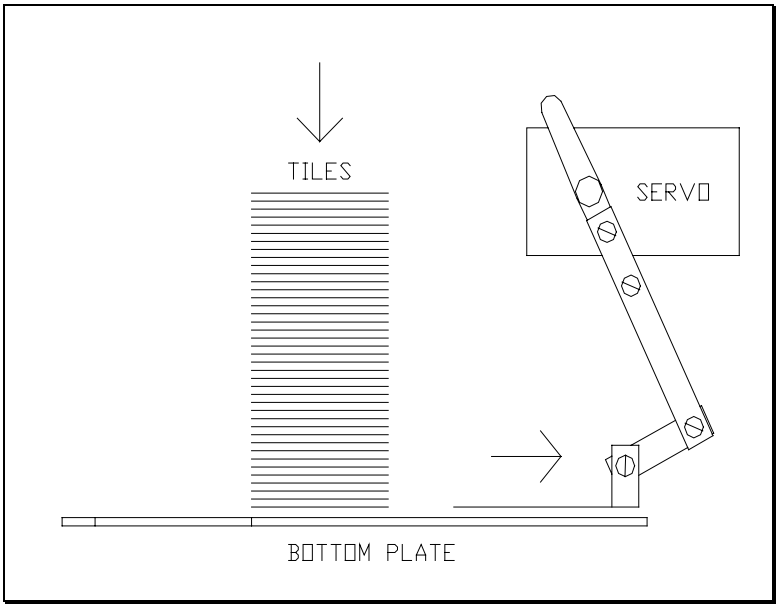

Figure 6: Dispenser Retract Cycle

Next, after ample time has been given for the servo to complete the first move, the servo is commanded forward. At this time, a tile is pushed from the bottom of the stack and falls through the hole in the bottom plate (Figure 7).

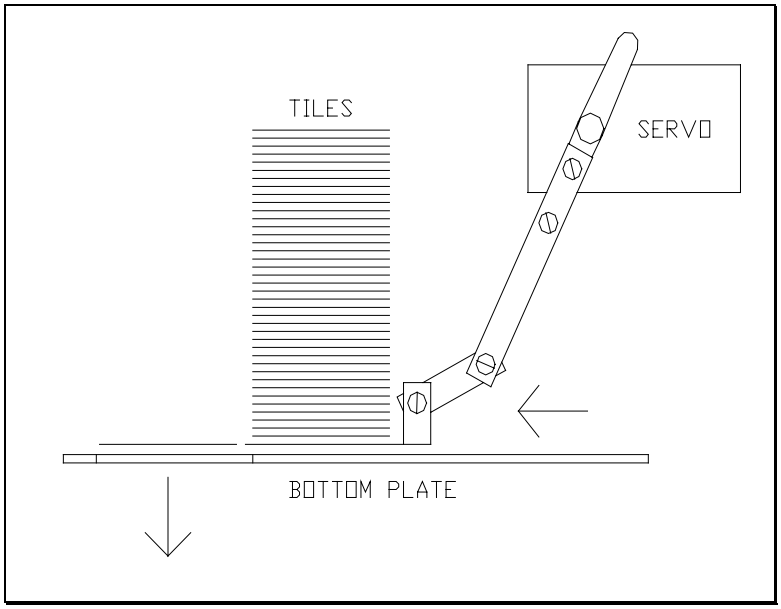

Figure 7: Dispenser Dispense Cycle

The entire process of dispensing a tile takes about 1/3 second. Therefore, TileBot is capable of dispensing up to three tiles per second, without stopping while dispensing.

SENSORS

Infrared Emitter/Detector Pairs

Infrared Light Emitting Diodes (LEDs) emit infrared light to illuminate objects nearby (modulated at 40KHz). The detector used is a Sharp GP1U58Y, which only detects infrared light modulated at a rate of 40KHz. This sensor contains an infrared sensor, amplifier, 40KHz filter, and a comparator. Once the sensor has been modified (Jan96), it outputs an analog signal corresponding to the intensity of infrared light detected. In unmodified form, only a digital 0/1 is output in the absence/presence of infrared light. TileBot considers only the sensors that point in the current direction of travel, except in the case where TileBot is turning. In this case, all sensors are considered. A similar setup will be used in the tile dispensing unit to determine the presence/absence of tiles. The only sensor associated with the dispenser is an IR emitter/detector pair that "looks" at the bottom of the stack of tiles to determine is there are tiles there. This sensor is highlighted in Figure 8.

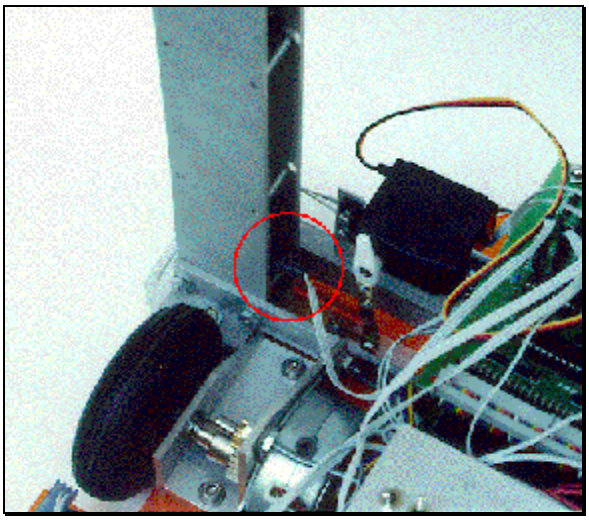

Figure 8: IR Sensor

Bump Sensors

An array of bump sensors surround the vehicle to determine if the vehicle has hit an obstacle. The vehicle will then stop until the obstacle has cleared. Any attempt to go around an obstacle would be futile as it would ruin the pattern TileBot is trying to create.

BEHAVIORS

Tile Dispensing

The placement of tiles is determined by the use of a matrix of 0's and 1's (1 indicates the location of a tile). As TileBot traverses the driving surface, it reads this matrix and dispenses tiles where appropriate (see Figure 9).

TileBot Input:

0,0,1,1,1,0,0
0,1,0,0,0,1,0
1,0,0,0,0,0,1
1,0,1,0,1,0,1
1,0,0,1,0,0,1
0,1,0,0,0,1,0
0,1,0,1,0,1,0
0,1,1,1,1,1,0
0,1,0,1,0,1,0

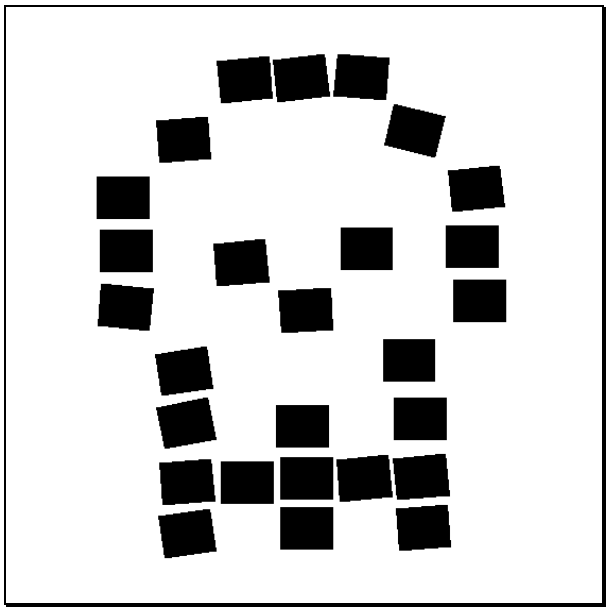

Figure 9: Scary Halloween Mosaic

Tile dispensing is accomplished by running TileBot from side to side, advancing down one tile height with each sweep. As TileBot traverses from side to side, it dispenses tiles over the side of the vehicle where required. This assures that dispensed tiles are kept clear of the wheels. See Figures 10 and 11:

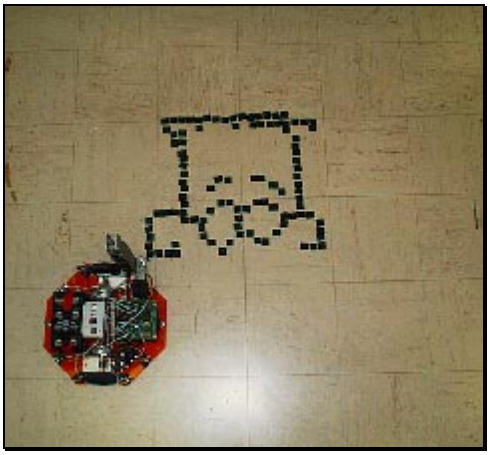

Figure 10: Traversing Image

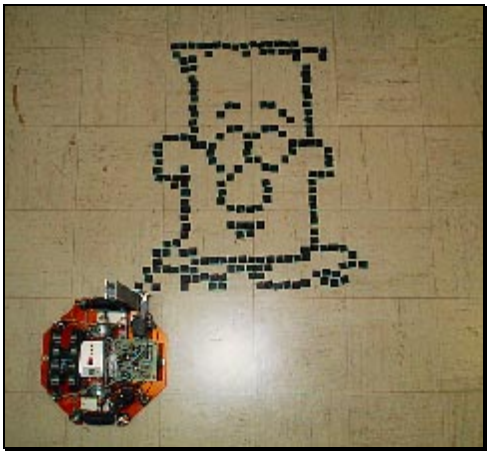

Figure 11: Completed Image

Obstacle Avoidance

This behavior depends on feedback from six infrared emitter/detector pairs and six bump sensors mounted along the front and back of the vehicle. The range of the IR sensors varies depending on the size and reflectivity of the obstacle. They are calibrated to detect most objects that are less than 10" from the vehicle. If TileBot encounters an obstacle, it will halt processing and wait for the obstacle to clear. Attempts to go around or otherwise avoid the obstacle would defeat the purpose of maintaining accurate parallel tracking for tile placement.

Tile Presence

In the event that TileBot should run out of tiles, an infrared sensor mounted on the dispenser will detect the absence of tiles and halt processing until the user re-fills the dispenser and presses the START button. Once the dispenser has been re-filled, TileBot will resume processing.

EXPERIMENTAL LAYOUT AND RESULTS

Dispenser Calibration

The dispenser was originally calibrated by repeatedly dispensing all of the tiles. The servo was run on less than optimal voltage to make dispensing problems more prevalent. When a defective tile was encountered (over sized, under sized, warped, etc.), causing a dispenser jam, it was eliminated from the stack. After several trials, most of the bad tiles were removed. At this time, the servo was restored to its original voltage. Although nothing can operate at 100%, since the calibration the dispenser rarely jams (knock on wood)! Positional repeatability of the tiles can be observed in Figure 12:

This image, like other TileBot images, was created by tracking from size to side, much like the motion of a dot-matrix printer head. Tiles are dispensed as required by the TileBot image processing program. The background is assumed to be white, while pixels in the image are represented by tiles. Though individual tiles tend to

Figure 12: Spot

rotate or slide slightly, the overall horizontal and vertical tracking of TileBot is much more accurate than that of a servo-powered vehicle. This is due to the use of stepper motors and a stepper motor controller designed and built by the author. The largest image created to date is an 80x42 image of the University of Florida Gator (Figure 13). This image measures 5ft x 2.5ft and uses 769 tiles.

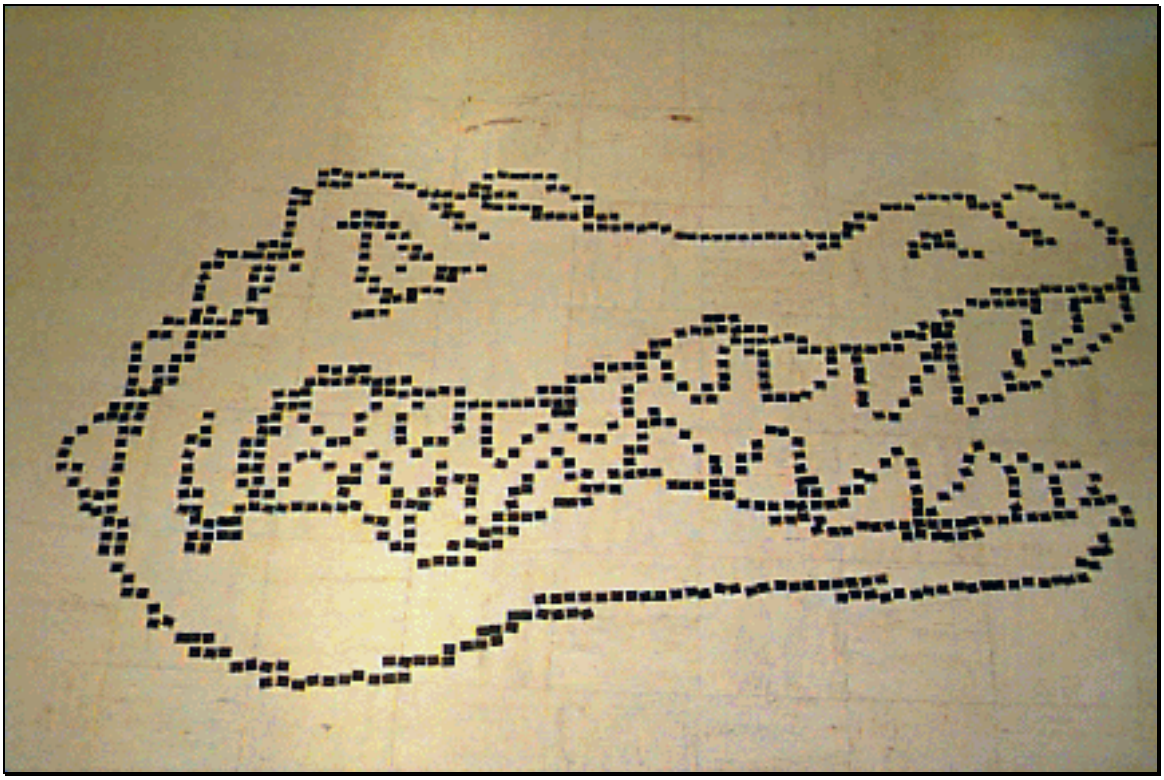

Figure 13: University of Florida Gator

CONCLUSION

Summary of Work

The following objectives were accomplished during the development of TileBot:

1) Autonomously dispense square tiles according to an image stored in memory.

2) Interaction with the environment through various sensors, actuators, and behaviors.

3) Power train and dispenser design providing smooth, repeatable dispensing and accurate tracking.

In spite of some of the shortcomings, TileBot has performed better than expected. It is the only robot of it's kind that the author has seen to date.

Technical Caveats

TileBot has changes slightly since it's original conception. I had hoped I could run with one pack of 8-AA cells. This was quickly changed to the current configuration when I had problems powering everything from the same pack. I did not fully realize the limitations of using stepper motors. They have proven to be more trouble to use than conventional DC motors, but they have provided excellent tracking ability, as demonstrated by the images TileBot has created. In this application, stepper motors are adequate. They would not be adequate in any application requiring high torque or low power consumption.

My first attempt at dispensing used a solenoid to push out the tiles. The solenoid had several drawbacks, including:

1. The solenoid did not have enough travel to complete the stroke.
2. The solenoid used up much power and got too hot to touch.
3. The solenoid would bind up.

The servo was used for these reasons.

Future Work

If I were to start over, I would do nothing differently. Stepper motors are more trouble to use than conventional DC motors, but I believe that the benefits will outweigh the inconveniences. As far as other future work on TileBot goes, the only thing left to do is to create a larger library of images to show off TileBot's capabilities.

REFERENCES

Jan96 Jantz, Scott, Sharp Sensor Modification, intelligent Machines Design Lab Notes, Department of Electrical Engineering, University of Florida, (1996).

Mar94 Martin, Fred, The 6.270 Robot Builder's Guide, The Media Lab, The Massachusetts Institute of Technology, (1994).

Mim86 Mims, Forrest M. III, Getting Started in Electronics, Radio Shack, (1986).

Nat89 National Semiconductor, Technical Data Sheet for the 74LS193 Synchronous 4-Bit Up/Down Binary Counter with Dual Clock, http://webdirect.natsemi.com/nscO/o (1989).

Sar95 Sargent, Murray III, and Shoemaker, Richard L., The Personal Computer from the Inside Out, Addison-Wesley Publishing Company, (1995).

APPENDIX

LOADED LIBRARIES

```
LIB_RW11.C
SERVO.ICB
SERVO.C
MUSIC.C
TILEBOT.C
DILBERT.C (Example)
```

TILEBOT.C

```
/*Programmer:           Alan Senior                              */
/*Class:                EEL 5666 - Mobile Robots                 */
/*Date:                 December 9, 1996                         */
/*Purpose:              This is the World Renowned TileBot Code! */

int TRUE=1,FALSE=0;
int FORWARD=0, REVERSE=1;
int waitTime=1;   /*Wait Time Before a Start (Seconds)*/

int H_Spacing=5;  /*Measured in Motor Steps*/
int V_Spacing=9;

/*Stepper Motor Globals*/
int byte7000=0xff;
int odometer=0;
int slowSpeed=40;
int fastSpeed=32;
int speed;
int direction;
int runStepper=FALSE;
int inStep=FALSE;

/*Dispenser Globals*/
int dispense=FALSE;
float servoStartAngle=100.0;
float servoHoldAngle=servoStartAngle+20.0;
float servoDispenseAngle=servoStartAngle+27.0;

/*Sensor Globals*/
int useSensors=TRUE;    /*FALSE to Ignore Sensors*/
int IR_Threshold=125;   /*For Prox Sensors*/
int tileThreshold=40;   /*For Tile Presence Sensor*/
int bumpThreshold=10;   /*For Bump Sensors*/
int IR[6];
int tilePresent;
int startButton;
int frontBump;
int rearBump;

void main()
{
        int i=0;
        int j=0;
        int k=0;
        int tripMeter;

        poke(0x7000,byte7000);
```

```
start_process(read_sensors());

while (!startButton)
{};
charge();
wait (waitTime*1000);

servo_on();
steppers_on();
direction=FORWARD;

initialize_dispenser();
start_process(dispense_tile());
start_process(step_motors());

/* Turn on IR Emitters */
if (useSensors==TRUE)
{
        byte7000=(byte7000 & 0x3f);
        poke(0x7000,byte7000);
}

speed=fastSpeed;
runStepper=TRUE;

while(j<V_Size)
{
        tripMeter=odometer;
        while(i<H_Size && i>=0)
        {

                /* Code to Respond to Sensors is Inserted here */
                if (useSensors==TRUE)
                {
                        if (direction==FORWARD) look_forward();
                        else look_back();

                        if (tilePresent==FALSE)
                        {
                                while (tilePresent==FALSE ||
                                       startButton==FALSE)
                                {
                                        runStepper=FALSE;
                                        play("1c2f1f2f1g2a1a2a");
                                }
                                runStepper=TRUE;
                        }
                }

                if (odometer>=(tripMeter+H_Spacing))
                {
                        tripMeter=odometer+H_Spacing;
                        if (bit(j*H_Size+i)==1)
                        {
                                speed=slowSpeed;
                                dispense=TRUE;
                        }
                        else
                        {
                                k=i+1-2*(direction==REVERSE);
                                if (bit(j*H_Size+k)==0 && k>0 &&
                                    k<(H_Size-1))
                                        speed=fastSpeed;
```

```
                                }
                                i=i+1-2*(direction==REVERSE);
                        }
                }
                i=i-1+2*(direction==REVERSE);
                scoot_down();
                j+=1;
        }

        servo_off();
        steppers_off();

        /* Turn off IR Emitters */
        byte7000=(byte7000 | 0x40);
        poke(0x7000,byte7000);

} /* End of Main */


/* This Code Initializes the Dispenser */

void initialize_dispenser()
{
        dispense=FALSE;
        servo_deg(servoDispenseAngle);
        wait(200);
        servo_deg(servoStartAngle);
        wait(200);
        servo_deg(servoHoldAngle);
        wait(90);
}


/* This Code Dispenses One Tile When dispense=TRUE, */
/* Then Sets dispense=FALSE.                        */

void dispense_tile()
{
        while (1)
        {
                if (dispense==TRUE)
                {
                        servo_deg(servoDispenseAngle);
                        wait(80);
                        servo_deg(servoStartAngle);
                        wait(160);
                        servo_deg(servoHoldAngle);
                        wait(90);
                        dispense=FALSE;
                }
        }
}


/* This Code Looks Forward */
void look_forward()
{
        int saveStepper=runStepper;
        while (IR[0]==TRUE || IR[1]==TRUE || IR[2]==TRUE)
        {
                runStepper=FALSE;
                play("1c2f1f2f1g2a1a2a");
        }
        while (frontBump==TRUE)
```

```
        {
                runStepper=FALSE;
                play("1c2f1f2f1g2a1a2a");
        }
        runStepper=saveStepper;
}


/* This Code Looks Back */
void look_back()
{
        int saveStepper=runStepper;
        while (IR[3]==TRUE || IR[4]==TRUE || IR[5]==TRUE)
        {
                runStepper=FALSE;
                play("1c2f1f2f1g2a1a2a");
        }
        while (rearBump==TRUE)
        {
                runStepper=FALSE;
                play("1c2f1f2f1g2a1a2a");
        }
        runStepper=saveStepper;
}


/* This Code Reads all Sensors */
void read_sensors()
{
        while(1)
        {

                IR[0]=(analog(2)>IR_Threshold)*useSensors;
                IR[1]=(analog(3)>IR_Threshold)*useSensors;
                IR[2]=(analog(4)>IR_Threshold)*useSensors;
                IR[3]=(analog(5)>IR_Threshold)*useSensors;
                IR[4]=(analog(6)>IR_Threshold)*useSensors;
                IR[5]=(analog(7)>IR_Threshold)*useSensors;
                poke(0x4000,0x2);
                tilePresent=(analog(0)>tileThreshold)*useSensors;
                poke(0x4000,0x3);
                startButton=(analog(0)>128);
                poke(0x4000,104);
                rearBump=(analog(1)>bumpThreshold)*useSensors;
                poke(0x4000,112);
                frontBump=(analog(1)>bumpThreshold)*useSensors;

                wait(100);
        }
}


/* This Code Controls the Stepper Motors */
void step_motors()
{
        while(1)
        {
                if (runStepper==TRUE)
                {
                        inStep=TRUE;
                        if (direction==0) byte7000=(byte7000 & 0xed);
                        else byte7000=(byte7000 | 0x12);
                        poke(0x7000,byte7000);
                        byte7000=(byte7000 & 0xf6);
```

```
                        poke(0x7000,byte7000);
                        byte7000=(byte7000 | 0x9);
                        poke(0x7000,byte7000);
                        inStep=FALSE;
                        odometer+=1;
                        wait(speed);
                }
        }
}


/* This code Scoots Down One Line */
void scoot_down()
{
        int i;
        int saveStepper=runStepper;
        int stepCount=30;
        int tripMeter;
        runStepper=FALSE;
        tripMeter=odometer;
        speed=slowSpeed;

        for (i=0;i<=stepCount;i++)
        {
                if (direction==FORWARD) leftStepForward();
                else leftStepReverse();
                look_forward();
                look_back();
        }

        runStepper=TRUE;
        while(odometer<=tripMeter+V_Spacing*4)
        {
                if (direction==FORWARD) look_forward();
                else look_back();
        }
        runStepper=FALSE;

        for (i=0;i<=stepCount;i++)
        {
                if (direction==FORWARD) leftStepReverse();
                else leftStepForward();
                look_forward();
                look_back();
        }

        if (direction==FORWARD) direction=REVERSE;
        else direction=FORWARD;
        tripMeter=odometer;

        speed=fastSpeed;
        runStepper=TRUE;
        while(odometer<=tripMeter+V_Spacing*4-10)
        {
                if (direction==FORWARD) look_forward();
                else look_back();
        }
        runStepper=FALSE;
        odometer=0;
        runStepper=saveStepper;
}

void leftStepForward()
{
```

```
        inStep=TRUE;
        byte7000=(byte7000 & 0xfd);
        poke(0x7000,byte7000);
        byte7000=(byte7000 & 0xfe);
        poke(0x7000,byte7000);
        byte7000=(byte7000 | 0x1);
        poke(0x7000,byte7000);
        inStep=FALSE;
        wait(speed);
}

void leftStepReverse()
{
        inStep=TRUE;
        byte7000=(byte7000 | 0x2);
        poke(0x7000,byte7000);
        byte7000=(byte7000 & 0xfe);
        poke(0x7000,byte7000);
        byte7000=(byte7000 | 0x1);
        poke(0x7000,byte7000);
        inStep=FALSE;
        wait(speed);
}

void rightStepForward()
{
        inStep=TRUE;
        byte7000=(byte7000 & 0xef);
        poke(0x7000,byte7000);
        byte7000=(byte7000 & 0xf7);
        poke(0x7000,byte7000);
        byte7000=(byte7000 | 0x8);
        poke(0x7000,byte7000);
        inStep=FALSE;
        wait(speed);
}

void rightStepReverse()
{
        inStep=TRUE;
        byte7000=(byte7000 | 0x10);
        poke(0x7000,byte7000);
        byte7000=(byte7000 & 0xf7);
        poke(0x7000,byte7000);
        byte7000=(byte7000 | 0x8);
        poke(0x7000,byte7000);
        inStep=FALSE;
        wait(speed);
}

void steppers_on()
{
        byte7000=(byte7000 & 0xdb);
        poke(0x7000,byte7000);
}

void steppers_off()
{
        byte7000=(byte7000 | 0x24);
        poke(0x7000,byte7000);
}

/* This is the code for the much-used function WAIT.    */
/* Mark Skowronski, February 28, 1995                   */
```

```
void wait(int milli_seconds)
{
        long timer_a;

        timer_a = mseconds() + (long) milli_seconds;
        while (timer_a > mseconds())
        {
                defer();
        }
}
```

DILBERT.C

```
/*Programmer:           Alan Senior                                */
/*Class:                EEL 5666 - Mobile Robots                   */
/*Date:                 December 9, 1996                           */
/*Purpose:              This Code Produces an Image of Dilbert     */
/*                      When Used In Conjunction With TILEBOT.C    */

int H_Size=25;
int V_Size=32;

int bit(int location)
{
return bits[location];
}

int bits[800]={0,0,0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,0,0,0,0,
               0,0,0,0,1,0,0,0,1,0,0,0,1,0,1,1,1,1,1,1,1,1,0,0,0,
               0,0,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,1,0,0,1,0,0,0,
               0,0,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,
               0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,
               0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,
               0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,
               0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,
               0,0,0,0,0,0,1,0,0,0,1,1,0,0,1,1,0,0,0,1,0,0,0,0,0,
               0,0,0,0,0,0,1,0,0,1,0,0,0,0,0,0,1,0,0,1,0,0,0,0,0,
               0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,1,0,1,0,0,0,0,0,
               0,0,0,0,0,0,1,0,0,0,1,1,0,0,1,1,0,0,0,1,0,0,0,0,0,
               0,0,0,1,1,1,1,1,0,0,1,0,0,1,1,0,0,1,0,0,1,0,0,0,0,
               0,0,1,0,0,0,0,0,1,0,0,0,1,0,0,0,0,1,0,0,1,0,0,0,0,
               0,0,1,0,0,0,0,0,1,0,0,0,1,1,0,0,1,0,0,0,0,1,0,0,0,
               0,0,1,0,0,1,0,0,0,1,0,1,0,0,1,1,0,0,0,0,0,1,0,0,0,
               0,0,1,1,1,1,1,0,0,0,0,1,0,0,0,0,0,0,0,0,1,1,1,0,0,
               0,0,0,0,1,1,0,0,0,0,1,0,0,0,0,0,1,0,0,0,0,1,0,0,0,
               0,0,0,0,0,1,0,0,0,0,1,0,0,0,0,1,0,0,0,0,1,0,0,0,0,
               0,0,0,0,0,1,0,0,0,1,0,0,0,0,1,0,0,0,0,1,0,0,0,0,0,
               0,0,0,0,0,1,0,0,0,0,1,1,1,1,0,0,0,0,0,1,0,0,0,0,0,
               0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,
               0,0,0,0,0,1,0,0,0,0,1,1,1,1,0,0,0,0,0,1,0,0,0,0,0,
               0,0,0,0,0,1,0,0,0,0,0,0,1,1,0,0,0,0,0,1,0,0,0,0,0,
               0,0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,
               0,0,1,1,0,1,1,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,
               0,0,1,0,0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,0,0,1,0,
               0,1,1,1,1,1,1,0,0,0,1,1,1,1,1,0,0,0,0,0,0,0,1,1,1,
               0,1,0,0,0,1,1,1,1,1,0,0,1,1,0,0,0,0,0,1,1,1,0,0,0,
               1,0,1,0,0,1,0,0,1,1,0,0,0,1,1,1,1,1,1,0,1,1,1,0,0};
```

ENCODE.M

```
clear;
[picture,map]=gifread('dilbert');

colormap(map);
colorbar;
count = 0;

imagesc(picture);
s=size(picture);

x = s(2);
y = s(1);

for i = 1:x,
   for j = 1:y,
     picture(j,i) = 1 - (picture(j,i) - 1);
        count = count + picture(j,i);
   end
end
count
figure(2)
imagesc(picture);
save numbers.txt picture -ascii
```

FORMAT.C

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

void main()
{
int H_Size=25;
int V_Size=32;

int j=0,k=0 ;
float i;
     FILE *fpr ;
     fpr=fopen("numbers.txt","r") ;
     FILE *fpw ;
     fpw=fopen("numbers2.txt","w") ;

while(k<V_Size)
{
     fscanf(fpr,"%f",&i) ;
     fprintf(fpw,"%d,",(int) i) ;
     printf("%d,",(int) i);
     j++;
     if (j==H_Size)
     {
       j=0;
       k++;
       fprintf(fpw,"\n") ;
       printf("\n");
     }

}
}
```