Project Vulcan

New Mexico Supercomputing Challenge

Final Report

April 1st, 2009

Team 48

Hope Christian School

Team Members:

Max VanBenthem

Matt Potok

Alex Jennings

Kent Delaney

Teacher:

Pamela Feather

Mentor:

Tom Laub


Firefighting
Project Vulcan

Acknowledgements

We would like to thank our mentors for all of the help they provided during the course of our project.

Mr. Tom Laub was extremely helpful in suggesting new ways to think about problems we faced. He also helped us by pointing out flaws in our logic or providing new logic for us to consider. Along with helping us think critically, Mr. Laub also dedicated much of his time ensuring that we progressed on schedule and meeting with us.

Finally, we would like to thank our teacher, Mrs. Pamela Feather. She was the one who suggested the Supercomputing Challenge. She allowed us to use some of our time in her class to complete our project.

Table of Contents

Introduction

We are team 48, also known as the 1337 Team. Our project, Project Vulcan, is to program a robot to be able to interact with its surroundings for one purpose: to locate and extinguish several candles. The robot will use a series of algorithms, or series of steps, to accomplish the tasks of finding and extinguishing these candles, as well as avoiding obstacles that may be in its path. We planned to use the C++ programming language to program this robot. Sensors located on specific areas of the robot allow the robot to sense candles, obstacles, and the boundary lines of its playing area. We set out with one goal: attempting not only to program this robot, but attempting to create a generic program that allows "thinking" on the part of any robot.

Problem Definition

The problem that we are faced with for this project is creating a program to allow the robot to quickly and efficiently find and extinguish candles in an obstacle-filled course. This program must be able to have the robot scan the environment with multiple sensors for obstacles and candles, move through an obstacle laden field to the candles while avoiding the obstacles, extinguish the candle with a fan, and repeat the process. This program will also contain problems that we must solve in order to complete the main problem. One of these minor problems is how we would access and use the sensors with the problem. Another problem would be developing a method of prioritizing obstacles before candles, or candles before obstacles, for the robot. Along with these problems we also have to connect to the robot through a USB cable or Bluetooth. Application of power along with judging how much velocity to give the motors is another problem which we must conquer.

Expected Results

Our expected results are comprised of four separate actions that, when applied together, will accomplish the problem our robot is faced with. These four actions are: scanning of the environment, taking the data from the scanner and plotting of a course to candles while avoiding obstacles, activation and alignment of fan to extinguish candles, and repeat of the actions previously stated. With the correct use of these actions the robot should be able to get to the candle and extinguish it with little or possibly no error.

The robot will scan the environment to start. This is because by doing this instead of just moving about first the robot will be able to know what its current surroundings are. From the data gathered from the scan it is able to plot a course to specified objects that are defined in the code. These objects can consist of the obstacles set in the course and candles that the robot is told to extinguish. This method of searching first and basing calculations on the data gathered will work better than guessing what the data is and assuming that a specific procedure will work no matter how the environment is set up. The logic behind this is that by basing calculations off of data collected the robot will be able to adapt to absolutely any environment it is subjected to. If all the data is assumed, there is an inherent risk of leaving the field or running into obstacles.

With the search-first-then-action strategy, the robot is able to adapt to situations at a greater extent because, like a person using their eyes, they see what is around them and try to avoid what obstacles are in their way based on what they saw. Just acting without scanning is like sending someone who is told to get to a specific point without running into anything, but has lost the ability to use any of their senses such as sight, touch, hearing, etc. Just think about it a little bit and you can come to the conclusion that the senseless person is going to collide with plenty of obstacles before they make it to the goal, while the person that can use their senses has a higher chance of avoiding obstacles and making it to the goal.

The sensors that are used by the robot consist of an optical sensor, three infrared sensors, and a sonic sensor. The sonic sensor is used to find where the obstacles are located and also used to give a general guess on the dimensions of the obstacles. The infrared sensors will be used to determine the location of the candles once the robot is close enough to pick up readings from them. The optical sensor will be used to keep the robot in the boundaries that are set up around the field in the form of lines.

Once the robot has finished successfully making its way to the candles while avoiding the obstacles throughout the course it will have to align its fan and activate it in order to extinguish the flame of the candle. When the candle is extinguished the robot will repeat the process of searching, navigating, extinguishing, and repeating indefinitely.

Approach

We decided that the best way to approach our problem lay in three simple steps: research our robot and its components, model different approaches to the logic and physical aspects of the program, and finally completing the program.

There were a number of factors involved which begged research before we could understand how to accomplish our goal and structure the program.

Understanding the robot itself is the key to programming it. Without an understanding of the robots components, programming is an arduous task. There are three types of sensors, one controller, a number of servos and motors and one central robot controller.

The three types of sensors – light, ultrasonic and infrared – are included to allow the robot to gather data from its surroundings. The playing field of the firefighting competition is a white field bounded by a black line. The light sensors are used to detect the black line and tell the robot that the edge of the field has been reached and the robot can no longer move forward. Infrared (IR) sensors detect the heat emitted by the candles, thereby telling the robot that a candle is present; when a reading from the IR sensor no longer returns a candle present, the candle is indicated as extinguished. Ultrasonic (US) sensors are used to find obstacles and extinguished candles. The reasoning behind this is that an extinguished candle will not emit any energy for the IR sensor to detect, but is still an obstacle and therefore needs to be avoided.

The particular light sensor installed on our robot is designed to allow the robot to detect a black line on a white background and vice versa. This sensor gathers visible light and interprets that light in black/white values. This sensor has five "eyes" that are able to detect white or black. Black essentially is the lack of light. The sensor collects data from the five parts, either a 1 or a 0, meaning light or dark, and sends that information to the robot controller. This allows the robot to "see" where it is moving on the course in relation to the border lines. If the robot detects a black line within its “eyesight,” the robot “knows” that it has reached the edge of the field.

Ultrasonic (US) sensors work by sending out a "ping" of ultrasonic sound, which is sound that is too high for human ears to hear. This sound travels until it reflects off an obstacle and bounces back toward the sensor. US sensors are designed to pick up an obstacle within a certain distance, meaning that after a certain distance, the reflected sound waves are too weak to be picked up by the sensor. In this case, the sensor registers no obstacle at that point. If an obstacle is within the sensor's area of "vision," the sensor interprets that there is something nearby. This information is then sent to the controller and can be utilized in the different calculations the robot will perform in order to navigate successfully through the obstacles.

The infrared (IR) sensors allow the robot to detect the heat from a candle, even if the candle is set closely behind an obstacle. IR sensors are specialized light sensors that pick up light with wavelengths longer than those of the visible spectrum. IR sensors work in a similar manner to the regular light sensors mentioned earlier. There is, however, only one "eye" present.

This robot uses the Serializer Wireless .NET controller by RoboticsConnection. This

contoller allows a .NET structure for C++ programs and allows a variety of connection possibilities, ranging from bluetooth to USB/Serial cable connections. This controller is made to be used with the different sensors on the robot. The controller is essentially a mini-computer processor, with pre-installed C++ libraries. It allows the robot to do calculations and activate certain classes at certain times, much like a regular computer processor.

Knowing the specifics about the robot allowed us to determine how to best structure a program for it. There were several ways to do this. One way was just to have a basic if-then statement telling the robot what to do. For example, if a human were reading the code, the code would read, "If there is an obstacle detected by this sensor, turn right 90 degrees, move forward 5 feet, turn left 90 degrees and continue," or "If there is a candle found by this sensor, move forward 9 inches then activate the fan." Of course, the distances would not be fixed as in these examples. Another idea would be to implement calculations to determine exact distances and actions. For example, using basic trigonometry would allow the determination of an exact degree amount to turn in order to clear the obstacle and a comfortable distance to move forward in order to move completely past the obstacle. Then an if-then statement would be used to move directly to any candle that may be in the immediate vicinity of the robot.

The idea for using trigonometry to navigate through the course was considered at first, but more research was done to ensure that this idea was as solid as it seemed. The use of trigonometry in having a robot navigate through the field is better than just saying “there’s an obstacle, turn 90 degrees, move forward” because it allows the robot to calculate an exact destination point and how to get there in an indirect way. This follows several steps: finding how many degrees to turn, calculating the distance needed to clear the obstacle, calculating how many degrees to turn (in the opposite direction of the first turn), and finally calculating how far to move forward from this position to arrive within an “extinguishing distance” of the candle. For example, an equation to find how many degrees to turn might be:

$$D = \sin^{-1}\left(\frac{X}{Z}\right) + d$$

Where “D” is the degrees to turn, “X” is the distance from the robot to the obstacle when the robot is at angle “Zero” (has not turned from its current course), “Z” is the diagonal distance from the robot to the edge of the obstacle, and “d” is a predetermined constant – in degrees – to allow the robot to clear the edge of the obstacle after turning. By taking the arcsine of X divided by Z, the angle from the robot facing degree “Zero” to the edge of the obstacle may be obtained. Adding the extra “d” degrees allows the robot to effectively avoid the obstacle with no problem.

Finding the distance to move in order to completely clear the obstacle is just another simple calculation:

$$C = \frac{X}{sin^{-1}\left(\frac{X}{Z}\right)} + c$$

Where “C” is the “clearance” distance being calculated and “c” is a pre-derived constant based

off of the dimensions of the robot and used to ensure a confident clearance of the obstacle.

It is not quite artificial intelligence (AI) because the robot has been told what to do with the data that it collects from its sensors, but it is similar to AI because the robot can run the calculations with ANY data that it collects, much like a human would say that 2 + 2 = 4 or 24 + 12 = 36. Any numbers can replace the two addends just like any data from the sensors can replace variables. In this way, the robot can "think" in a very basic and guided manner.

With this logic built, a language was needed. There were two possibilities that were immediately available to us: JAVA and C++. After researching the controller in the robot, C++ was chosen. Most of the research done in this aspect was done by reading through the pre-installed header files. These header files provide information needed to initialize and manipulate sensor and motor classes correctly and effectively. Along with the pre-installed header files, a few test files were also included to make sure that the controller was able to communicate with its various members.

Any basic coding document consists of three different segments: included files, variable declarations, and code to be executed using the declared variables. Included files generally are header files that define new information how certain information can be processed or manipulated.

A typical C++ program is structured in three main parts: preprocessor directives, variable declarations, and function declarations. This allows a well-structured program to be created.

Preprocessor directives are directions specifically for a computer processer to take before the program is compiled. These directions, denoted by a pound sign (#) may include defining constants for your program or including files that allow the manipulation of data. Defining constants requires the DEFINE command followed by a variable name and a value for that variable, for example:

#define PI 3.14

This command defines the constant PI as 3.14. A constant defined in this manner may not be changed during the program's execution, whereas any variable declared after the preprocessor directives may be altered.

Including files, as mentioned, allows the inclusion of code that makes data manipulation possible. The directives

#include <iostream>
#include <iomanip>

allow input and output between a user and a screen as well as manipulation of that input/output. Some included files contain math functions and other specialized functions, giving C++ a wide array of uses. Many included files end with the ".h" extension, marking them as a C header file, or a file that defines certain functions and variables outside of a program's scope.

The next portion of a C++ program is variable declarations. These require two parts: a variable type and a name. Variable types tell the processor what type of information is to be stored under that variables name. The simplest types of variables are integer (int), character (char), and string. A value can be assigned to variables when declaring them as well, but this is not required:

int Num = 35;

This command creates a variable named Num and sets its value to 35. Variables may also be defined as constants by placing the *const* keyword in front of the variable type.

The final segment of a C++ program is the declaration of functions to be called during the course of the program's execution. Functions also require a type in front of the function name. This type, however, determines what type of information is returned by the function. Declaring a function takes place as follows:

```
Int functionName()
{
   Code to be executed
}
```

The built-in C++ libraries contained several test programs to ensure that the robot controller was indeed working. The three main parts of one of these files is outlined below:

Including Files:

```
#include "stdafx.h"
#include "PidMotor.h"
```

Defining variables:

```
const FLOAT64 CPidMotor::DEFAULT_VELOCITY_DIVIDER_DEFAULT =
ActuatorConstants::DEFAULT_VELOCITY_DIVIDER_DEFAULT;

const FLOAT64 CPidMotor::DEFAULT_GEAR_REDUCTION =
ActuatorConstants::DEFAULT_GEAR_REDUCTION;

const FLOAT64 CPidMotor::DEFAULT_WHEEL_DIAMETER =
ActuatorConstants::DEFAULT_WHEEL_DIAMETER;

const FLOAT64 CPidMotor::DEFAULT_WHEEL_CIRCUMFERENCE =
ActuatorConstants::DEFAULT_WHEEL_CIRCUMFERENCE;

const FLOAT64 CPidMotor::DEFAULT_TRACK_CIRCUMFERENCE =
ActuatorConstants::DEFAULT_TRACK_CIRCUMFERENCE;
```

Function Definitions:

```
string CPidMotor::GenerateCommand(COMMAND_TYPE eCommandType)
```

{
string strCommand;

Code to be executed

return (strCommand);
}

PidMotor.cpp code snippets, test file contained in the Serializer libraries in the robot controller

This example defines what files to include, defines some variables, and defines a function that returns string data.

Our first step, as mentioned, was research. Because almost our entire objective is centered around the robot that we programmed, much of our research was about the robot: its component sensors, motors, controller, and how they all worked.

The second step we took was to model different ways to have the robot navigate through the course and extinguish candles. We also had to determine how to prioritize searching for candles and obstacles. For example, let's say we have finding candles as a higher priority. The robot sees the candle, ignores any obstacle that may or may not be in between the robot and the candle, and executes code to move the robot straight to the candle. The robot will bump into the obstacle and have to start the course over. Or, say the opposite. The robot searches for obstacles first and executes the proper code for avoiding obstacle. The problem with this is that the robot will see an obstacle almost anywhere on the field; therefore, it will move according to the obstacles and ignore the candles entirely. We created two main models – using the NetLogo modeling software discussed in Appendix I – that simulated both of these strategies; these models were modified several times, giving us working models for both strategies. The code for these two models is also included at the end of Appendix I.

Models A and B were similar in many aspects, such as navigational methods; however, they changed the priorities. Model A focused on obstacles first, then candles, whereas Model B searched for candles first, then obstacles. Model B was more successful and therefore will be discussed in greater detail. This model searched for candles with code to avoid obstacles while moving forward and while moving toward the candle constructed inside each function. Model B, however, had problems facing oddly shaped obstacles. For example, if an "L" shaped obstacle is placed directly next to the field's boundary, the robot would get caught in the cul-de-sac bordered by the obstacle and boundary line and fail to escape. This remained the only major problem that was unsolved when we finished with our modeling.

Both models were successful during their simulation. However, we decided to use the logic from Model B to structure our actual program. Therefore the logic would be:

- Search in a 360 degree arc for any candles near the robot.

- If a candle is found, head toward it, avoiding any obstacles, and activate its fan to extinguish the candle.
- If a candle is not found, head forward and slightly to one side - avoiding any obstacles - and search for a candle again.

Avoiding an obstacle would be very simple: If an obstacle is found with the ultrasonic sensor, turn X degrees to the right/left.

Our approach to the programming was simple as well: move the robot, then work on its sensors. The logic behind this was that if the robot cannot move, the sensors and its overall goal are pointless. However, connecting to the robot made the programming difficult. The Bluetooth and a USB/Serial cable failed to allow a connection to the robot. During this time, we worked on our previously discussed models and attempted to fix the problem. A discussion from a leader in the New Mexico RoboRAVE yielded no fruit. We managed to find a piece of software from a similar robot that finally allowed us to communicate with the robot.

This approach of research, modeling, and then taking action was excellent for our project. Despite the difficultly connecting to the robot, we were able to determine that a program can be made to allow a robot to "think" for itself and adapt to different situations.

Conclusion

Though this project had a few unpredicted and suboptimal events, it can be rated as an overall success. As connecting to the robot proved much more difficult than originally planned, we were never able to write and test a successful program for the actual robot. However, through NetLogo, we were able to simulate the firefighting robot challenge. We were able to write two distinct programs that successfully completed the challenge. Both programs had advantages and disadvantages. Had we been able to successfully connect to the robot, we would have used a hybrid of these programs to maximize robot success. Our material goal of producing a robot programmed to extinguish candles and navigate an obstacle-laden field was not reached. However, in theory, we succeeded in producing a program that would allow the robot to do as we had planned. In this sense, our project was a success.

References

Beginning Programming for Dummies aided in our figuring out how classes of the SerializerCpp library were declared.

Brain, Marshall. "How USB Ports Work." 1998. How Stuff Works. A Discovery Company. HowStuffWorks, Inc. 17 December 2008 <http://computer.howstuffworks.com/usb3.htm>.

Loyd, Ryan, Matthew Paiz, and Mark Wunsch. Dynamic Software Evolution: An Evolutionary Approach to Artificail Intelligence. New Mexico Supercomputing Challenge Final Report. 5 April 2006.

"Programming Your Robot to Navigate." RidgeSoft.com, v. 1.2. 2005 RidgeSoft LLC.

Wang, Wallace. "Beginning Programming: All-in-one Desk Reference For Dummies." Hoboken, NJ: Wiley Publishing, Inc. 2008.

Wilensky, U. (1999). NetLogo. http://ccl.northwestern.edu/netlogo/. Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL.

Appendix I: NetLogo

We used NetLogo to simulate the robot and its function of finding and extinguishing candles. This program provided an agent based simulation language that was relatively simple to learn and capable of successfully imitating the robot. By evaluating different approaches to solving the problem, we were able to determine the best program for the actual robot.

NetLogo uses a programming language particular to itself, but similar to other languages such as C++ and Java. However, the detailed syntax differs from these more commonly known programming languages. One difficulty our group faced was learning this new language in order to successfully simulate the robot program. We accomplished this task mainly by writing simple programs to complete minor tasks. We then slowly increased the magnitude of our programs to encompass larger and larger tasks.

The evolution of our programming began with simply setting up the playing field. All the patches in the model area were set the color blue. This is a simple line of code, requiring only two lines of code:

```
ask patches [
set pcolor blue ]
```

Figure 1

Then the patches on the borders were set black. This represented the playing field that the actual robot would have to navigate through. Setting the edges black proved slightly more difficult:

```
ask patches with [ max-pxcor = pxcor ] [ set pcolor black ]
ask patches with [ min-pxcor = pxcor ] [ set pcolor black ]
ask patches with [ max-pycor = pycor ] [ set pcolor black ]
ask patches with [ min-pycor = pycor ] [ set pcolor black ]
```

Figure 2

We then created a robot, called a turtle in NetLogo. To create a robot in NetLogo, you simply input the code: create-robots (number of robots). This appeared as a small arrow on the model area. We then wrote a program that allowed the robot to move around the field and turn around when it reached the border. In order to accomplish this, we wrote a short line of code telling the robot to move back one space and turn 130 degrees to the right if it found itself on a black patch.

```
ask robot 0 [
   ; robot 0 asks the patch it is on (the patch that is 0 patches ahead of it)
   ask patch-ahead 0 [
    ; if the patches color is black then robot 0 backs up one patch and turns right 130
    ; degrees, thus turning back into the field and remaining in bounds
    if pcolor = black [
     ask robot 0 [ bk 1 rt 130 ]
     ]
    ]
   ]
   End
```

Figure 3

Next, we turned several blocks different colors to simulate obstacles and candles. The red blocks represented obstacles, and the yellow blocks represented candles. We used a similar algorithm to the borderline avoidance to allow the robot to navigate around the obstacles. This

time, however, if the robot saw a red square five or less squares away in a radius of sixty degrees, it would turn right ninety degrees, move forward one patch, and turn left ninety degrees. This algorithm allowed the simulated robot to successfully move around an obstacle in its path. In order to accomplish this task, we had to use two loops: one to loop the distance search and another to loop the angle search. The code to loop the distance search is a while statement: while [distToObstacle <= 5]. The angle incrementation is contained within this distance incrementation:

```
while [ distToObstacle <= 5 ]
        [ set LEFT_TO_RIGHT_SEARCHER (-30)
        while [ LEFT_TO_RIGHT_SEARCHER <= 30 ] [ … ] ... ].
```

Figure 4

The final step in our programming development required the robot to locate and extinguish the candle. Extinguishing the candle was fairly simple. All that was required to accomplish this was to tell the robot to set the patch color to a different color than yellow. In the simulation, if a yellow space is one patch in front of the robot, it is turned grey or green. Locating the candle proved to be the hardest piece of the program. Our team developed two separate programs to accomplish this task.

The first program, Model A, approached this task in the same manner as the obstacle avoidance system. This time, however, the robot would move towards the object it saw. It used the same double-loop system as the obstacle avoidance code. First, it looped the distance while the distance was less than or equal to six. Within that loop, the robot searched from negative thirty degrees to positive thirty degrees. This code was nearly identical to the code located in figure 4. If the robot saw a yellow square six or less patches away in a radius of sixty degrees, it would turn the correct angle towards the candle and move one less patch forward than the total distance to the candle. Turning the correct angle proved to be a slight problem. We made the process a little more complicated than it needed to be. Instead of turning the robot left the absolute value of the angle that returned a yellow patch, or right the value of the angle that returned a yellow patch, we originally programmed the robot to turn left sixty degrees minus the angle. We corrected this mistake by replacing the incorrect code with:

```
ask robot 0 [
        lt abs x
        fd ( distToCandle - 1 )
```

Figure 5

and

```
ask robot 0 [
        rt x
        fd ( distToCandle - 1 )
```

Figure 6

Moving forward one patch less than the total distance to the candle places the candle one patch in front of the robot, allowing it to be extinguished. This program found candles successfully, but it did so in a seemingly random search. The robot had limited line of sight, allowing it to only see in a particular window. If there were no candles, it continued forward, even if there was nothing in that area.

The second program, Model B, took on an innovative technique. This program told the robot to spin in a circle every few times it moved forward. While spinning it searched for a candle six or less patches away. If it saw a yellow patch, it would move forward one less patch than the total distance to the yellow patch. This program also successfully located candles. It

seemed somewhat slower than Model A, but it was a much more organized system. The spin took a lot of time, but, overall, Model B was more successful at finding the candle than Model A.

These two models gave us a better understanding of how the actual robot would act. It prepared us to write the actual program, and gave us an understanding of what we needed to do. Without the NetLogo modeling, it would have taken much longer to develop a successful program for the firefighting robot.

Model A Netlogo Simulation Code

```
GLOBALS
[
LOOP_CANDLE_SETTER
LEFT_TO_RIGHT_SEARCHER
distToObstacle
x
distToCandle
]

; robot breed declaration
breed [ robots robot ]

to setup
  ; clears the screen of all present items for the setup of the firefighting field
  clear-all

  ; sets all patch colors to blue, because black will be used for the boundaries of
  ; the firefighting field
  ask patches [
    set pcolor blue
  ]

  ; sets all patches with their x coordinate equal to the maximum and minimum x coordinates
  ; of the field, likewise for y coordinate values.  This creates a black "box" around the
  ; blue field

; queries all patches                  sets the color black
;         |       with x equal to max-x         |
  ask patches with [ max-pxcor = pxcor ] [ set pcolor black ]
  ask patches with [ min-pxcor = pxcor ] [ set pcolor black ]
  ask patches with [ max-pycor = pycor ] [ set pcolor black ]
  ask patches with [ min-pycor = pycor ] [ set pcolor black ]

  ; creates four obstacles
  repeat 4 [
    ; queries all patches  selects random x and y coordinates (the minus 8 is to center the
    ; obstacles in the field.       |
    ;                 |          |       sets the color to red
      ask patch (random 20 - 6) (random 20 - 6) [ set pcolor red ask neighbors [ set pcolor red ]]
  ]

  ; places a candle in the midts of an obstacle.  Candles are colored yellow
```

```
  ; NOTE: more candles will be added later, after a successful first extinguishing

  repeat 3 [
  ; LOOP_CANDLE_SETTER is set to "no" to allow the while loop to run and place one candle
  set LOOP_CANDLE_SETTER "no"
  while [ LOOP_CANDLE_SETTER = "no" ] [
    ask patch (random 20 - 6) (random 20 - 6) [
      ; checks the patch color, if the patch is red, the code tries again
      if pcolor != red [
        set pcolor yellow
        set LOOP_CANDLE_SETTER "yes"
      ]
    ]
  ]
]
  ; creates a turtle of the robot breed
  create-robots 1 [
    ; sets the y coordinate to the minimum y coordinate plus 1, placing the robot one patch
    ; above the lowest boundary, and the x coordinate at the maximum x coordinate minus 1,
    ; making the robots position the very lower right corner of the valid plaing field
    set ycor (min-pycor + 1) set xcor (max-pxcor - 1)

    ; sets the robots color to orange, for visibility purposes
    set color orange
  ]
end

; code body to allow the robot to detect the boundaries of the field
to checkLine
  ; queries robot 0
  ask robot 0 [
    ; robot 0 asks the patch it is on (the patch that is 0 patches ahead of it)
    ask patch-ahead 0 [
      ; if the patches color is black then robot 0 backs up one patch and turns right 130
      ; degrees, thus turning back into the field and remaining in bounds
      if pcolor = black [
        ask robot 0 [ bk 1 rt 130 ]
       ]
      ]
    end

; if the patch color is green (exstinguished candle) it will back up 1 and turn left
```

```
; 30 degrees
to avoidCandle

  ask robot 0 [
    ask patch-ahead 1 [

        if pcolor = green [
        ask robot 0 [ bk 1 lt 80 ]
        ]
      ]
    ]
end

; code body to allow the finding and avoiding of obstacles (red patches)
to avoidObstacle
  set distToObstacle ( 0 )
  while [ distToObstacle <= 5 ] [
  set LEFT_TO_RIGHT_SEARCHER (-30)
  ; loops while LEFT_TO_RIGHT_SEARCHER is less than or equal to 1
  while [ LEFT_TO_RIGHT_SEARCHER <= 30 ] [
    ; queries robot 0
    ask robot 0 [
      ; robot 0 queries the patch that is right LEFT_TO_RIGHT_SEARCHER patches and 5
patches
      ; ahead
      ifelse LEFT_TO_RIGHT_SEARCHER < 0
      [ ask patch-left-and-ahead abs LEFT_TO_RIGHT_SEARCHER distToObstacle [
        ; if the patch color is red, then robot 0 turns right 90 degrees, moves forward
        ; one patch, then turns left 90 degrees, effectively avoiding the obstacle
          if pcolor = red [
            ask robot 0 [
              rt 90
              fd 1
              lt 90
            ]
          ]
        ]
      ]
      [ ask patch-right-and-ahead LEFT_TO_RIGHT_SEARCHER distToObstacle [
        ; if the patch color is red, then robot 0 turns right 90 degrees, moves forward
        ; one patch, then turns left 90 degrees, effectively avoiding the obstacle
          if pcolor = red [
            ask robot 0 [
```

```
            rt 90
            fd 1
            lt 90
          ]
        ]
      ]
    ]
  ]
  ; increments LEFT_TO_RIGHT_SEARCHER in order to limit the while loop
  set LEFT_TO_RIGHT_SEARCHER ( LEFT_TO_RIGHT_SEARCHER + 1)
  ]
  set distToObstacle ( distToObstacle + 1 )
  ]
end

; code that allows robot to find the yellow block
to findCandle
   set distToCandle ( 0 )
   while [ distToCandle <= 6 ] [
   set x (-30)
   while [ x <= 30 ] [
   ; queries robot 0
    ask robot 0 [
     ; robot 0 queries the patch that is right LEFT_TO_RIGHT_SEARCHER patches and 5
patches
      ; ahead
      ifelse x < 0
      [ ask patch-left-and-ahead abs x distToCandle [
        ; if the patch color is yellow, move towards patch
          if pcolor = yellow [
            ask robot 0 [
              lt abs x
              fd ( distToCandle - 1 )

            ]
          ]
        ]
      ]
      [ ask patch-right-and-ahead x distToCandle [
        ; if the patch color is yellow, move towards patch
          if pcolor = yellow [
            ask robot 0 [
              rt x
```

```
            fd ( distToCandle - 1 )
          ]
        ]
      ]
    ]
  ]
  ; increments LEFT_TO_RIGHT_SEARCHER in order to limit the while loop
  set x ( x + 1)
  ]
  set distToCandle ( distToCandle + 1 )

  ]
end

; code that allows robot to simulate extinguishing the candle
to exstinguishCandle
  ask robot 0 [
  ; robot asks patch ahead of it if it is yellow and extinguishes it (turns it blue)
  ; if it is
   ask patch-ahead 1
    [ if pcolor = yellow [
        set pcolor green
    ]
 ]
 ]
end



; code that initiates all motions and actions of the robot
to move
  ; asks the first robot (value of n - 1, where n is the total number of robots in the
  ; field. in this case n = 1) to move forward one patch
  ask robot 0 [
    fd 1
    checkLine
    avoidObstacle
    findCandle
    exstinguishCandle
    avoidCandle
  ]
end
```

Model B Netlogo Simulation Code

```
GLOBALS
[

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; All variables have a numerical value, including true/false. ;
; True/false variables will be set at 1/0 respectively.       ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

candleSetYesNo        ; true/false if a candle has been successfully set
candleDistance        ; distance searched from the turtle for a candle (in patches)
MAXcandleDistance     ; maximum distance searched for a candle
candleYesNo           ; true/false if a candle has been found by the turtle
degreeCounter         ; number of degrees turned from the starting direction.  1 = 0 turned degrees
obstacleDistance      ; distance searched from the turtle for an obstacle (in patches)
MAXobstacleDistance   ; maximum distance searched for an obstacle
obstacleYesNo         ; true/false if an obstacle has been found by the turtle
]

; create the robot breed
breed [robots robot]

to setup
  ; clear the whole screen if pre-existing items are present
  clear-all

  ; creates the active robot at the lower right corner, which in this field is at
  ; (15, -15) in patches
  create-robots 1 [ set xcor 15 set ycor -15 set color orange ]

  ; This long line of code asks patches that do not have their x coordinates
  ; or y coordinates equal to the maximum x coordinates or y coordinates (respectively)
  ; to set their color blue.  This creates the playing field with boundaries in one
  ; easy step
  ask patches with [ pxcor != max-pxcor and pxcor != min-pxcor and pycor != max-pycor and
pycor != min-pycor ] [ set pcolor blue ]

  ; this creates four obstacles
  repeat 4 [
    ; selects a random patch, within a defined area, and sets its color to red
    ; then its neighbors are turned red in order to create a larger obstacle
    ask patch (random 16 - 8) (random 16 - 8) [set pcolor red ask neighbors [ set pcolor red ] ]
  ]
```

```
  ; this creates four candles
  repeat 4 [
    ; candleYesNo is set to false ( 0 )
    set candleSetYesNo 0
    ; loops until candleYesNo is set to true by placing a candle in a random patch
    while [ candleSetYesNo = 0 ] [
      ask patch (random 16 - 8) (random 16 - 8) [
        if pcolor != red [
          if pcolor != yellow [
            set candleSetYesNo 1
            set pcolor yellow
          ]
        ]
      ]
    ]
  ]
end

to checkObstacle
  ; sets the maximum search distance to 5 patches
  set MAXobstacleDistance 5
  ; sets obstacleYesNo to false
  set obstacleYesNo 0
  ; degrees to the right or left, negative = left, positive = right
  set degreeCounter (-30)

  ; loop until all the possible directions have been searched or an obstacle has been found
  while [ degreeCounter <= 30 and obstacleYesNo = 0 ] [
    ; sets the distance to look away from turtle
    set obstacleDistance 1
    ; loops until obstacleDistance reaches the maximum distance or an obstacle is found
    while [ obstacleDistance <= MAXobstacleDistance and obstacleYesNo = 0 ] [
      ; if degreeCounter is negative, it searches left
      ask robot 0 [
        if degreeCounter < 0 [
          ask patch-left-and-ahead (abs degreeCounter) obstacleDistance [
            if pcolor = red or pcolor = grey [
              set obstacleYesNo 1
              ask robot 0 [ rt 30 ]
            ]
          ]
        ]
```

```
      ; if degreeCounter = 0, then the turtle searches stright ahead
      if degreeCounter = 0 [
        ask patch-ahead obstacleDistance [
          if pcolor = red [
            set obstacleYesNo 1
            ask robot 0 [ rt 90 ]
          ]
        ]
      ]

      ; if degreeCounter is positive, then the turtle searches right
      if degreeCounter > 0 [
        ask patch-right-and-ahead degreeCounter obstacleDistance [
          if pcolor = red [
            set obstacleYesNo 1
            ask robot 0 [ lt 30 ]
          ]
        ]
      ]
    ]
    ; increments the distance to search
    set obstacleDistance (obstacleDistance + 1)
  ]
  ; increments the turn amount
  set degreeCounter (degreeCounter + 1)
]
end

to checkForObstacle
  ; if a robot detects an obstacle in the way, it turns left to avoid it
  ask robot 0 [
    ask patch-ahead 1 [
      if pcolor = red [
        ask robot 0 [
          lt 90
        ]
      ]
    ]
  ]
end

to headToCandle
```

```
  ; if a candle has been found, then the robot moves forward that distance - 1,
  ; checking for obstacles after each advance
  repeat (candleDistance - 1) [
    ask robot 0 [
      fd 1
      checkForObstacle
      checkLine
    ]
  ]

  ; if the robot finds a candle, it is extinguished
  ask robot 0 [
    ask patch-ahead 1 [
      if pcolor = yellow [
        set pcolor grey
      ]
    ]
  ]
end

to checkCandle
  ; the maximum distance from the turtle to look for a candle
  set MAXcandleDistance 10
  ; candleYesNo is set to false
  set candleYesNo 0
  ; degreeCounter is "zeroed" out, 1 = 0
  set degreeCounter 1

  ; loops until degreeCounter records a complete turn or a candle is found
  while [ degreeCounter <= 360 and candleYesNo = 0 ]  [
    ; sets the distance from the turtle to look for a candle
    set candleDistance 1
    ; loops while that distance is less than the maximum distance or a candle is found
    while [ candleDistance <= MAXcandleDistance and candleYesNo = 0 ] [
      ask robot 0 [
        ask patch-ahead candleDistance [
          if pcolor = yellow [
            set candleYesNo 1
            headToCandle
          ]
        ]
      ]
      ; increments candleDistance
```

```
      set candleDistance (candleDistance + 1)
    ]
    ; asks the robot to turn rith 1 degree then records the turn by incrementing degreeCounter
    ask robot 0 [ rt 1 ]
    set degreeCounter (degreeCounter + 1)
  ]
end

to checkLine
  ; if the robot sees a black patch under it, then it backs up and turns back into the field
  ask robot 0 [
    ask patch-ahead 0 [
      if pcolor = black [
        ask robot 0 [
          bk 1
          rt 135
        ]
      ]
    ]
  ]
end

to move
  ask robot 0 [
    ; asks the robot to move forward 10 patches and turn right 10 degrees
    repeat 10 [
      fd 1
      ; after moving forward, the robot checks for the boundary
      checkLine
      ; the robot checks for obstacles before moving forward
      checkObstacle
      ; robot doesn't travel in a straight line
      rt 2
    ]
    ; after the completion of movement, the robot checks for a candle
    ; executing sub-bodies of code as defined by the conditions
    ; set in the "function"
    checkCandle
  ]
end
```