Part II
Developing A Toolbox Of Behaviors

In Part II we develop a toolbox of utility programs. The programs impart the robot with a collection of behaviors that enable it to handle specific tasks. Each chapter focuses on a single behavior, evolving algorithms that can work in a variety of situations of increasing complexity. In Part III we utilize combinations of these behaviors to create solutions to real-world problems.

We build on the programming skills developed in Part I by utilizing new commands and functions from the language as well as show how to use arrays to manipulate data more efficiently. Additional robot commands and functions are introduced along with more sophisticated interrogation and manipulation of the standard sensors on the robot. We also utilize customizable sensors to handle more demanding situations and show how to use advanced features of the standard sensors.

Upon completing Part II you will be able to:

- Create complicated programs and employ advanced programming techniques.
- Utilize all the sensors on the robot and analyze their data more intricately.
- Utilize arrays and array commands and functions along with looping constructs to manipulate large amounts of data.
- Improve on the behaviors introduced in this part as well as create new ones of your own.
- Appreciate the advantages of using RobotBASIC as a research and development tool so as to minimize abortive efforts in a real-world project.

Chapter 7
Following A Line

In Chapter 5 we made the robot move around the screen freely while avoiding objects in the environment. A robot is a device that can be made to do useful work. To be able to achieve its assigned tasks the robot usually will need to move to specific locations where it will perform the required work. There are various ways we can move the robot around:

- Move along a prescribed path defined by a line
- Freely move along a path that the robot determines for itself
- Move to a specific destination while keeping within a specified limited boundary

In subsequent chapters we will explore the second and third options. This chapter will explore the first option. The advantage of having the robot move along a designated path is that we can ensure where the robot will be all the time as it progresses from one location to another. It is also easy to make sure that the robot will have no obstacles along its path or at least avoid having to program it with a sophisticated obstacle avoidance behavior.

An example application for a robot of this kind is an automated waiter that carries food items along a continuous loop starting at the kitchen winding around and between the tables, and returning to the kitchen. It would not be desirable to have a track that protrudes above the ground due to the risk of customers tripping over the exposed tracks. A robot that can follow a line painted on the ground would be preferable. The line does not have to be visible to humans. Only the robot's sensors need to see it.

Developing a robot that can follow a line on the floor (perhaps black tape on a white floor) is a common activity at many robotics clubs. The project is straightforward enough that it usually can be understood and accomplished by novice robot enthusiasts, yet it is complex enough to introduce them to many aspects of robotics.

7.1 The Base Program

In this chapter we will develop a few algorithms to perform line following, but before we can do this we need to develop a base program in which we will place the code that implements the various algorithms. The base program sets up the robot and the environment and then starts the robot on its way to follow the line using the algorithm that we want to test.

The first thing we need is to draw a line on the screen for the robot to follow. Next we need to create and place the robot on the screen. Finally we want the robot to start executing the Line-Following algorithm we are trying to test.

The code in Figure 7.1 contains three subroutines called *InitializeRobot, DrawLine* and *FollowLine*. All the main program does is call each of these in turn. The third line after the *MainProgram* label makes the robot move forward 10 pixels. The purpose of this will be discussed below.

Notice how the use of subroutines makes it easier to understand what the program accomplishes. The subroutine names indicate what the subroutines do. The main program becomes an over all manager. Of course the actual details of each subroutine's actions may need further explanation but if you keep this policy of *modularization* throughout your programs, whenever possible, the programs become self-documenting.

The subroutine *DrawLine* creates a line for the robot to follow. The first statement sets the width of the lines (in pixels). The next sets their color and the one that follows positions the cursor on the screen. A series of LineTo commands draw the line one segment at a time. Refer to Appendix C.7 for details on these commands. Also see section 7.4 below for a better way to implement this routine.

The *IntitializeRobot* subroutine positions the robot. The command rLocate *x,y,heading* creates the robot and places it on the screen at the specified location and heading. Since the robot's default radius is 20 pixels and this routine places the center of the robot 30 pixels to the right of the start of the line, the front edge of the robot will be 10 pixels away from the line. This is why we need to forward the robot 10 pixels before we start the line following routine. This action brings the front of the robot to the beginning of the line in preparation for following it. In a later improvement (section 7.5) this action will not be necessary.

RobotBASIC normally issues an error if the robot bumps into a color on the screen (collision with some obstacle). Since the robot must be able to move over the line, we must tell the system the color of the line so that it can differentiate it from an obstacle. We do this with the rInvisible *Green* command. *Green* is used here because the line color was set to green in the *DrawLine* subroutine. Refer to Appendix C.9 for a detailed discussion on the rInvisible command

The final action of the *MainProgram* is to call the *FollowLine* subroutine. This subroutine is the code that actually performs the task of following the line. All the routines we will develop in this chapter will be replacements for this subroutine. Figure 7.1B shows the output screen when the program in Figure 7.1 is run. For now *FollowLine* is left empty so no line following will occur.

```
MainProgram:
  gosub DrawLine
  goSub InitializeRobot
  rForward 10   // move the robot over to the line
  goSub FollowLine
End
//=======================================================
InitializeRobot:
  //-- place the robot at the beginning of the line
  //-- and face it left 90 degrees
  rLocate  200, 71, -90
  rInvisible Green   //-- Green is a line not an object
Return
//=======================================================
DrawLine:
  linewidth 4
  setcolor Green
  gotoxy 170,71
  lineto 160,72
  lineto 145,80
  lineto 140,90
  lineto 130,100
  lineto 125,110
  lineto 120,140
  lineto 130,200
  lineto 140,250
  lineto 130,270
  lineto 145,300
  lineto 200,350
```

```
  lineto 300,325
  lineto 450,375
  lineto 450,450
  lineto 600,450
  lineto 600,400
  lineto 650,200
  lineto 500,350
return
//=========================================================
FollowLine:
  //-----Line Following algorithm
  //--we will put code here to make the robot follow a line
Return
```

Figure 7.1: This code draws a line on the screen and places the robot at its start

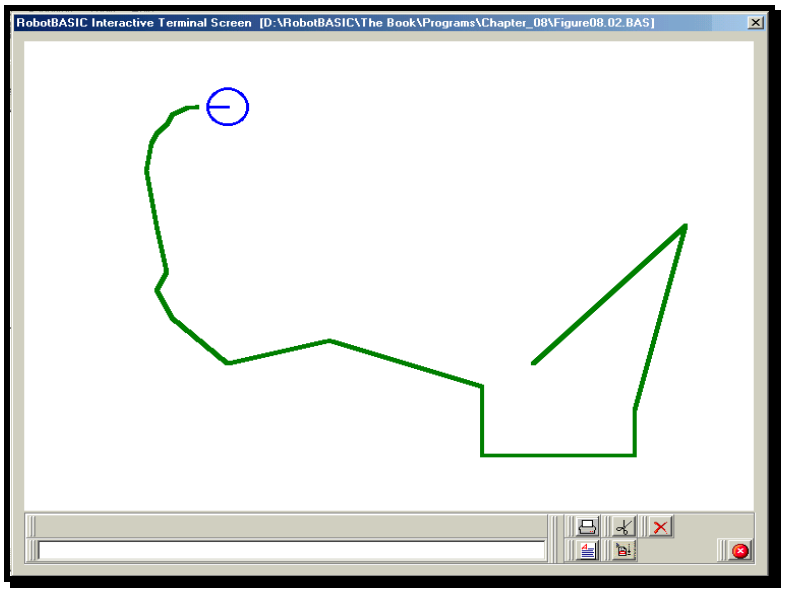

Figure 7.1B: The robot is ready to approach the line

7.2 An Initial Algorithm

RobotBASIC's robot has three line sensors. One is mounted directly in front of the robot, and the other two are spaced 10 degrees left and right of the front sensor. Figure

7.2 shows this setup. The scale has been enlarged to make the setup obvious. Refer to this diagram to visualize the action of the algorithms that will be developed.

Figure 7.2: The Line-Sensors Setup In RobotBASIC

With three sensors there are many choices for how to develop a line-following robot. You could, for example use only the front sensor and have the robot constantly swing left and right as it attempts to keep the sensor on the line. Such an algorithm can work, but the robot follows the line with an oscillating snaking sort of motion that is far from efficient.

On deeper analysis, you might decide that a better implementation would be to use the two outside sensors. In this case, the robot should try to keep the line between the two outside sensors. It can do this by turning a little to the right every time the sensor on the right detects the line and turning left when the left sensor is triggered.

- **Reading The Line Sensors**
  All three line sensors are accessed simultaneously with the single function rSense(*Color*). If you do not specify a color by using rSense(), the first color in the list of invisible colors passed to the command rInvisible will be considered as the line color to be sensed. You *must* specify a list of invisible colors before you do any line sensing, and the color of the line must be in the list (preferably the first one on the list). If you do not do this, the robot will not be able to sense the line since it will consider it an obstacle and will report an error if you make the robot move over the line.

  The rSense() function returns a number from 0 to 7. This number represents the sensory condition (on/off) with the least significant bit being the right-most sensor.

Each sensor is *on* if it senses a line underneath it and is *off* otherwise. In the situation of Figure 7.2, rSense() would return a value of 2 (010 in binary) because only the center sensor is seeing the line. A value of 6 (110 in binary) means that the left and front sensors are sensing the line while the right sensor is off the line. This could happen if the line made a sharp turn to the left as shown in Figure 7.2.

We can determine the status of a specific sensor by using a *binary* AND (&) operator as shown in the examples of Figure 7.3.

EXAMPLE	ACTION
if rSense( ) & 1	true if right sensor sees the line
If rSense( ) = 4	true if only the left sensor sees the line
if rSense( ) & 6	true if left OR middle OR both sensors see the line
if rSense( )	true if any sensor sees the line
a = rSense( )	
if (a = 2)	true if only the middle sensor sees the line
if a & 7	true if any sensor sees the line
if a = 7	true only if ALL the sensors see the line

Figure 7.3: The **rSense( )** function reads the three line sensors

- **A First Attempt**

Figure 7.4 shows a subroutine that can follow a line using the above logic. The routine simply turns right or left depending on where it sees the line. The while-loop creates a loop that, in this case, continues forever (or until the user stops the program). To test any of the routines given from now onwards, replace the FollowLine subroutine in the base program of Figure 7.1, with the new figures given (Figure 7.4 in this case). If you test this subroutine you will see that it fails if the line turns too sharply. We will address this problem shortly.

```
FollowLine:
  while true
    if rSense() & 1 then rTurn 1
    if rSense() & 4 then rTurn -1
    rForward 1
  wend
Return
```

Figure 7.4: This subroutine will follow a relatively straight line.

- **An Improvement**
  One problem with the routine in Figure 7.4 is that the robot does not know when it loses the line and continues moving until it crashes into a wall. Figure 7.5 shows how the robot can determine when it is no longer on the line. The robot constantly checks to see if any of the sensors are seeing the line. Since it is possible that a thin line could be between two of the sensors (and thus make the robot incorrectly assume it has lost the line, the algorithm will continue trying to follow the line until the sensors have not seen the line 50 times in a row. If you substitute this code into the base program, you will see that the robot stops shortly after losing the line. This is an improvement, but we still need to find a way to keep the robot on the line.

```
FollowLine:
  c=0
  while c<50    //exit loop if line is not seen for 50 tries
    if rSense() & 1 then rTurn 1
    if rSense() & 4 then rTurn -1
    rForward 1
    if rSense()    // if any sensor sees the line
       c = 0       // start the counter over
    else
       c = c + 1   // increment counter if no line is seen
    endif
  wend
Return
```

Figure 7.5: This subroutine knows when the end of the line has been reached

7.3 Sharp Turns Cause A Problem

As mentioned earlier, the algorithms in Figure 7.4 and 7.5 fail if the line turns sharply. The robot will do just fine if the line is relatively straight, but it will lose the line when there is a sharp turn in it.

- **Possible Solutions**
  In order to solve this problem, we need to understand exactly why it is happening. The robot fails to follow the line when the line turns faster than the robot is turning – in this case more than about a 45-degree change because the algorithm makes the robot turn about one pixel left or right for each pixel that it moves forward. When this happens the robot moves past the turn and will not see the line on any of the sensors. It continues moving forward and loses the line.

There are relatively straightforward approaches to solving this problem. We can, for example, make sure the robot stays on the line by ensuring that it does not move forward past a turn in the line. This can be done by continuing to turn until it is safe to move forward. Another solution would be to let the robot move past the turn in the line, but give it a means of finding its way back to the line.

Either of the above strategies can provide a possible solution for the robot, but they do it in a very different manner. The differences will be reflected in the behavior you see as the robot attempts to follow a line using the above methodologies. In the first case, the robot will appear to slow down when it sees a sharp turn because it executes more turning than forwarding. In the second algorithm, the robot will constantly move forward, but try to make its way back to the line after it has lost it due to a sharp curve.

You might think that the second strategy is better. After all, it should allow the robot to reach the end of the line more quickly if we can implement it properly. However, consider for a moment that the robot in question could be a car driving down a road and not just following a line on the screen. The second algorithm would indeed let the car take a shorter path to the end of the road, but it does so by letting the car take short-cuts by driving off the road when the road makes a sharp turn and then getting back on the road a little further on.

It is important to realize that neither of these strategies is necessarily better than the other. Each one has advantages and disadvantages depending on the situation and the environment.

One of the advantages of using a simulator is that you can test and improve a variety of algorithms very quickly and test them in various environments just as easily. We will develop two algorithms to implement both strategies discussed above.

- **A First Strategy**
  Figure 7.6 shows a subroutine that implements the first strategy discussed above. If you run the program, you will see the robot behaving exactly as predicted. For simplicity the code does not check if the end of the line has been reached.

```
FollowLine:
  while true
    rForward 1
    while rSense() & 1
      rTurn 1
    wend
    while rSense() & 4
      rturn -1
    wend
  wend
Return
```

Figure 7.6: This routine keeps the robot on the line even at sharp turns.

Compare Figure 7.4 and Figure 7.6, the subroutine in Figure 7.4 lost the line in a fast turn because the robot only checks once (with an **if** statement) to see if it needs to turn, and only turns once (if needed) before proceeding forward. This means that the robot can lose the line if the line turns faster than the robot can turn.

In Figure 7.6, the robot uses a while-wend loop to continue to turn as much as is necessary to stay on the turning line before moving forward. This means that the robot will not move forward until it has turned sufficiently to remain on the line. Extremely sharp turns, such as the last one in Figure 7.1B, still present a problem, but the robot performs well in most situations.

- **A Second Strategy**
  The second strategy is implemented in Figure 7.7. The routine allows the robot to leave the line when it turns sharply and then reacquire it a short distance later.

  Once the robot loses the line it cannot use the line sensors to determine which way to turn. To solve this problem, we need a way for the robot to *remember* which way it was turning the *last* time it saw the line. This will normally be the direction the robot should turn if it has lost the line. Each time the robot makes a normal turn (the if-decisions in Figure 7.7) the subroutine stores the turn direction in the variable *LastTurn* to remember which direction the robot was turning. Later in the routine, if none of the sensors are on (indicating we probably have lost the line), the robot will be able to head back towards the line. Extremely sharp turns are still a problem even for this algorithm.

In this algorithm, since it is acceptable to lose the line, we can speed up the robot's progress by moving it forward two pixels at a time. While the robot is still over the line it will turn only one degree at a time to stay on it, but if the line is lost, the robot will turn three-degree turns to help it get back on course. Notice that these choices for how much to move or turn are somewhat arbitrary. With a little experimentation, you can determine the optimum values for your situation. This reminds us again of the advantage of using a simulation. With RobotBASIC you can change the values and see how the robot responds to your changes very quickly.

```
FollowLine:
  while true
    if rSense() & 1
      rTurn 1
      LastTurn = 1 //remember which direction we WERE turning
    endif
    if rSense() & 4
      rTurn -1
      LastTurn = -1 // remember which direction we WERE turning
    endif
    rForward 2  // since we don't care if we lose the line,
                // move forward twice
    if rSense()=0
      rTurn 3*LastTurn // if we lose the line make a BIG
    endif              // turn back towards it
  wend
Return
```

Figure 7.7: This routine lets the robot find the line after it has lost it in a turn

- **Very Sharp Turns**

  The routine in Figure 7.6 stays on the line nicely and even handles 90-degree turns. However, if the line turns much more than 90 degrees, the robot can still lose the line. There are many ways to solve this problem. Figure 7.8 shows potential solution.

  The principle is that when one of the outside sensors AND the middle sensor are on simultaneously, the robot assumes that the line must be making a sharp turn. When this situation is detected, the robot moves forward so that its center is near the point where the line turns. The robot then turns until its outside sensor finds the line again.

Having done all this, the routine proceeds as before with the while-loops keeping the robot on the line in the same manner as in Figure 7.6.

When you run the program you will see that the new algorithm does in fact handle very sharp turns. You will also see that this new turning behavior happens even on moderately sharp turns, making the robot correctly follow more complex lines. However, the robot's movement is now somewhat erratic which may not be acceptable in some situations.

There are many factors that can cause an algorithm to fail. The above algorithm, for example, will not work properly if the line width is reduced from four pixels to three. Any algorithm is only a potential solution until it has been thoroughly tested in a variety of expected environments. A robot's ability to perform properly depends on the programmer's ability to predict the situations the robot is likely to face. Subsequent chapters will explore this idea further.

```
FollowLine:
  while true
    rForward 1
    if rSense() = 3
      rForward 20   //move the centre over the corner
      while rSense() = 0
        rTurn 1    //turn back to the line
      wend
    endif
    if rSense() = 6
      rForward 20   //move the centre over the corner
      while rSense() = 0
        rTurn -1   //turn back to the line
      wend
    endif
    //-- reposition over the line
    while rSense() & 1
      rTurn 1
    wend
    while rSense() & 4
      rturn -1
    wend
  wend
```

```
Return
```

Figure 7.8: This routine deals with sharp turns in a unique way, allowing it to not only handle very sharp turns, but also acquire the line if it finds it while roaming randomly.

7.4 Random Roaming With Line Following (Race-Track)

This section has been left out

7.5 Summary

In this chapter you have:

- Been introduced to several algorithms for following a line.
- Seen how proper interpretation of the sensory data can improve the robot's performance while carrying out complex tasks.
- Learned how arrays, Data and mPolygon can be used to draw more efficiently.
- Seen how an array is a more efficient way to store and manipulate data.
- Learned that some algorithms may work properly under certain environmental conditions but fail if these conditions are modified.

Now, try to do the exercises in the next section.

7.6 Exercises

1) Run the programs in this chapter to see how they perform. Add debugging statements to help analyze the robot's behavior, and find out why some of the algorithms fail on sharp turns.

2) Try to determine the optimum values for the rForward and rTurn commands (as discussed in Section 7.3) for the routine in Figure 7.7.

3) Choose your favorite algorithm from this chapter (or develop one of your own) and combine it with the *DrawObjects* subroutine in Figure 5.2 of Chapter 5 so that your robot can follow *any* line that the *user* draws.

4) The routine in Figure 7.8 is particularly sensitive to the width of the line being followed. It works great if the line has a width of 4. Try other line widths and explain the behavior that occurs. Check to see if the line width affects any of the other algorithms in this chapter.

5) Modify the line-following algorithm of your choice so that the robot will check for objects that might block its path. When one is found, the robot should turn 180 degrees and follow the line in the reverse direction. Try out the algorithm you develop with obstacles on the line.

6) The line following algorithm given in Figure 7.9 works most of the time but it does not guarantee that the robot will keep going around the racetrack in the same direction. The way the algorithm works may cause the robot to back track. Can you write a new algorithm to prevent this?
Hint: Some memory of the direction of travel may be necessary.

7) The new main program in Figure 7.9 calls *RoamAround* then calls *FollowLine*. If *FollowLine* ever finishes as in Figure 7.5, then the main program will go to the next line, which is End. Convert the main program so that it will not end, but keep roaming then following a line then roaming and so on endlessly.
Hint: Use a while loop.

Chapter 8
Following A Wall

There are occasions when it may become necessary for the robot to follow the contour of an object:

- If a robot encounters an object while moving along an intended path, it might go around the object by navigating around the perimeter of the object.
- A robot that delivers mail in an office environment, for example, might follow a wall down a hallway, visiting each office in turn to deposit its cargo and collect new mail.
- A strategy for solving a maze of corridors is to keep following the walls around in one direction (left or right).

In this chapter you will learn various strategies for enabling the robot to follow the perimeter of an object while staying close to it but not crashing into it or moving too far away.

8.1 Constructing A Wall

Before we can examine the algorithms we will need a relatively complex contour with which to test our strategies. Also we will need a base program, which we will use throughout with only a few changes to accommodate the various algorithms.

The robot will start by moving forward until it encounters an obstacle. When it encounters an obstacle the robot will abandon the moving-forward behavior and start the wall-following behavior.

Figure 8.1 shows a template with a main program and three subroutines. The main program calls the subroutines in order as they become needed and locates the robot on the screen. The line that sets the variable *TurnDir* will be needed in later sections and will be discussed there. We set the list of invisible colors and put the pen down. We put the pen down in order to leave a trail behind the robot while it is following the wall. This helps in observing the robot's behavior and gauging the algorithms' effectiveness (or lack thereof), as you will notice later. You have seen the pen feature in Chapter 4 and will learn more about it in Chapters 10 and 11 (see Appendix C.9 for more details).

The first color on the list of invisible colors will be used by the pen to draw when it is lowered since no color was specified when the **rPen** command was issued. Similarly the second color will be the default color used by the **rDFeel()** function. We will use **rDFeel()** later in the chapter.

```
MainProgram:
   gosub DrawWall
   rLocate 100,300,50
   rInvisible Cyan,Red
   rPen Down
   gosub RoamAround
   TurnDir = -1
   gosub FollowWall
End
//=========================================================
DrawWall:
    ClearScr
    LineWidth 4
    Data Wall;-161, 177,  220, 124,  375, 155,  485, 275
    Data Wall; 624, 300,  668, 370,  517, 412,  499, 320
    Data Wall; 499, 321,  389, 387,  361, 311,  369, 283
    Data Wall; 348, 235,  334, 275,  318, 223,  251, 319
    Data Wall; 161, 177,  247,-193
    MPolygon Wall,Blue
Return
//=========================================================
RoamAround:
  while not(rBumper()&4)
    rForward 1
  wend
Return
//=========================================================
FollowWall:
   //we will develop this later
Return
//=========================================================
```

Figure 8.1: This code draws a wall for the robot to follow and starts it moving forward.

The subroutine *DrawWall* does exactly that using mPolygon and the array *Wall* created by the series of Data statements, as in Chapter 7.

The subroutine *RoamAround* makes the robot move forward until it encounters an obstacle. When the robot encounters an obstacle the routine is terminated, which causes the program flow to go back to the main program, which then starts *FollowWall*. This subroutine is left empty for now. We will develop various wall following algorithms that will be substitutions for this routine.

8.2 A Basic Algorithm

In order to understand how the robot can follow a wall, imagine that you are blindfolded and are asked to stay close to a wall as you follow it to a desired destination. You would probably put out one hand (your right hand if the wall was on your right) to help you know that the wall is still there. As the distance between you and the wall becomes larger your hand will eventually stop touching the wall. You would then need to turn to your right and move forward to get closer to the wall again. If you find yourself getting closer to the wall you would have to bend your arm. To maintain your arm stretched out you will need to turn away from the wall to avoid running into it.

Figure 8.2 shows one method for telling the robot how to achieve the above logic. Replace the *FollowWall* subroutine of Figure 8.1 with the one in Figure 8.2.

```
FollowWall:
  while true
     // anything on right makes you turn left
     while rFeel() & 3
       rTurn -1
     wend
     rForward 1
     rTurn 1
  wend
Return
```

Figure 8.2: This code is a basic algorithm for following a wall.

The outer while-loop makes the robot follow the wall forever. The inner while-loop turns the robot away from the wall as long as either of the infrared sensors on the right side of the robot can detect the wall. The robot then moves forward and turns back towards the wall.

- **Problems With The Basic Algorithm**
  There are two shortcomings with this algorithm. If you look at Figure 8.3 you will see both of them. The first problem is that the robot tends to move in arcs around the wall rather than following it in a parallel line. The second problem is that the robot crashes into the first hard turn it encounters.

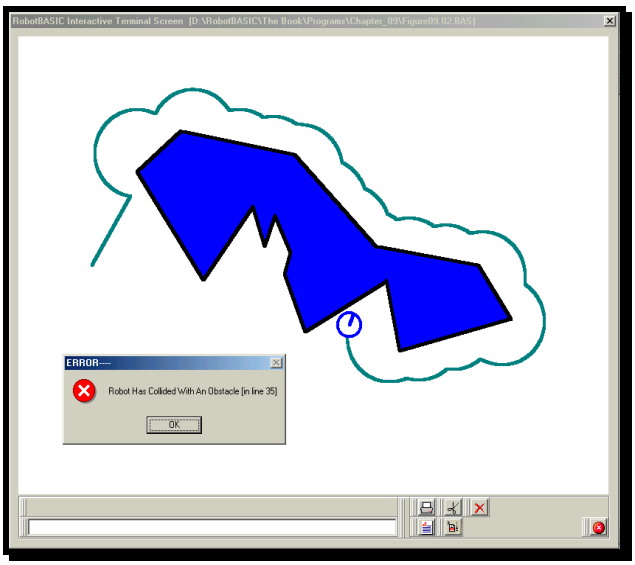

Figure 8.3: Simple Algorithm Fails.

Let us analyze why the logic failed. However before we can do this let us observe the infrared sensors while the algorithm is running. If you replace the rFeel() function with the rDFeel() function you will be able to observe the infrared beams while the robot is moving. Since no color is specified the second color on the invisible colors list (red) will be used to display the infrared beams (read about rDFeel() in Appendix C.9).

Observing the infrared beams is a great help in analyzing what the robot sees, and can give real insight into why it fails in situations you think should work. Combining rDFeel() with Debug statements can help you figure out many complicated and puzzling situations.

You will notice that the robot is moving in arcs due to the way infrared beams are tested. We are testing for either or both of the right beams (rFeel() & 3, 3 = 00011 in

binary). This means that the robot will turn away from the wall until the right hand beam is not sensing the wall, which is almost 90 degrees. The robot then forwards and turns. This forwarding and turning is the reason we get the arcs. The robot will move in an arc until it encounters the wall again and turn away 90 degrees and so on.

The reason for the crash is that the 90 and 45 degrees right sensors did not sense the wall at the angle in the wall you can see in Figure 8.3. This means that the robot will continue forwarding. Unfortunately there is no way for the robot to know that there is still part of the wall ahead and thus will crash into it.

- **Improving The Algorithm**
  To prevent the robot from turning too far away from the wall we will ignore the 90 degrees sensor. Also to give the robot the ability to see ahead of it we will test the front sensor. So instead of testing for rFeel()&*3* we will test for rFeel()&*6*. Replace the value 3 with 6 (binary 00110) in Figure 8.2 and run the program again.

  As you can see from Figure 8.4, the robot does indeed follow the wall in a straight line. However if you look closely you will notice that the robot still tends to loop around sharp corners. This happens because the robot cannot turn fast enough to follow the sharp turn because it only turns one degree for every 1 pixel forward move. We will solve this problem shortly, but first lets examine a more critical problem.

  The robot does not crash at the first sharp bend in the wall but it does crash later on. The front infrared sensor failed to detect the sharp protrusion in the wall. If you use rDFeel() you will see that it just misses detecting the sharp v in the wall. So, even though testing the front sensor helped, it still fails to catch all situations.

  You already know how to solve this problem from previous chapters. If you test for the bumper sensor along with the front sensor you should be able to catch most of the awkward situations.

- **Using The Bumpers**
  We will show how to use the bumpers with the infrared sensors to follow a wall, but before we give the new code, let us consider what modifications are needed to change the behavior from following a wall on the right to following it on the left.

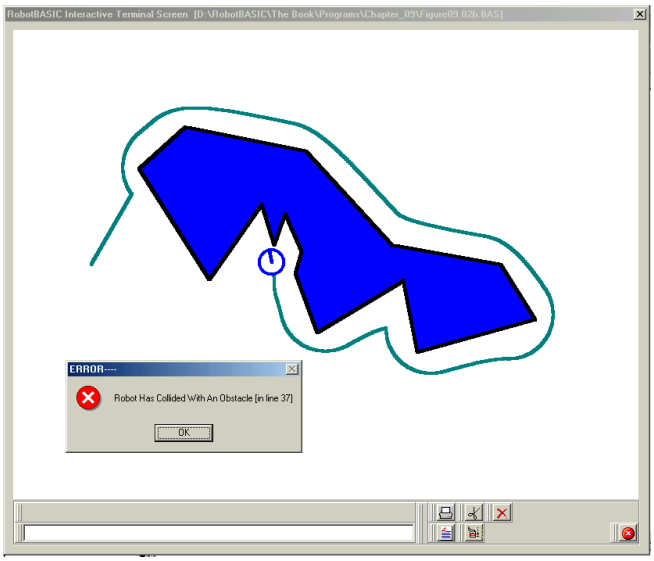

Figure 8.4: An improved Simple Algorithm.

First we will need to change the sensors used. Second, we need to turn in the opposite direction. If the wall is too close to the left we turn right and we turn left if the robot is too far from the wall.

However we want to be able to change between turning left or right easily without changing more than a variable in the program. This is why we have the line *TurnDir = -1* in the main program in Figure 8.1. This variable acts as a switch. If it is *–1* the robot will follow the wall to the left and if it is *1* the robot will follow the wall to the right.

We will also have to add some code to allow for this switch. Figure 8.5 shows the algorithm that achieves all this. Notice that we now check to see if the front bumper is closed as well as checking for the infrared sensors. Also notice how we set the variable *FN* to be used in the rFeel() & *FN* statement. Remember if you are following the wall to the right then the right 45 degrees and front infrared sensors need to be considered (i.e. 00110 = 6), and if you follow the wall to the left then the left 45 degrees and front sensors are to be tested (i.e. 01100 = 12).

Run the program and try changing *TurnDir* to 1 and see how the robot now follows the wall to the right instead of to the left.

The program in Figure 8.5 performs reasonably well. As mentioned earlier though, the robot still arcs far away from the wall when it rounds a sharp corner. The algorithm can still be improved further.

```
FollowWall:
  if TurnDir > 0
     FN = 6
  else
     FN = 12
  endif
  while true
     while (rFeel()&FN) or (rBumper()&4)
       rTurn -TurnDir
     wend
     rForward 1
     rTurn TurnDir
  wend
Return
```

Figure 8.5: This program turns away if the wall is seen by either the infrared sensor OR the bumper sensor.

8.3 Staying Close on Sharp Corners

This section has been left out

8.4 A Different Approach

This section has been left out

8.5 Summary

In this chapter you have:

- Learned how to make the robot follow a wall using a variety of algorithms and sensors.
- Seen the process of developing and debugging an algorithm.
- Discovered how unpredictable situations can arise and be difficult to analyze.
- Learned that programming a robot requires not only an understanding of the sensors but also the ability to deal with them using both logical and bit-wise operators.
- Learned about the extended rRange() function.
- Seen further uses for arrays, MPolygon and Data commands.

Now, try to do the exercises in the next section.

8.6 Exercises

1) Run the subroutines in this chapter to see how they perform. Try adding debugging statements to help you analyze the robot's behavior.

2) Try all the algorithms in this chapter with the wall given in Figure 8.10. Can you determine why some algorithms fail? If an algorithm does not fail can you see why it may be considered better or worse than another that also did not fail?

3) In the algorithms of Figures 8.8 and 8.6 (modified as discussed) try changing the parameters to see how they affect the robot's behavior. In Figure 8.8 how would you make the robot stay closer to the wall? What would be the minimum number to give *RangeLimit*? Why?
   Hint: The rRange() function returns a value relative to the front. The front is in line with the robot's center. So an rRange(*90*) value of 20 will mean that the object is almost touching the robot at its right side.

4) Draw circles, rectangles and triangles and then test the algorithms with these types of simple objects. Which algorithms follow the contour faithfully (i.e. the robot draws a circle around a circle and a rectangle around a rectangle etc.)? With the algorithm of figure 8.8 you can specify exactly how far the robot will be. Try following the objects at various distances (closer and further).