Digital ICs

Autonomous Golf Ball Collector

Software Architecture & Behaviour

Final Project Report

Student: Amr Nosir

Std. no.: 100917017

Supervisor: Maitham Shams

Date: 11 Apr 2017

Table of Contents

List of Figures

Abstract

Collecting golf balls is a menial task, where all current solutions require human labour on some level. Additionally, large scale solutions require the use of trucks or other fuel dependent vehicles. Our goal was to create a small autonomous robot that can detect golf balls in the surrounding area, retrieve them, and avoid obstacles along the way. The robot used a Raspberry Pi 3B as its on board microcontroller chip that controlled all behaviour apart from the object detection algorithm, which was done on a separate computer through a server-client connection. The robot is currently capable of performing its core functionality, and would be suitable for performance in more complicated situations with further testing and fine tuning. This report discusses the development of the software architecture and behaviour of the robot, as well as their results. Each sections was written so it can be understood on its own, however for a clear picture, the whole report should be read.

1 Introduction

1.1 Overview & Specifications

Our goal was to build a fully autonomous golf ball collector. The robot would be capable of recognizing golf balls in its field of view and their position relative to itself. The robot would then attempt to retrieve each ball detected and navigate around obstacles in its path. Once done, the robot will rotate to face a new direction and repeat the process until it meets a pre-set end condition. The project was divided into four main components where each team member is responsible for one. Those components are: The camera and object detection, the vehicle, the arm, and the software that integrates the components.

1.2 Motivations & Significance

Collecting golf balls is a menial task, where all current solutions require human labour on some level. Additionally, large scale solutions require the use of trucks or other fuel dependent vehicles. Our goal was to create a solution that eliminates those two requirements so it can be convenient and environmentally friendly. A single robot would be very useful for hobbyists, while multiple robots can be used by establishments to cover wide areas.

The core concept behind the robot can be expanded further to meet other requirements, such as tennis ball or garbage collection. The reason golf balls were selected as the target was due to their uniformity, which made the development of an object detection algorithm for them simpler. However, by increasing the sophistication of the object detection algorithm and the arm, other objects can also be considered in future expansions.

2 Professional Considerations

For any engineering project, the engineers working on it must make non-technical considerations. These considerations affect both their final product as well as their workflow and ensure that the project proceeds smoothly without issues. They also ensure that the final product will be ethical and safe to use by users of all backgrounds. The considerations are as follows.

2.1 Health & Safety

The safety of any product to its users is a crucial concern for its developers. While there are no such considerations to be made for the development of embedded software, the team as a whole ensured that the final product would have no exposed wires that could become a risk. Also, the team members responsible for the mechanical parts of the project made sure that their components were safe while in use. Finally, although the electrical components handled were low-power, they were still handle with utmost care throughout development.

2.2 Engineering Professionalism

Each team member displayed the professional conduct expected from an engineer. Regular meetings were conducted to discuss progress and any issues that arise. Within each meeting and throughout the project, each member maintained a professional, objective-oriented approach which resulted in the overall success of the project. Additionally, we were forthcoming with any difficulties encountered along the way, which prevented unforeseen catastrophic failures from showing up towards the end.

2.3 Project Management

The project was divided into four parts, where each member was responsible for one of them. During our first meeting, each member picked the part that suited their skillset with the agreement of

the other members. A broad schedule was also agreed upon to accommodate for integration and testing, where each member was expected to complete the majority of their individual parts by the end of January. Regular meetings allowed the team to know how everything was coming along and maintain the alignment of individual goals with the project requirements. In addition to progress updates during the meetings, ideas were freely exchanged and discussed. As a result, by the time of this report's writing the vast majority of the project has been completed and what remains is only a matter of further testing.

Personally, the software developed was divided into milestones to be achieved by certain points in time while allowing time for both individual testing, project integration, and unexpected circumstances. During the first semester development fell behind schedule, however, extra work was exerted during the winter break in order to catch up. This personal schedule successfully integrated with our team's schedule and contributed to the success of the project.

3 Theory & Techniques

3.1 The Microcontroller

The first step was identifying what kind of microcontroller will be used. There were two options available, the Arduino Uno and the Raspberry Pi 3B. While the Arduino Uno is cheaper and easier to program, it is limited by its ability to only run one program at a time and its lower computing power compared to the Raspberry Pi. Therefore, given that the object detection program and the arm would need to be separate from the rest of the program and will demand significant processing power, it was decided that the Raspberry Pi 3B would be the microcontroller chosen.

The Raspberry Pi will use the Linux-based operating system Raspbian, which is officially supported by the Raspberry organization. The options available for writing the main program were C,

C++, Java, and Python. Since an object-oriented approach to the program would not be needed, both Java and C++ were ruled out. Python, although more readable and less error-prone, it is slower and less efficient than C. Therefore, C was chosen as the programming language. Initial software design and testing is done on a PC running a virtual machine before being tested on the raspberry Pi. This is possible because the virtual machine with a Linux based operating system similar to the Pi, which used the POSIX Application Programming Interface (API). POSIX is a family of standards specified by the IEEE Computer Society that standardizes the API of multiple Unix and Unix-like operating systems (including Linux) to allow for easy porting of software between those operating systems.

3.2 General Software Development Approach

For the development of the software architecture and the robot's behaviour, the requirements were first gathered from the supervisor and the other team members. Once a clear picture of what needs to be done was painted, the overall approach and methods to be used for the software's development were outlined based on industry practices.

The general approach used was a top-down approach. The software was developed starting at the highest level. Each level would be more specialized than the level above it and would be used as a component of that higher level. Development would begin on the next level, once the higher level was tested and found to be in a satisfactory state. As a result of the requirement analysis, the overall software breakdown can be seen in figure 1, where the Shell program is the highest level. Each level is discussed further in the report.

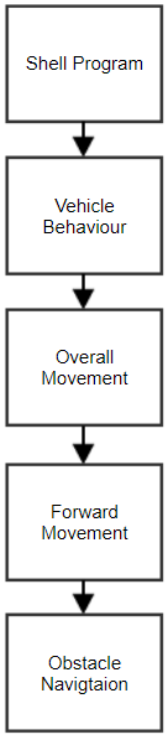


Figure 1: Software Breakdown

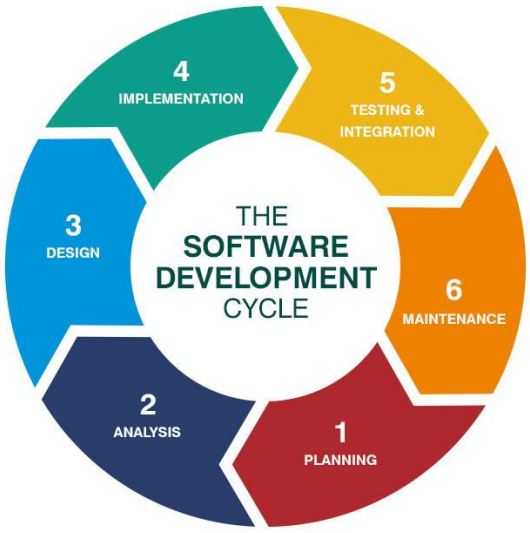


Figure 2: The Software Development Cycle

The development of each level was based on the software development cycle seen in Figure 2. Planning and analysis were collapsed into a single step that specified what this level of the program should be able to do. The next step was designing a flowchart that outlined the program's blocks. Once implemented, each level was extensively tested on its own, then further tested after integration with the rest of the program. Statements were printed on the screen to indicate how the robot is behaving. Finally, maintenance involved making any necessary fine tuning and editing needed as the program increased in complexity and approached completion.

The majority of programming and testing were performed on a virtual machine that ran a Linux based system similar to the one on the Raspberry Pi. Since the operating systems are very similar, a program developed on one, would usually work on the other without encountering compatibility issues. This decision was made out of convenience, since it allowed for working on the project independently from accessibility to the Pi. However, a regular test was performed to ensure that there are indeed no incompatibility issues. The virtual machine used was the COMP2404-2406 machine provided by Carleton's School of Computer Science and was used by the author in one of his courses.

3.3 Specific Programming Methods

More specifically, within each level efficiency was kept in mind. Functions were created to increase modularity and improve readability. Functions that would be very similar were collapsed into a single function that had slightly different behaviour based on its inputs. This was done to ensure that the

writing of unnecessary or repetitive code was limited. Therefore, memory and computational efficiency were maintained from the get go, with further optimization performed when noted.

Libraries of related functions were created for multiple reasons. Most importantly, since, some functions would need to be used in multiple levels of the program, such as rotating, having only one source file for that function would be more efficient. Only one source file will need to be edited when needed and each source file can deal with a specific portion of the behaviour. One of those source files was dedicated for defining all fixed values, and simplified changing those values, and file paths and can be seen in Appendix A. Additionally, those functions can be used in the future in other programs with ease by incorporating those source files and making a few edits where needed.

Each level was implemented by first programming and testing its core functionality. Next, variations and exceptions were programmed in related batches and the whole level was tested. This iterative implementation style allowed for easier debugging, since complexity was not added until the base was well tested. The levels were also regularly tested as a complete program to ascertain that no issues were encountered when the whole program ran.

4 Shell Program

The shell program is the highest level of the software and is the one ran to start up the robot. It is responsible for integrating the camera's analysis of the area with the robot's behaviour. The code for the shell program can be seen in Appendix B.

4.1 Design

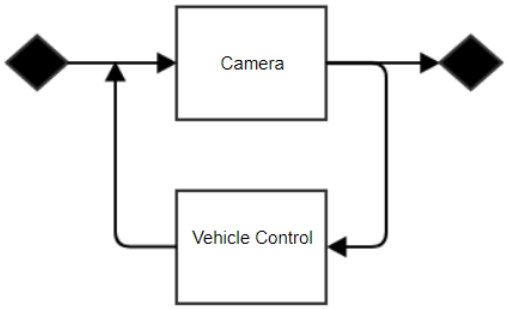


Figure 3: Shell Program Flowchart

As seen in figure 3, the shell program will cycle between the camera and vehicle control programs until an exit condition is met. The exit condition being the number of times the robot is needed to go through the routine. There were two contending designs on how to implement the shell program. The first was having vehicle control be part of the shell program. The second was having vehicle control be a separate program like the camera's.

The second design was selected for a few reasons. First, the object detection program was at first intended to work on the Raspberry Pi. Since it is a processing intensive program, it would have needed as much of the Pi's resources as possible to work efficiently. A larger shell program would take up more run time memory while idle, which would be more useful for the camera. The second reason was the benefits of having vehicle control be an independent program. By being independent, vehicle control would be independent from the start up and shutdown requirements of the robot. This makes

tweaking either the shell program or the vehicle control program a much simpler task. It would also allow both programs to be used in different situations if needed. For example, this way the vehicle control program can be used as part of a bigger software in the future.

4.2 Implementation

To implement the above design, the shell program, called the parent here, must be able to execute the other programs then return and continue from where it left off. To do so the fork() function is used. This function creates a duplicate process called the child. When a program is started it is given a process ID (PID) that is a positive integer. A child however, is always given a PID of 0, thus allowing the child to execute different code using an if statement check. The fork() function uses a copy-on-write approach to when duplicating the process, which means that it creates a new copy of the data only when that data is changed. Hence the function is overall lightweight, making it ideal to use for executing external programs. Meanwhile, the parent can be made to wait for the child process to exit before performing its task using the wait() function, this includes waiting for programs called through the child to exit.

To execute an external program in C, the execl() function is used. To do so, execl() must be given the path to the program, the name of that program, and any arguments needed to run it. The function replaces the current process, which prevents any code written after it from being executed. Therefore, using fork() is necessary to avoid overwriting the rest of the program. The loop implemented in the shell program uses two fork() into execl() routines, one for each program.

5 Vehicle Behaviour

The second level of the software deals with controlling the vehicle's behaviour. As mentioned in the discussion of the shell program, vehicle behaviour will be controlled by a program called vehicle control. This level of the software deals with all the logic involved in dealing with situations before and after movement. It reads a text file with the object detection's results, then makes decisions accordingly. The colour-coded flowchart of this program is shown in figure 4. Each colour represents a portion of the program that will be discussed further in the subsections. The function called RetrieveBall() deals with overall movement, which corresponds to the third level shown previously in figure 1 and will be discussed thoroughly in section 6.

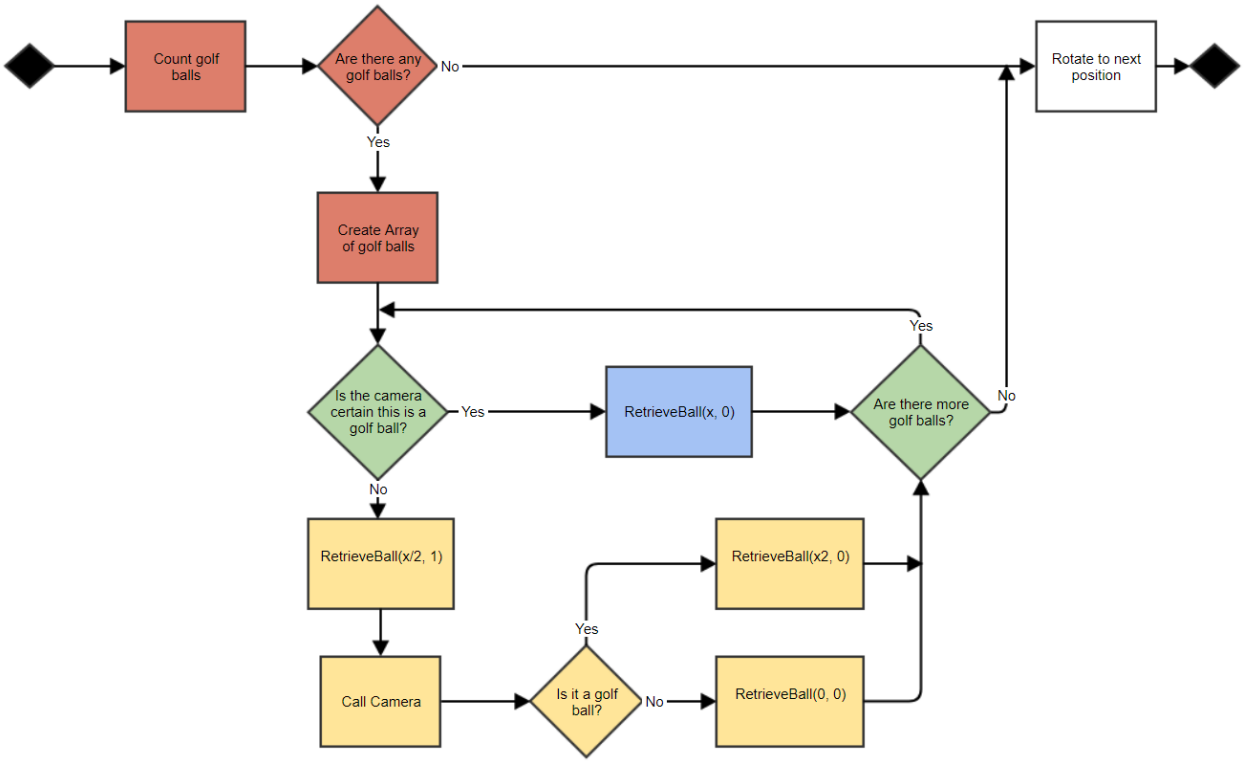


Figure 4: Vehicle Control Flowchart

5.1 Overall Design

Once the object detection algorithm is done analysing the picture captured by the camera, it outputs a text file with data on any ball detected. This data is:

- **Distance** to the ball in millimeters
- **Angle** of rotation needed to face the ball
- **Check,** which is needed if the object detection algorithm is uncertain of the conclusion that the detected object is actually a golf ball

After discussion with the teammate working on the object detection program, we decided that the data would be separated by space bars and that each line in the file will correspond to a ball. The program would read the text file and store its data in local variables. It would then attempt to retrieve each ball, based on the information it has. If the object detection algorithm was uncertain about a ball, the vehicle would stop at a halfway point then run the camera program again. Finally, the vehicle would rotate to face the next position the robot will need to collect balls from. The degree of rotation is fixed and based on the camera's field of view. The code for vehicle control can be seen in Appendix C.

5.2 Number of Balls

As shown in figure 5 and in red in figure 4, the first task the program performs on start up is counting the number of golf balls that were detected. If there is no text file or the text file is empty, it will indicate that this area is clear and the vehicle will rotate to face the next area. If at least one golf ball was detected, an array will be

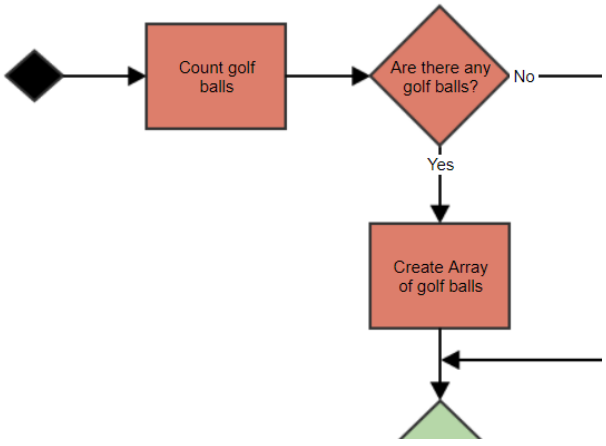


Figure 5: Vehicle Control - Step 1

created to store it. The implementation of this function can be seen in Appendix D.

Each ball is stored as a struct, a composite data type, called Ball. The ball struct has five integer values: distance, angle, check, offsetDis, and offsetAng. The first three values are provided by the object detection algorithm, while the others may or may not be used based on the value of check and are therefore initialized as '0'. Check is the value that indicates uncertainty, '0' means the detected object is certainly a golf ball and a check is not needed. On the other hand, '1' indicates a check is needed to be certain of the result. It is implemented as an integer instead of a boolean because instances may arise where different levels of uncertainty may require different responses. This makes future expansions easier without affecting current performance.

The array created will be of type Ball and will be created using the number of golf balls detected as its size. A static array was used instead of dynamic allocation of memory because none of the benefits of dynamic memory were needed, therefore only complexity would have been added. This is due to the number of balls being fixed and the vehicle control program needing to be rerun for each area. Hence, all the memory allocated to an array will be used and the array's size will not be changed, which makes dynamic memory allocation unnecessary. Currently, vehicle control finds the number of balls by counting the number of lines in the text file, then reads the file again to enter the data into the array. This will be changed so that the first line of the text file will have the number of balls, such that the program will only need to read the entire file once.

5.3 Handling Each Ball

The program will loop through each ball stored in the area and attempt to retrieve it. How it exactly attempts to do that depends on the certainty that the detected object is actually a golf ball. The conditions are shown in green in figure 4. Also in figure 4, blue corresponds to the golf ball certainty route, while yellow corresponds to the uncertainty route. This is further elaborated on in the

subsections below. In figures 6 and 7, ‘x’ refers to the distance to the ball given by the object detection algorithm. Once all golf balls are handled, the vehicle is rotated to the next position and the program exits.

5.3.1 Certainly a Golf Ball

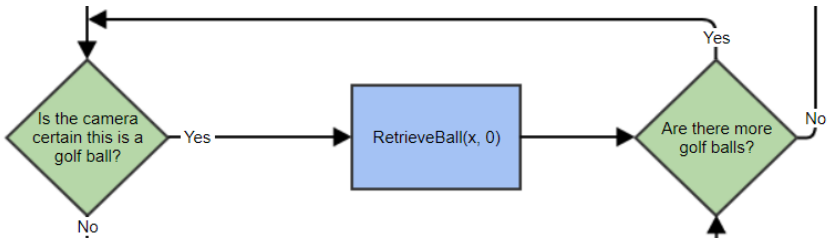


Figure 6: Vehicle Control - Step 2 a

As shown in figure 6, if the object detection program was certain that the detected object is a golf ball, it returns a check value of ‘0’ which means it does not need to be rerun for this ball. The overall movement function is called and proceeds to retrieve the ball.

5.3.2 Might be a Golf Ball

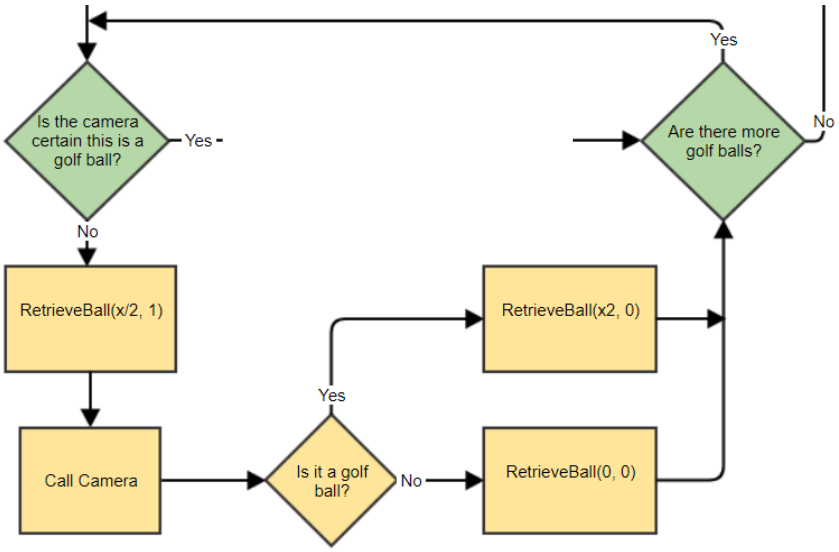


Figure 7: Vehicle Control - Step 2 b

Figure 7, on the other hand shows how the program would proceed if the value of check was ‘1’ for the ball. First, it would store the current values for distance and angle in offsetDis and offsetAng, delete the current text file, then move to the half way point ‘x/2’. There, the flow will return to vehicle control and the camera program will be called to rerun the object detection algorithm from a closer point. This will create a results text file that will be used to update the information on the current ball. The vehicle will then either go back to base or retrieve the ball. To be able to get back to base the offset values stored will be used to compensate for deviations.

6 Overall Movement

This level of the software deals with the vehicle's movement from a higher-level perspective. It deals with how the vehicle should move throughout its routine for each ball. The final implementation was done through a function called retrieveBall(). This function would take five data points that correspond to the ball's information and use them to make decision on how to proceed. Therefore, retrieveBall() is capable of handling all the movement requirements of vehicle control.

6.1 Design Process

The first step of designing the overall movement function was identifying the core movement pattern the vehicle is expected to follow. This pattern is shown as four steps in figure 8. This pattern was then expanded to outline seven specific steps that will be executed to accomplish the core functionality.

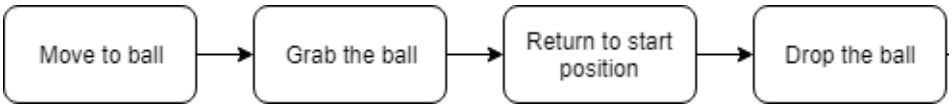


Figure 8: Overall Movement Pattern

These steps are shown in red in figure 9. The pattern was then expanded to include special cases as well in order to avoid the creation of multiple similar functions. The expansion is shown in green and blue, also in figure 9. The result was having retrieveBall() be successfully used as the only overall movement function.

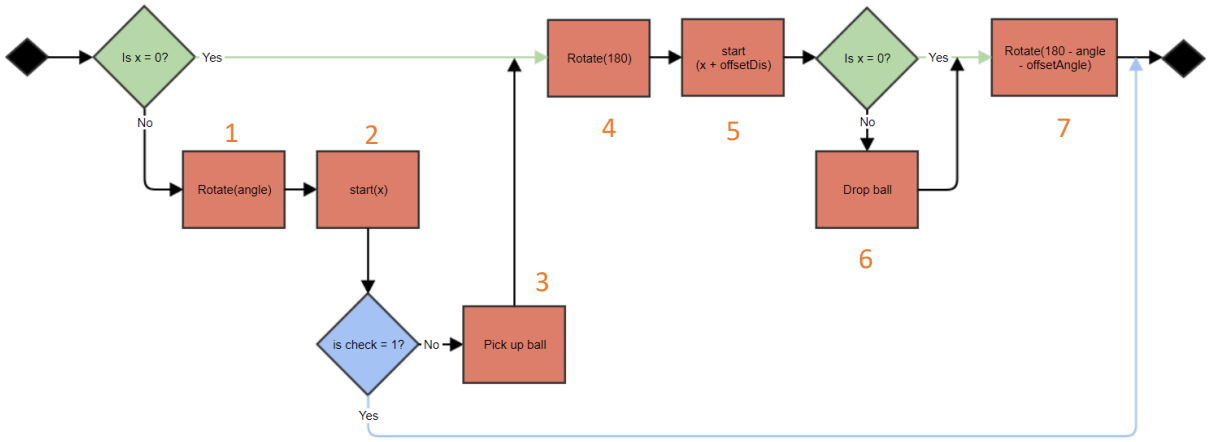


Figure 9: RetrieveBall() Flowchart

6.2 Final Function Design

RetrieveBall() takes in five inputs that correspond to the values stored for each ball. Those are as follows:

- **Distance 'x'** - The straight line distance the vehicle needs to move
- **Angle** - The direction the vehicle needs to move in
- **Check** - Whether the camera needs to perform another check
- **OffsetDis** - The straight line distance the vehicle needs to move
- **OffsetAng** - The angle the vehicle rotated to get to its current position at the beginning

Based on the values of each of these values the vehicles exact behaviour will vary. RetrieveBall() will have the vehicle perform all seven steps, unless one of two conditions are met as seen in figure 9. If the camera needs to be rerun the function will exit after step 2, shown in blue above. If the distance given is '0', the function interprets this as an instruction to return to the starting position bcause the camera rerun found that the previous result was a false positive. The code for the retrieveBall() can be found in Appendix E.

6.3 Rotate()

Rotate() is a supporting function that is used in multiple levels of the software. As the name suggests, it is responsible for turning the vehicle a specific number of degrees. It would turn clockwise if given a positive value and anticlockwise if negative. The exact implementation of how rotation is performed was done by the team member responsible for the vehicle. The function described would simply output a text statement when executed while testing. During integration of the software with the vehicle, rotate() was expanded to include a check that selected the input and output pins based on whether the given value was positive or negative. It is implemented in the same source file as forward movement and can be found on Appendix F.

6.4 Start()

The start() function is an intermediary function between the overall and forward movement levels. It initialized all the variables needed to start forward movement towards the destination. This function was needed due to the design requirement of having forward movement being implemented by a single function and the use of forward movement in obstacle avoidance. By using start(), the vehicle ensures it reaches its final destination regardless of obstacles encountered. The details of start() and forward movement are discussed in detail in section 7. It is implemented in the same source file as forward movement and can be found on Appendix F.

6.5 The Arm

To pick up and drop the ball in steps 3 and 6, respectively, the arm needs to be controlled. After discussions with the team member responsible for the arm, it was understood that effective control of the arm could not be done in C and would be better controlled through python scripts. To run the scripts, the system() function was used. System() allows for the execution of terminal commands within a C program. Therefore, by having two scripts for picking up and dropping the ball and using the

terminal commands for running them through system(), the arm could be easily controlled. They are implemented through two functions that can be found in Appendix E.

6.6 Offset Values

As mentioned previously offsetDis and offsetAng are updated for a ball that needs to be confirmed before the vehicle moves. Because the distance and angle to the ball from the start position would be overwritten to those at the new position, the vehicle needs to be able to return to its original position at the end. By adding offsetDis to the current distance being used the vehicle will be able to return to the starting position. OffsetAng on the other hand is used to face the initial position once movement is complete. The vehicle is guaranteed to return to its exact initial position and orientation under the assumption that the vehicle will not need to rotate to face a new direction after confirming the existence of a golf ball. It should be noted that overcoming this assumption is part of the future plans discussed in section 10.2.

7 Forward Movement

This level of the program deals with moving the vehicle in a straight line towards a destination or for a fixed distance. It is implemented through a function called forward(). Through several variables and checks, the exact behaviour of forward() is tweaked to meet slightly different requirements. While retrieveBall() handles what movements the vehicle should perform, forward() handles how moving forward is executed. This level and the obstacle navigation level are interdependent, and therefore where developed simultaneously for the most part. However, forward movement was treated as a higher level during development and for this report because it is more important for the core functionality of the robot than obstacle navigation. Additionally, obstacle navigation is considered an exceptional situation that breaks the usual flow of the program.

7.1 Design Process

Much like retrieveBall(), design aimed to create only one function that executes forward movement. This aimed to reduce duplicate code and overall complexity, as well as improve readability and adaptability. This increases the functions universality, flexibility and ease of debugging. The result was a forward movement function with flexible behaviour that can be easily called by any function that needs it.

Before designing how the function will work, the parameters used to describe the problem needed to be established. To describe the vehicle's positioning, an x-y-axis system is used, where the x-axis is the direct path to the destination and the y-axis is the perpendicular distance from that direct path.

The process of designing forward() started with the core functionality shown in red in figure 10. Forward() would select how to determine the distance to travel, move a unit (u) amount of distance, update the determinant, then check if it is done. Exceptions to this flow arise from encountering obstacles and how those encounters affect the vehicle's movement towards its destination. Therefore, the exception flows, shown in red in figure 10, had to be developed in conjunction with the development of the obstacle navigation level. Each section is discussed further below.

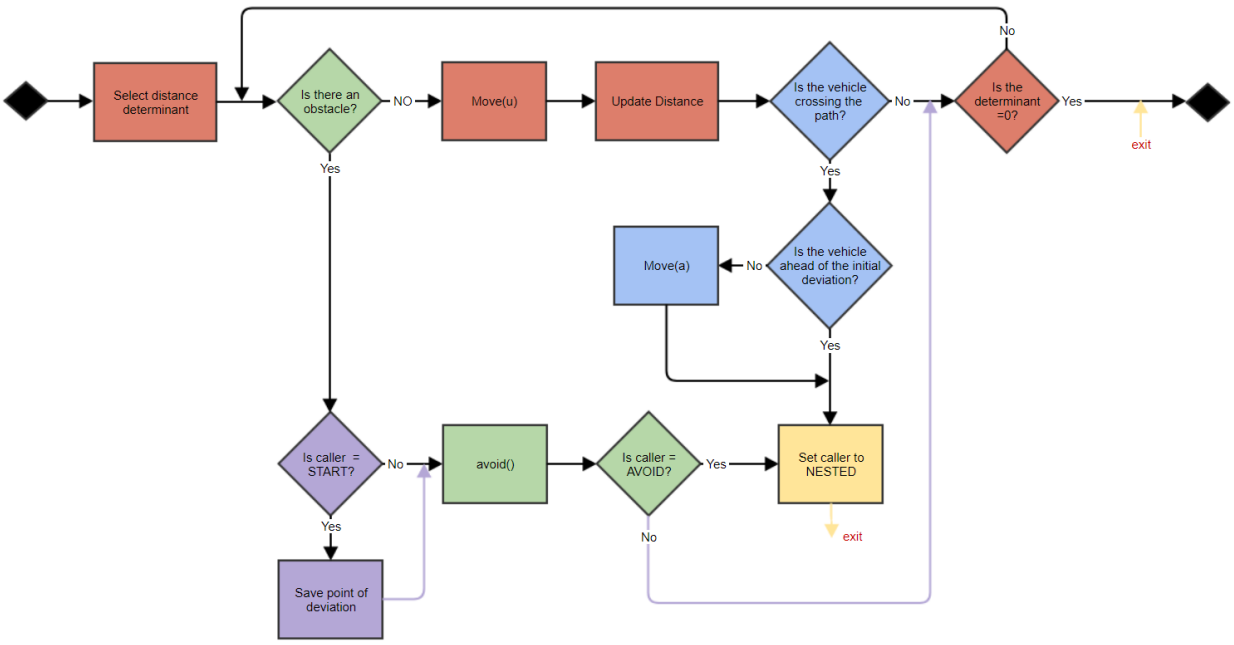


Figure 10: Forward() Flowchart

7.2 Overall Design

Forward() takes six integer variables as inputs. These are as follows:

- **Distance** - A fixed distance to be moved
- **RemainingX** - Measures the x-axis distance from the destination
- **RemainingY** - Measures the y-axis distance from the correct path
- **Orientation** - The direction the vehicle is currently facing
- **Caller** - An ID identifying how the function was called
- **DevPointX** - The x-axis point at which an obstacle is encountered on the correct path

Out of these six variables only distance and caller are not pointers. The pointers are used to keep track of the vehicle's status until it reaches its destination. The caller variable is also used as the return value of the function to provide information on how forward() was exited. The code for this function can be found in Appendix F.

There are three values for caller. The values affect the behaviour of forward() and the obstacle navigation functions. These caller IDs are:

- **START** - Ensures that the vehicle reaches a specific destination
- **AVOID** - Used to move fixed distances
- **NESTED** - As a return value, indicates an obstacle was encountered during an AVOID call

Each caller ID will be discussed in more detail in the sub-sections below.

7.2.1 START

As mentioned previously in the discussion of retrieveBall(), an intermediary function start() is used to call forward(). Start() takes in distance as its only input, then initializes all the pointers to be used by forward() and sets the value of remainingX to the distance. It then calls forward() using the START caller ID. START indicates that the vehicle needs to reach the final destination regardless of obstacles on its way. The vehicle reaches the destination when remainingX and remainingY are '0'. The logic of the program ensures that this is the case and its specifics will be discussed further below.

7.2.2 AVOID

To avoid obstacles, forward() is only needed to cover a fixed amount of distance. Forward() may also need to be overridden when another obstacle is detected. The exact logic for this is discussed further in section 8.1, but this is sufficient for this section. By using AVOID to call forward(), the function knows to either travel the fixed distance, or break the loop off when an obstacle is encountered.

7.2.3 NESTED

As mentioned above, if an obstacle is detected during a forward() called by AVOID, the program's flow needs to change, to indicate this, caller is changed to NESTED in such situations and is checked for by caller. Additionally, if the vehicle crosses the x-axis while in an obstacle avoidance

routine, the NESTED caller ID is used to override the routine and have the vehicle proceed in the correct path. The details of returning to path can be found in section 8.2.

7.3 Core Movement

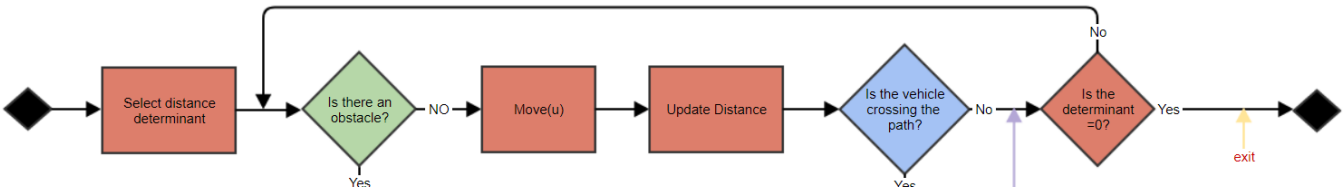


Figure 11: Forward() Core Movement

As shown in red in figure 10 previously and in figure 11 here, the core movement involves selecting a determinant for the distance, then moving units (u) of distance until that determinant is '0'.

7.3.1 The Determinant

The determinant is the value used by the movement loop to determine whether it reached its destination or not. It is a pointer that points to either the value of remainingX or the address of the distance value passed to forward(). If START is the caller remainingX is used, else the value of distance is used.

7.3.2 Move(u)

This is the part of the function that translates to actual movement when the entire project is put together. During software testing, 'u' was given a fixed value of '10' that was used to update distances, then display them as text on the screen. While implementing the function with the team member responsible for the vehicle, move(u) was replaced with a delay of fixed time to keep the motors running. The values of distances were transformed into the number of loops needed to cover the distance based on the time taken to execute the conditional code plus the delay.

7.3.3 Orientation

Keeping track of the orientation of the vehicle is necessary to keeping track of the robot's position relative to the destination. Since obstacle navigation uses a series of right angle turns to avoid obstacles and is the only source of turns during forward movement, there are only four orientations tracked. Those are FORWARD, BACKWARD, RIGHT, and LEFT, with FORWARD being the direction facing the destination at start, as shown in figure 12. Orientation is initialized as FORWARD when start() is used to call forward() from the overall movement level.

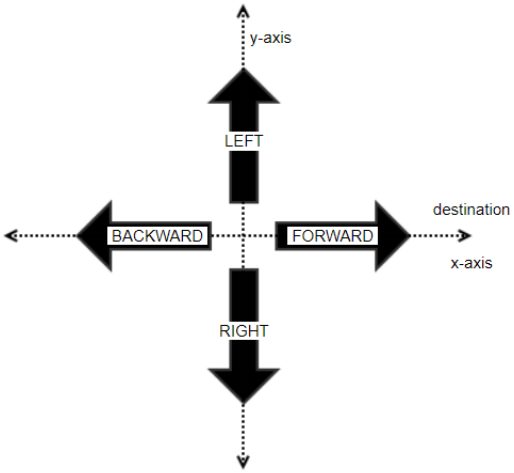


Figure 12: Orientation Key

7.3.4 Updating Distance

Distance is updated in two ways. First is updating the values of remainingX and remainingY based on the orientation of the vehicle using a dedicated function. This is done regardless of the caller ID and allows the program to always know where the vehicle is currently positioned relative to its final destination. The second method updates the value of distance and is only used if the caller is not START. Orientation is not considered for the second method because unless the caller is START, the vehicle is trying to move a fixed distance in its current orientation.

7.4 Obstacle Handling

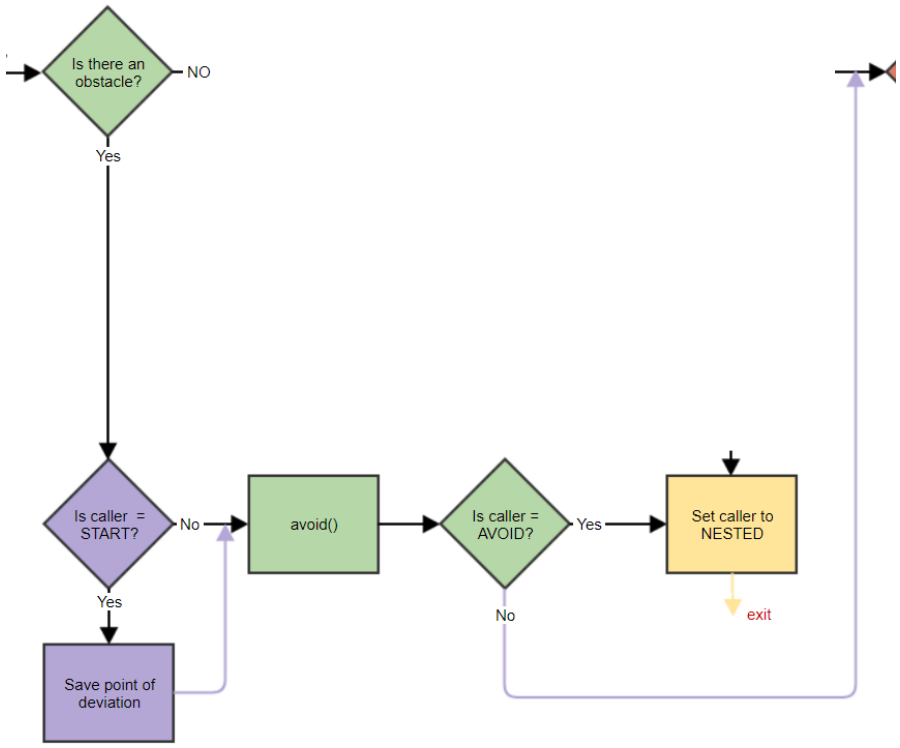


Figure 13: Forward() Obstacle Handling

The logic for handling the detection of an obstacle is shown in green and purple in figure 10 previously and here in figure 13. When an object is detected by the sensor the usual progression of forward() is paused and the program can then proceed in one of two ways based on whether the caller is START or not.

7.4.1 Caller is START

If the caller is START, the program follows the purple paths shown in figures 10 and 13. The first step taken is to save the x-axis point of deviation in devPointX. This is necessary for path correction after navigating a complicated obstacle. How devPointX is used is discussed further in section 8.2. The program goes to the obstacle navigation level through avoid(). Once the obstacle or obstacles are navigated the vehicle continues following the forward() loop. In this case the value of the determinant

does not need to be updated, because throughout obstacle navigation the vehicle's position was updated. This includes updating the value of remainingX which the determinant in a START-called forward() points to.

7.4.2 Caller is not START

Since the only other caller ID used to call forward() is AVOID, this means that if an object is detected while moving, the function will always follow the green path in figures 10 and 13, then exit and return NESTED. This is necessary to avoid unnecessary movements while avoiding multiple obstacles that make the program inefficient.

7.5 Crossing the Path

On rare occasions while avoiding an obstacle, the vehicle will end up receiving an instruction that would have it cross and overshoot the correct path if followed. To eliminate this issue, if the vehicle is facing RIGHT or LEFT, the forward() function will check if remainingY is '0' after every distance update. This is shown as the blue path in figure 10 previously and in figure 14 here. There are two conditions for how the vehicle will proceed after branching out from the correct path based on devPointX. If remainingX is less than devPointX, the vehicle stops and returns NESTED before exiting the function. By returning NESTED to obstacle navigation, it will override its current flow return to the correct path using right-angle turn. Otherwise, the vehicle is crossing the correct path at or before the point it first deviated from the correct path. If the vehicle attempts to return to the correct path here, it will get stuck in a loop it can't escape. To avoid that, the vehicle moves a fixed distance 'a' using forward() and the AVOID caller ID before attempting to correct its path.

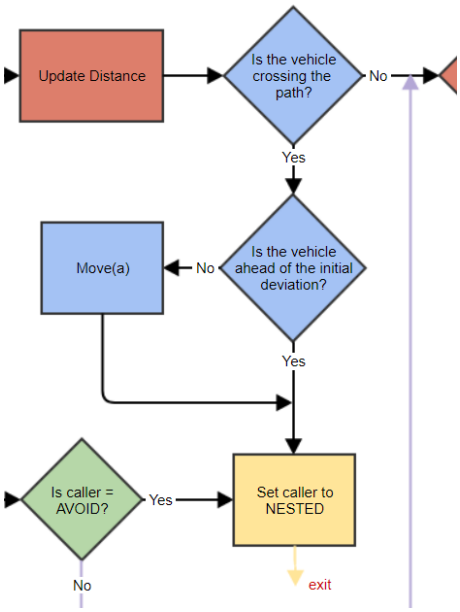


Figure 14 Forward() Crossing the Path

8 Obstacle Navigation

Obstacle navigation is the final level of the software. It deals with avoiding obstacles detected during forward movement and path correction when needed. As mentioned above it was developed almost concurrently with the forward movement level and both calls and is called by the forward() function detailed previously. All functions in obstacle navigation that call forward() use the AVOID caller ID. There were two distinct requirements needed by this level of the program, avoiding obstacles and path correction, therefore two main functions were developed, avoid() and returnToPath() respectively. These two functions are detailed here.

8.1 Avoiding an Obstacle

Avoid() is the function used to navigate around a detected obstacle. It attempts to do so through a series of right-angle turns and fixed distance forward movements. Additionally, avoid can be recursively called to navigate around large or irregular obstacles.

8.1.1 Design Process

There were three contending designs while designing the program.

8.1.1.1 Design 1

The First design would have the vehicle move along the y-axis after detecting an object in front of it. It would then attempt to return to the correct path through a single diagonal movement. This design is demonstrated in figure 15 below, showing how it would need to behave for a slightly irregular obstacle. The benefit of this solution is that it minimizes the total movement and number of turns the vehicle would need to usually make. However, its implementation would have been complicated for several reasons. To be able to reliably return to the correct path, the vehicle needs to be able to accurately know its current position relative to its destination. Although this is doable by decomposing the diagonal movement into x and y axis components, as the obstacles get more complicated doing this

would become time consuming quickly. Distance updating would require keeping track of more than four orientations in addition to doing the math for diagonal distance decomposition before it can update the covered distance.

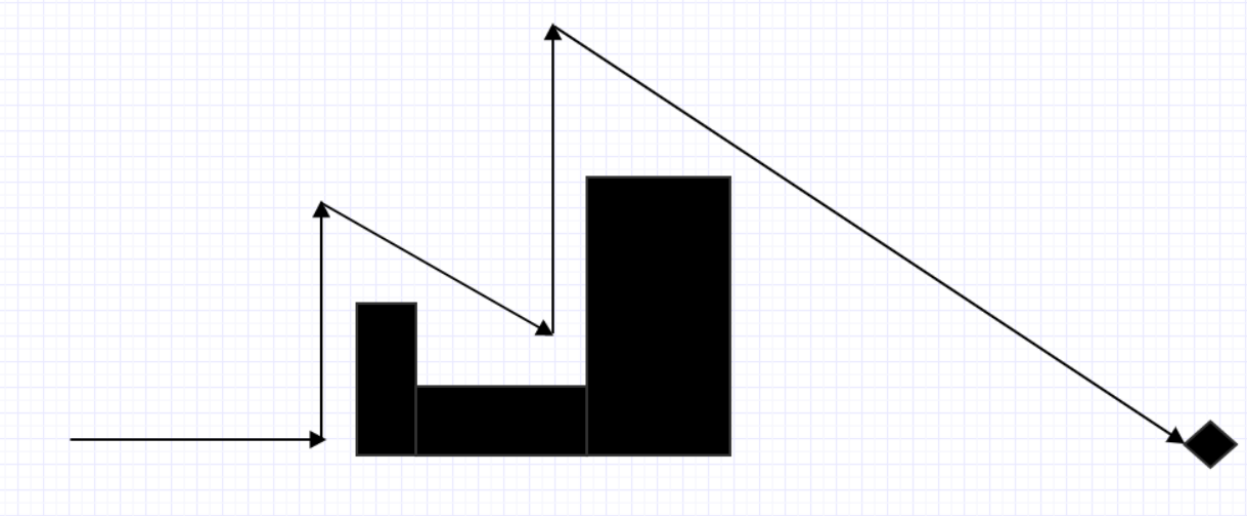


Figure 15: Avoid() Design 1

8.1.1.2 Design 2

The second design involved having the vehicle perform a series of right-angle turns and fixed distance movements. How this would work can be seen in figure 16 below. Once the vehicle detects an obstacle, it will enter a predetermined routine that is only altered by the detection of another obstacle. Eventually, the vehicle will attempt to return to the correct path in a straight line. The main benefit of this design is that it is simple to implement, which makes it easy to turn into a standardized function. By the addition of checks and recursive exception routines, the design also becomes easy to adapt for functioning in complex or unexpected situations. The main drawback however, is that the vehicle may end up taking non-optimal paths around obstacles because the distance covered while avoiding is independent of the obstacles size

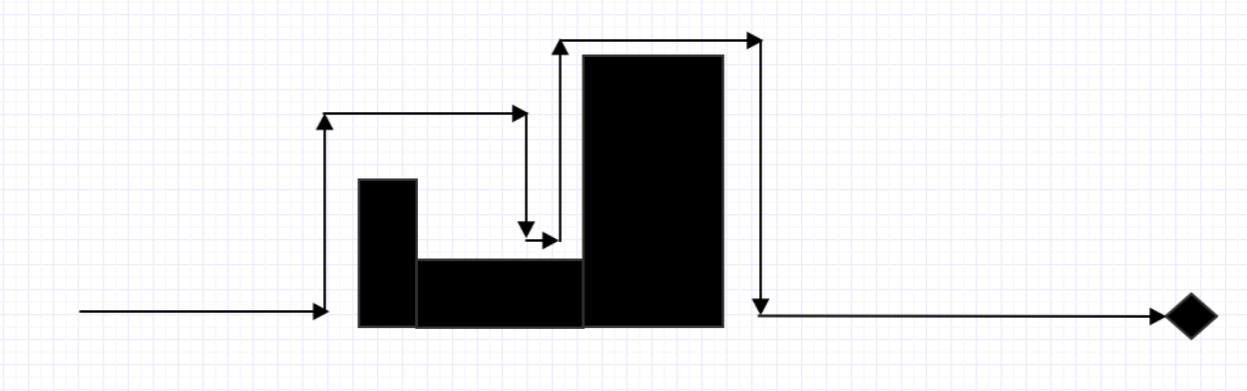


Figure 16: Avoid() Design 2

8.1.1.3 Design 3

The third and final design considered would use side sensors to stay within a certain distance of the obstacle. As shown in figure 17, the vehicle will turn once the side sensors inform it that the area is clear. The vehicle will then repeat this process until it reaches the correct path. This design will allow for easy position tracking and have the vehicle take an efficient path around most obstacles. Very irregular obstacles with many variations however, will significantly increase the time taken to navigate around them if this design is used as opposed to the other two.

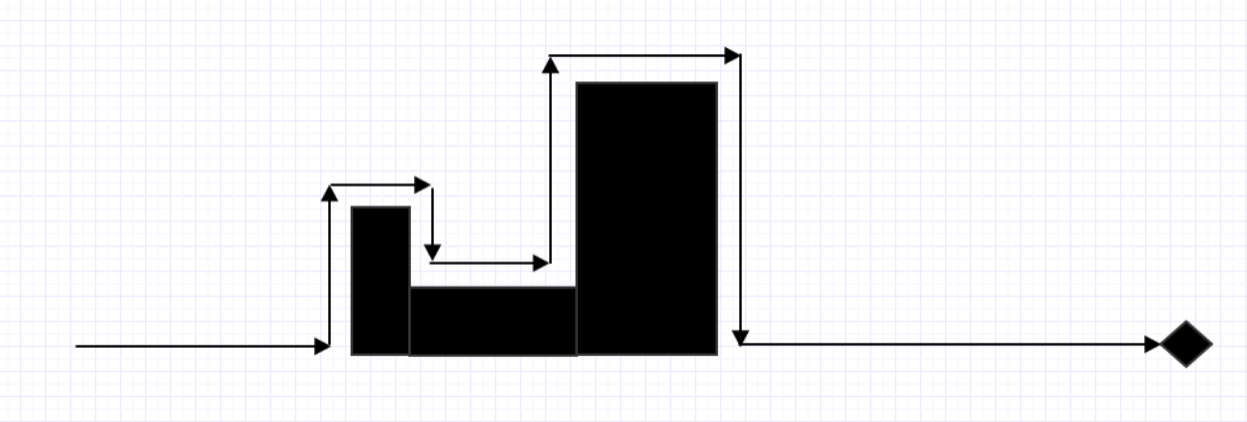


Figure 17: Avoid() Design 3

8.1.1.4 Design Selection

After investigating how each design will behave in a wide variety of situations to pick the best solution. The design would need to allow for efficient position tracking in standard time, which is something that would get complicated quickly in design 1. Furthermore, the fact that design 1 would need to move using right-angled turns if its y-axis path is blocked meant that either design 2 or 3 would need to be implanted to overcome such obstacles. To maintain efficiency, the choices were limited to designs 2 and 3

The decision between designs 2 and 3 were made based on how highly irregular obstacles would be handled by each of them. If the irregularities occurred at the end of the fixed distance traveled in design 2, both designs would take a similar amount of time to navigate around the obstacle. If the irregularities were more frequent however, design would be much quicker. This happens because design 2 only checks for the object it is avoiding after every forward() call unless the obstacle blocks its path. On the other hand, design 3 will try to stay close to the obstacle throughout, which will have it make significantly more turns and take more detours. For better visualization, the comparison between them in such conditions is shown in figure 18.

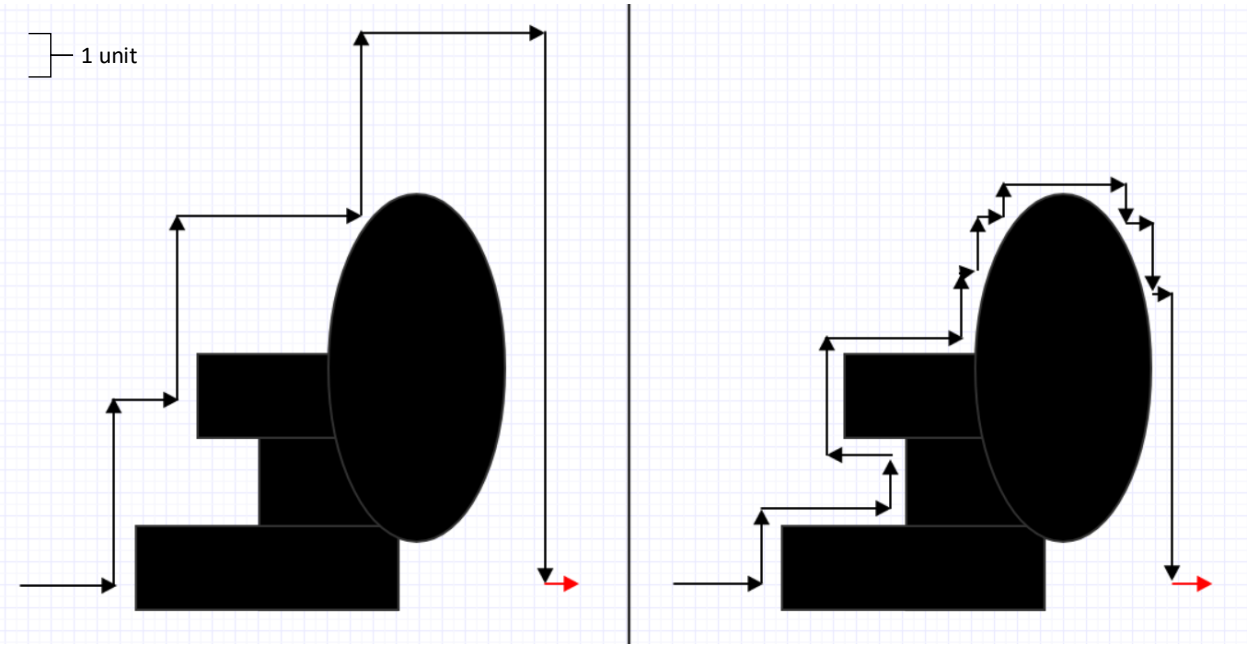


Figure 18: Irregular obstacle navigation by design 2 (left) and design 3 (right)

Using the example in figure 18, and by assuming that a second is needed to complete one turn and half a second is needed to cover each unit the time to, the time to navigate the obstacle can be calculated for comparison. The assumptions were based on consultation with the team member responsible for the vehicle. By taking the start point as encountering the obstacle and the endpoint as facing the red direction, we can see that:

- Design 2 takes:
    - 8 x 1 = 8 secs for all turns
    - 4 x 2 + 0.75 + 1.5 + 6 = 16.25 secs for movement approximately
    - 8 + 16.25 = 24.25 secs in total
- Design 3 takes:
    - 18 x 1 = 18 secs for all turns
    - 5x0.75 + 2x1.25 + 2x0.5 + 2x1.5 + 5x0.25 + 2.5 = 14 secs for movement approximately
    - 18 + 14 = 32 secs in total

Therefore, it can be surmised that as the obstacle becomes more irregular, design 3's efficiency dramatically decreases. As a result, design 2 was chosen for the development of the avoid() function.

8.1.1.4 Design Elaboration

The next step was elaborating on the selected design and developing its program flow. As with functions discussed previously, development started with the core functionality then was expanded to include exception flows and efficiency requirements. Additionally, the function was developed to be the only function that performs the avoiding routine, much like the main functions discussed in previous sections.

The first step of designing avoid() was implementing the core routine of the function. This routine was split into the four steps shown in figure 19. The steps are defined by the turns made by the vehicle at their beginning. Steps 1 and 4 are called the primary steps because they mark the beginning and end of the routine and both turns are clockwise by default. Steps 2 and 3 are called the secondary steps and both start with anticlockwise turns by default. Each step calls a function called reorient(), which calls rotate() then updates the orientation of the vehicle, then uses forward() to move a fixed distance 'a' through the AVOID caller ID discussed in section 7.

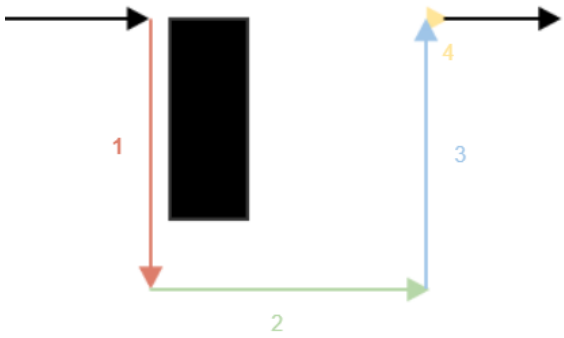


Figure 19: Basic avoid() Steps

After the core routine was successfully developed and tested, the function was edited to make it viable for using when avoiding multiple obstacles and to improve efficiency. The final flow chart can be seen in figure 20. In the flowchart the primary steps are shown in red, while the secondary steps are shown in blue. The details of the flowchart will be discussed further in the relevant subsections.

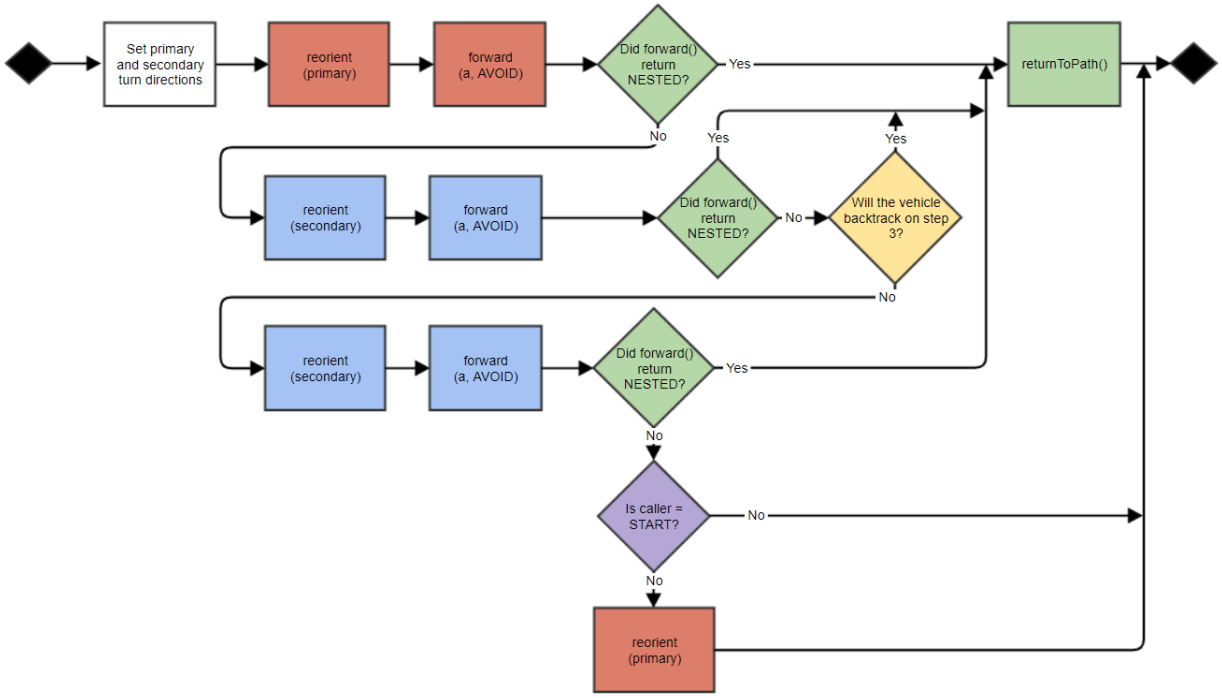


Figure 20: Avoid() Flowchart

8.1.2 Overall Design

Avoid() takes five integer variables as inputs. These are as follows:

- **RemainingX** - Measures the x-axis distance from the destination
- **RemainingY** - Measures the y-axis distance from the correct path
- **Orientation** - The direction the vehicle is currently facing
- **Caller** - An ID identifying how the function was called
- **DevPointX** - The x-axis point at which an obstacle is encountered on the correct path

Since these are the same five variables seen in forward(), they are also all pointers except caller. The pointers are used to keep track of the vehicle's status, hence they are primarily used to be passed to forward() and the path correction function. The caller variable is used because there is slight variance in avoid() behaviour based on whether it was a nested call or the first call. Its code is found in Appendix G.

8.1.3 Setting Turn Directions

As discussed in section 7.5, the vehicle may need to overshoot the correct path on rare occasions where it can't navigate around the obstacle on that side. On these occasions if the vehicle encounters further obstacles, the default settings for primary and secondary turns may cause it to overlap with the original path and run into the very first obstacle it encountered at devPointX. To overcome this issue, the primary and secondary turns need to be mirrored, i.e. be anticlockwise and clockwise respectively. This is done by checking the value of remainingY, if it is positive that means the vehicle is 'above' the correct path and the turns need to be mirrored. The default values are used otherwise.

8.1.4 Handling Multiple Obstacles

How avoid() handles multiple obstacles is shown with the green nodes in figure 20. As discussed previously in section 7.4, forward() returns the value of its caller variable. If it encounters an obstacle while being used to avoid another, it returns NESTED. Avoid() checks if NESTED is returned after every call to forward(). If the check is true this means that the vehicle navigated around an obstacle during that call of forward(). The remaining flow is then overridden, and the vehicle attempts to return to the correct path. By doing this the vehicle avoids continuing the algorithm and performing unnecessary movements.

8.1.4.1 Backtrack Check

After the second step and its NESTED check, the program checks whether the vehicle will end up backtracking on step 3. The check is based on the vehicle's current orientation and y-axis position. This is shown in figure 20 with the yellow node and is needed as an efficiency measure. Without this check, if the vehicle encounters an obstacle on its way back to the correct path, it may end up moving back on

step 3, then having to recover that x-axis distance when it is back to the correct path. With this check however, step 3 is overridden and the vehicle continues directly towards the correct path from that point onwards.

8.1.4.2 The Fourth Step

Before the fourth step is performed the program checks the caller ID. If the caller is not START, this step is not performed. If avoid() is used to navigate around additional obstacles while avoiding one and performs the fourth step, the vehicle will need to be reoriented to efficiently return to the correct path, which nullifies the fourth step. Therefore, it is more efficient to only use it through the first avoid() call.

8.2 Path Correction

ReturnToPath() is the function used for path correction and any situation where the vehicle needs to efficiently return to the correct path. The function is entirely case based and uses information on the vehicle to figure out how to do that. It attempts to bring the vehicle to the correct path in at most two steps unless an obstacle is encountered. From the forward movement level's perspective returnToPath() is indistinguishable from avoid(), which is why they are considered to be in the same program level. It can also be used recursively to avoid the duplication of its code in different cases.

8.2.1 Overall Design

ReturnToPath() takes four integer variables as inputs. These are as follows:

- **RemainingX** - Measures the x-axis distance from the destination
- **RemainingY** - Measures the y-axis distance from the correct path
- **Orientation** - The direction the vehicle is currently facing
- **DevPointX** - The x-axis point at which an obstacle is encountered on the correct path

Since these are the same four variables seen in forward() and avoid(), they are also all pointers. All the pointers are used to make logic decisions, as well as the arguments for forward() calls. Its code is found in Appendix G.

8.2.2 Case Handling

The logic followed by returnToPath() depends on the vehicle's current orientation. Each case is discussed below.

8.2.2.1 FORWARD

If the vehicle is facing forward, returnToPath() first checks how far away from the initial deviation point the vehicle currently is by finding the difference between devPointX and remainingX. If it is behind or at the deviation x-axis point, the vehicle will move forward until it is ahead of the point before proceeding. The function will then use the current value of remainingY to determine the direction of the vehicle's turns and how much it needs to move to reach the correct path and face the right direction.

8.2.2.2 RIGHT & LEFT

Whether facing RIGHT or LEFT, the behaviour of returnToPath() in both cases is more or less the same for both, just mirrored. Just like in the FORWARD case, the first step is checking whether the vehicle is ahead of the initial deviation point or not. If it is not, the vehicle is reoriented to face FORWARD then returnToPath() is called again before exiting the function. If the vehicle is already ahead of devPointX, returnToPath() uses the value of remainingY to choose how to get back to the correct path and face forward.

8.2.2.3 BACKWARD

Due to the way obstacle navigation algorithm is handled, the program should never call returnToPath() when facing BACKWARD. However, as a failsafe this case reorients the vehicle to face FORWARD then calls returnToPath() again.

8.2.3 Handling Obstacles

As mentioned above, the forward movement level makes no distinction between avoid() and returnToPath(). This is due to both functions requiring the vehicle to only move the distance they give forward() unless an obstacle is detected, hence the AVOID caller ID is sufficient for both of them. This means that if an obstacle is encountered during returnToPath()'s forward movement, it will be handled as if avoid() encountered another obstacle during its routine. As a result, no special instructions were needed to ensure obstacles are well handled. returnToPath() only needed to perform the NESTED check after every forward() call to be able to recursively call itself and correct the vehicle's path.

8.3 The End Result

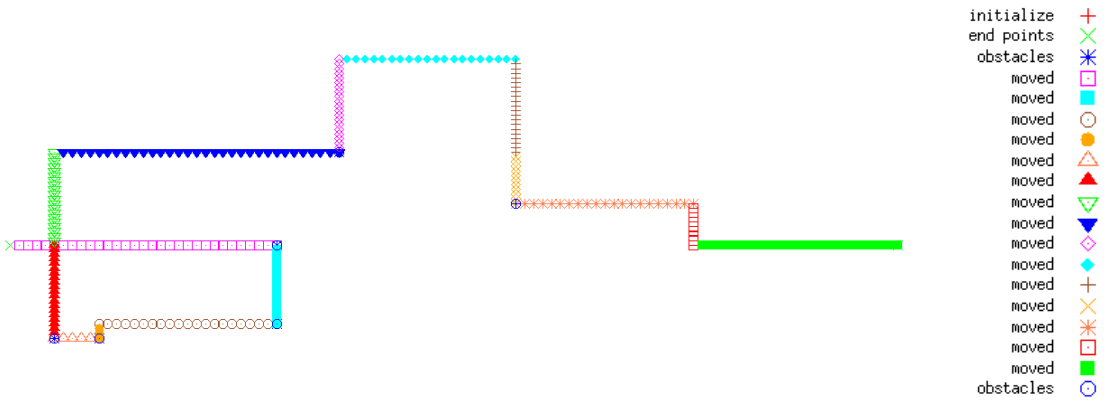


Figure 21: Obstacle Navigation Simulation and Its Key

By using avoid() and returnToPath() together, the program is capable of handling obstacle courses more complicated than what can be normally expected in a location used for golf. This flexibility can be shown in the simulation shown in figure 21. The simulation was performed by adapting the

program to be able to plot the its movement and obstacles as points on a graph. This was done by using a freely available, command-driven graphical display tool for Unix developed by N. Devillard. The positions where the vehicle would ‘detect’ obstacles were read from a text file and checked for using the vehicles current position. Additional obstacle navigation simulation results can be seem in Appendix H.

9 File Organization

The program was organized into a number of source files and headers that mean to improve the program's readability and allow for future reuse of functions. Those files are:

- definitions - Defines all the fixed values needed by the entire software
- main - Implements the shell program
- vehicleControl - Implements the vehicle control program and the overall movement level
- multiball - Implements the functions needed to handle multiple balls
- moves - Implements forward movement level and rotate()
- obstacles - Implements the obstacle navigation level

The files needed to launch the camera program and control the arm and motors of the vehicle were incorporated during the integration of the project parts.

10 Conclusion

10.1 Contributions

The contributions made by the author to the entire project are:

- Development of a software architecture that integrates the various aspects of the program
- A vehicle control program that independent from the shell program and is capable of handling all results from the object detection algorithm, including:
    - Multiple Balls
    - Uncertainty of results
- The behaviour algorithm that oversees the robot's movement
- An obstacle navigation algorithm capable of handling complex obstacle courses and efficiently correcting its path
- Several functions that can be reused, repurposed, and updated easily
- Libraries that define fixed values and organize functions based on their use

10.2 Future Work

Naturally, there is always room for improvement. Areas of interest for future work include:

- Complete integration of all the developed features of the software with the components developed by the other team members
- Expanding the retrieveBall() function to allow it to handle a wider variety of potential movement requirements
- The introduction of sensors to the sides of the vehicle that will make decisions on which direction to rotate more intelligent, therefore improving obstacle navigation
- Improvements and fine-tuning to all algorithms as they are noted during testing

10.3 Summary

By following an iterative top down approach to the development of the software architecture and behaviour of the robot, both sections of the software were successfully coded and tested. The result was a program that utilized modularity, and function reuse to efficiently meet the design requirements of the robot. Pending complete integration of the other parts of the project with the developed software, the robot is currently capable of handling a wide variety of situations on the field, regardless of whether they are due to the results of the object detection algorithm or a complicated set of obstacles in its path.

References

There was nothing to actually cite in the body of the report. However, the following websites and tools were used throughout the project and for this report:

This website was used extensively to learn about how C functions. The links and discussions shared within it provided a wealth of information:
https://stackoverflow.com/

Provided information on software development cycle and the relevant figure in this report:
https://online.husson.edu/software-development-cycle/

The virtual machine was provided by Carleton University through:
https://carleton.ca/scs/technical-support/virtual-machines/

The flowcharts were drawn using:
gliffy.com

The obstacle navigation performance was simulated using:
http://ndevilla.free.fr/gnuplot/

Appendices

Appendix A – Definitions

```
#ifndef definitions
#define definitions

#define DOC         "test.txt"                                  //The name of the file with the camera results

#define CTRL_PATH   "/home/student/Desktop/Project/vehicleControl"  //Path to vehicle control
#define VEH_CTRL    "vehicleControl"                            //Name of vehcile control executable

#define CAM_PATH    "home/student/Desktop/Project/camera"           //Path to camera
#define CAM         "camera"                                    //Name of camera executable

#define NEXTPOS 45                  //The degree rotation to next position

//The codes used for orientation
#define FORWARD 00                  //The code for facing the ball
#define BACKWARD 11                 //The code for facing away from the ball
#define RIGHT 01                    //The code for facing 90deg clockwise away from the ball
#define LEFT 10                     //The code for facing 90deg anticlockwise away from the ball

//The codes used by forward()
#define START 0                     //The caller code for calling forward through start
#define AVOID 1                     //The caller code for calling forward through avoid
#define NESTED 2                    //The caller code resulting from calling avoid while avoiding

//The codes that define the direction of rotations for avoid()
#define CLOCKWISE 90                //Used by avoid for 90deg right turns
#define ANTICLOCKWISE -90           //Used by avoid for 90deg left turns

#endif
```

Appendix B – Shell Program

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

#include "definitions.h"

int main()
{
    //The code to create a result text file during development
    /*
    FILE *file;
    file = fopen(DOC, "w+");
    fprintf(file, "%d %d %d", 100, 50, 0);
    fprintf(file, "%d %d %d", 40, 25, 0);
    fclose(file);
    */

    pid_t id; //The variable used to store process id

    //Clear four areas
    for (int i = 0; i < 4; i++)
    {
        //fork() to create the process used for the camera
        id = fork();

        if (id == 0)
        {
            printf("\nRun camera\n");
            execl(CAM_PATH, CAM, NULL); //Child runs camera code
        }
        else
        {//Parent waits for camera completion
            waitpid(id, 0, 0);

            printf("\nCamera done\n");
        }
```

```
34
35              printf("\nCamera done\n");
36          }
37
38          //fork() to create the process used for vehicle Control
39          id = fork();
40
41          if (id == 0)
42          {//Child runs vehicle control program
43              printf("\nRun vehicle control\n");
44              execl(CTRL_PATH, VEH_CTRL, NULL);
45          }
46          else
47          {//Parent waits for vehicle control completion
48              waitpid(id, 0, 0);
49
50              printf("\nArea %i cleared\n", i);
51          }
52      }
53      return 0;
54  }
```

Appendix C – Vehicle Control Program

```c
// The vehicle control everything
// Integration into the whole project yet undetermined
// Exact vehicle controls pending

#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
#include <stdbool.h>

//My personal headers
#include "definitions.h"
#include "multiball.h"
#include "moves.h"

//Functions
void retrieveBall(int distance, int angle, int check, int offsetDis, int offsetAng);
void grabBall();
void dropBall();

int main()
{
    int  ballCount, actualDis, i;
    FILE *file;

    //Counting lines
    file = fopen(DOC, "r");
    ballCount = countBalls(file);

    Ball balls[ballCount] ; //Create correctly sized array

    i = 0;      //Reset index

    file = fopen(DOC, "r");

```

```
34
35      while(!feof (file) && i < ballCount && !ferror(file))
36      {
37              fscanf(file, "%d %d %d",  &balls[i].distance, &balls[i].angle, &balls[i].check);
38              balls[i].offsetDis = 0;
39              balls[i].offsetAng = 0;
40              i++;
41      }
42
43      fclose(file);
44
45
46      for( i = 0; i < ballCount; i++)
47      {
48           if (balls[i].check == 1)
49          {
50              //The ball might be a lie
51              actualDis = balls[i].distance / 2;
52              balls[i].offsetDis = actualDis;
53              balls[i].offsetAng = balls[i].angle;
54
55              retrieveBall(actualDis, balls[i].angle, balls[i].check, balls[i].offsetDis, balls[i].offsetAng);
56
57              pid_t id = fork();  //fork to reaccess camera
58
59              if(id == 0)
60              {
61                  printf("\nI am the camera bro!\n");
62
63                  //Remove the test text before calling the camera
64                  //In case there is no ball, the test text will not be updated
65                  //By removing it we can check whether there is a ball or not by checking for the doc's existence
66                  remove(DOC);
67
68                  execl(CAM_PATH, CAM, NULL);
69              }
```

```
69              }
70              else
71              {
72                  waitpid(id, 0, 0);
73
74                  //If test exists, program reads it, then goes for the ball
75                  if(access("/home/student/Desktop/Project/test.txt", F_OK) != -1)
76                  {
77                      file = fopen(DOC, "r"); //Update this ball's information
78                      while(!feof (file) && !ferror(file))
79                      {
80                          fscanf(file, "%d %d %d",  &balls[i].distance, &balls[i].angle, &balls[i].check);
81                      }
82
83                      retrieveBall(balls[i].distance, balls[i].angle, balls[i].check, balls[i].offsetDis, balls[i].offsetAng);
84                  }
85                  else    //Else returns to base
86                  {
87                      retrieveBall(0, balls[i].angle, 0, balls[i].offsetDis, balls[i].offsetAng);
88                  }
89              }
90          }
91          else
92          {
93              //The ball is not a lie
94              retrieveBall(balls[i].distance, balls[i].angle, 0, balls[i].offsetDis, balls[i].offsetAng);
95              balls[i].offsetDis = 0;
96              balls[i].offsetAng = 0;
97          }
98      }
99
100     rotate(NEXTPOS); //Face next area's position
101     printf("Now in next camera position \n");
102
103     remove(DOC);    //remove test now that it is done
104     return 0;
105 }
```

Appendix D – Multiple Balls

```c
#ifndef multiball
#define multiball

//Defines the structures and functions needed to handle multiple balls

//The structure that define a ball attributes
typedef struct{

    int distance;           //distance = distance to detected ball in cm
    int angle;              //angle = signed angle to face detected ball
    int check;              //check =  whether a second object detection run is needed. True = needed. False = not needed
    int offsetDis;          //Distance from base to current location. 0 if starting at base
    int offsetAng;          //Angle between base and current location. 0 if starting at base

}  Ball;

//Function that counts balls by counting lines in the cameras output
int countBalls(FILE *file);


#endif
```

```c
#include <stdio.h>
#include <stdbool.h>

#include"definitions.h"
#include"multiball.h"

int countBalls(FILE *file)
{
    int c;
    int count = 1;

    while(!feof (file))
    {
        c = fgetc(file);
        if(c == '\n')
        {
            count++;
        }

    }

    fclose(file);

    return count;
}
```

Appendix E – Overall Movement

```
107  //Does Overall movement
108  void retrieveBall(int distance, int angle, int check, int offsetDis, int offsetAng)
109  {
110      if(distance != 0)   //Steps 1-3 are only needed when approaching a ball or what seems like it
111      {
112          //Step 1
113          rotate(angle);
114
115          //Step 2
116          start(distance);
117
118          //Step 3
119          if (check == 1)
120          {
121              printf("Back to Camera for check\n");
122              return;
123          }
124          else
125          {
126              grabBall();
127          }
128      }
129
130      //Step 4
131      rotate(180);
132
133      //Step 5
134      start(distance + offsetDis);
135
136      //Step 6
137      if(distance != 0)
138      {
139          dropBall();
140      }
141
142      //Step 7
143      rotate(180 - angle - offsetAng);
144      printf("\n");
145
146      return;
147  }
```

Appendix F – Forward Movement and Rotation

```h
#ifndef moves
#define moves

//Defines the basic movement control of the vehicle

//Rotates the vehicle
void rotate(int angle);

//Initializes movement
void start(int distance);

//Moves the vehicle forward
int forward(int distance, int *remainingX, int *remainingY, int *orientation, int caller, int *devPointX, int *testcounter);

//Updates the correct distance based on orientation
void updateDis(int moved, int *remainingX, int *remainingY, int *orientation);

#endif
```

```c
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>

#include"definitions.h"
#include"moves.h"
#include "obstacles.h"


 //Rotates the vehicle
void rotate(int angle)
{
    printf("Turn %i degrees \n", angle);
    return;
}

```

```
16
17  //Initializes all the values for movement control
18  void start(int distance)
19  {
20      int orient = FORWARD;
21      int *orientation;
22      orientation = &orient;
23
24      int moved = 0;
25      int toDoX, toDoY;
26      int *remainingX, *remainingY;
27
28      toDoX = distance - moved;
29      toDoY = 0;
30
31      remainingX =  &toDoX;
32      remainingY = &toDoY;
33
34      int devPoint = 0;       //Used by pointer
35      int *devPointX;         //It is the X axis point at which the very first deviation from the correct path occurs
36      devPointX = &devPoint;
37
38      //These values are used exclusively for testing
39      int count = 0;
40      int *testcounter;
41      testcounter = &count;
42
43      forward(0, remainingX, remainingY, orientation, START, devPointX, testcounter);
44
45      return;
46  }
47
```

```
47
48  //Moves the vehicle forward
49  //Will take care of moving in a straight line
50  //If an obstacle is encountered, it will call the avoid function
51  //The Caller IDs are:
52  //  START: is the initial call, and is aiming for the ball
53  //  AVOID: is the avoid function, and moves a fixed distance only
54  //  NESTED: is used to return that the ball encountered additional obstacles
55  //******THE testcounter IS ONLY FOR TESTING*********
56  int forward(int distance, int *remainingX, int *remainingY, int *orientation, int caller, int *devPointX, int *testcounter)
57  {
58      int *determinant;           //It is the value used to decide whether a called forward function accomplished it's task
59
60      if(caller == START)
61      {
62          determinant = remainingX;
63      }
64      else
65      {
66          determinant = &distance;
67          distance = abs(distance);
68      }
69
70      //The movement part
71      while(*determinant > 0 )
72      {
73          *testcounter = *testcounter + 1;
74
75          if(*testcounter == 3 || *testcounter == 5 || *testcounter == 8 || *testcounter == 190 )
76          {
77              printf("\nObstacle detected\n");
78
79              if(caller == START)
80              {
81                  *devPointX = *remainingX;   //Saves the X position of the first deviation. Used to find way back
82              }
83
```

```
83
84              avoid(remainingX, remainingY, orientation, caller, devPointX,testcounter);
85
86              if(caller == AVOID)
87              {
88                  caller = NESTED;
89                  break;
90              }
91          }
92          else
93          {
94              updateDis(10, remainingX, remainingY, orientation);
95              printf("Moved 10 cm. remainingX is %d, remainingY is %d\n", *remainingX, *remainingY);
96
97              if (caller != START)
98              {
99                  distance = distance - 10;
100             }
101
102             //logic when crossing y-axis
103             if((*orientation == LEFT || *orientation == RIGHT) && *remainingY == 0)
104             {
105                 if(disFromDevPoint(remainingX, devPointX) >= 0 && distance < 20)
106                 {
107                     forward(20, remainingX, remainingY, orientation, AVOID, devPointX, testcounter);
108                 }
109
110                 return NESTED;
111             }
112         }
113     }
114
115     return caller;
116 }
117
```

```
117
118     //Updates the correct remaining distance
119     void updateDis(int moved, int *remainingX, int *remainingY, int *orientation)
120     {
121         switch(*orientation)
122         {
123             case FORWARD :
124                 *remainingX = *remainingX - moved;
125                 break;
126             case BACKWARD :
127                 *remainingX = *remainingX + moved;
128                 break;
129             case RIGHT :
130                 *remainingY = *remainingY - moved;
131                 break;
132             case LEFT :
133                 *remainingY = *remainingY + moved;
134                 break;
135         }
136
137         return;
138     }
```

Appendix G – Obstacle Navigation

```
#ifndef obstacles
#define obstacles

//Takes care of avoiding obstacles and path correction

//The main function that corrects the path
void avoid(int *remainingX, int *remainingY, int *orientation, int caller, int *devPointX, int *testcounter);

//The function that rotates the vehicle and updates itscase orientation
void reorient(int angle, int *orientation);

//Function returns the vehicle to the correct path
void returntoPath(int *remainingX, int *remainingY, int *orientation, int *devPointX, int *testcounter);

//Returns X distance to devPointX
int disFromDevPoint(int *remainingX, int *devPointX);

#endif
```

```c
#include <stdio.h>
#include <stdbool.h>

#include "definitions.h"
#include "obstacles.h"
#include "moves.h"

//avoid obstacles by makin a square around it
//******THE testcounter IS ONLY FOR TESTING*********
void avoid(int *remainingX, int *remainingY, int *orientation, int caller, int *devPointX, int *testcounter)
{
    int multiAvoid;                 //Used to check if avoid was called recursively.
    int primary, secondary;         //Used to deteremine direction of turns based on mode

    if(*remainingY <= 0)
    {   //This mode is the default
        primary = CLOCKWISE;            //sequence of turns: Right, Left, Left, Right
        secondary = ANTICLOCKWISE;
    }
    else
    {   //This mode is entered only if the default mode was not possible
        primary = ANTICLOCKWISE;        //Sequence of turns: Left, Right, Right, Left
        secondary = CLOCKWISE;
    }

    //Step 1
    reorient(primary, orientation);
    multiAvoid = forward(20, remainingX, remainingY, orientation, AVOID, devPointX, testcounter);
    if(multiAvoid == NESTED)
    {
        returntoPath(remainingX, remainingY, orientation, devPointX, testcounter);
        return;
    }

```

```
34
35      //Step 2
36      reorient(secondary, orientation);
37      multiAvoid = forward(20, remainingX, remainingY, orientation, AVOID, devPointX, testcounter);
38      if(multiAvoid == NESTED || (*orientation == RIGHT && *remainingY > 0) || (*orientation == LEFT && *remainingY < 0))
39      {
40          returntoPath(remainingX, remainingY, orientation, devPointX, testcounter);
41          return;
42      }
43
44      //Step 3
45      reorient(secondary, orientation);
46      multiAvoid = forward(20, remainingX, remainingY, orientation, AVOID, devPointX, testcounter);
47      if(multiAvoid == NESTED)
48      {
49          returntoPath(remainingX, remainingY, orientation, devPointX, testcounter);
50          return;
51      }
52
53      //Step 4 only done if not avoiding multiple obstacles, i.e. avoid is called by first forward
54      if(caller == START)
55      {
56          reorient(primary, orientation);
57      }
58
59      return;
60  }
61
```

```
61
62  void reorient(int angle, int *orientation)
63  {
64      rotate(angle);
65
66      if(angle == 180)
67      {
68          switch(*orientation)
69          {
70              case FORWARD :
71                  *orientation = BACKWARD;
72                  break;
73              case BACKWARD :
74                  *orientation = FORWARD;
75                  break;
76              case RIGHT :
77                  *orientation = LEFT;
78                  break;
79              case LEFT :
80                  *orientation = RIGHT;
81                  break;
82          }
83      }
84      else if(angle > 0)  //Checks if rotation is clockwise
85      {
86          switch(*orientation)
87          {
88              case FORWARD :
89                  *orientation = RIGHT;
90                  break;
91              case BACKWARD :
92                  *orientation = LEFT;
93                  break;
94              case RIGHT :
95                  *orientation = BACKWARD;
96                  break;
```

```
                *orientation = BACKWARD;
                break;
            case LEFT :
                *orientation = FORWARD;
                break;
        }
    }
    else
    {
        switch(*orientation)
        {
            case FORWARD :
                *orientation = LEFT;
                break;
            case BACKWARD :
                *orientation = RIGHT;
                break;
            case RIGHT :
                *orientation = FORWARD;
                break;
            case LEFT :
                *orientation = BACKWARD;
                break;
        }
    }

    return;
}

```

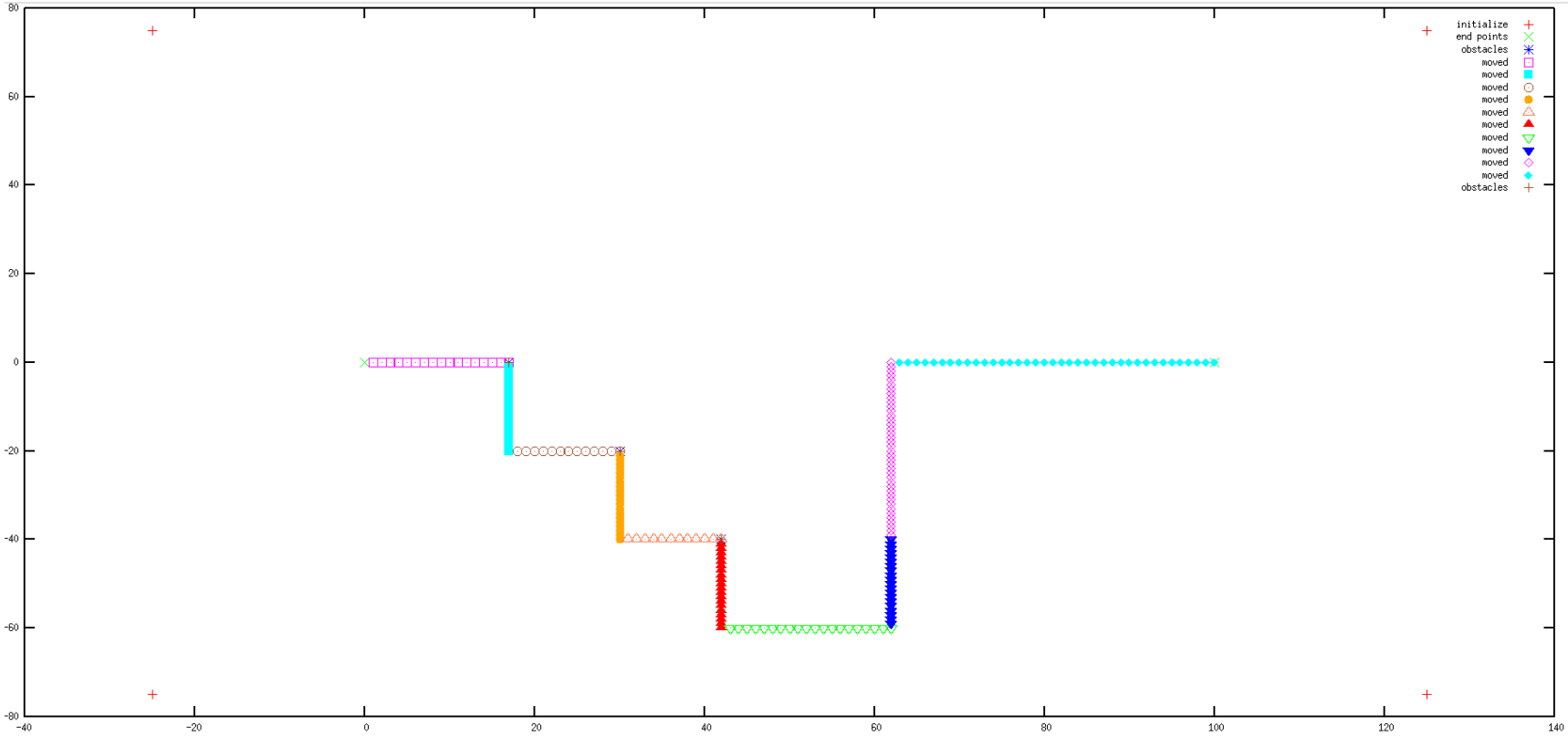
```
123
124 void returntoPath(int *remainingX, int *remainingY, int *orientation, int *devPointX, int *testcounter)
125 {
126     int multiAvoid;
127
128     switch(*orientation)
129     {
130         case FORWARD :
131             if(disFromDevPoint(remainingX, devPointX) >= 0)
132             {
133                 multiAvoid = forward((disFromDevPoint(remainingX, devPointX) + 10), remainingX, remainingY, orientation, AVOID, devPointX);
134                 if(multiAvoid == NESTED)
135                 {
136                     returntoPath(remainingX, remainingY, orientation, devPointX);
137                     return;
138                 }
139             }
140
141             if(*remainingY > 0)
142             {
143                 reorient(90, orientation);
144                 multiAvoid = forward(*remainingY, remainingX, remainingY, orientation, AVOID, devPointX, testcounter);
145                 if(multiAvoid == NESTED)
146                 {
147                     returntoPath(remainingX, remainingY, orientation, devPointX, testcounter);
148                     return;
149                 }
150
151                 reorient(-90, orientation);
152             }
153             else if(*remainingY < 0)
154             {
155                 reorient(-90, orientation);
156                 multiAvoid = forward(*remainingY, remainingX, remainingY, orientation, AVOID, devPointX, testcounter);
157                 if(multiAvoid == NESTED)
158                 {
```

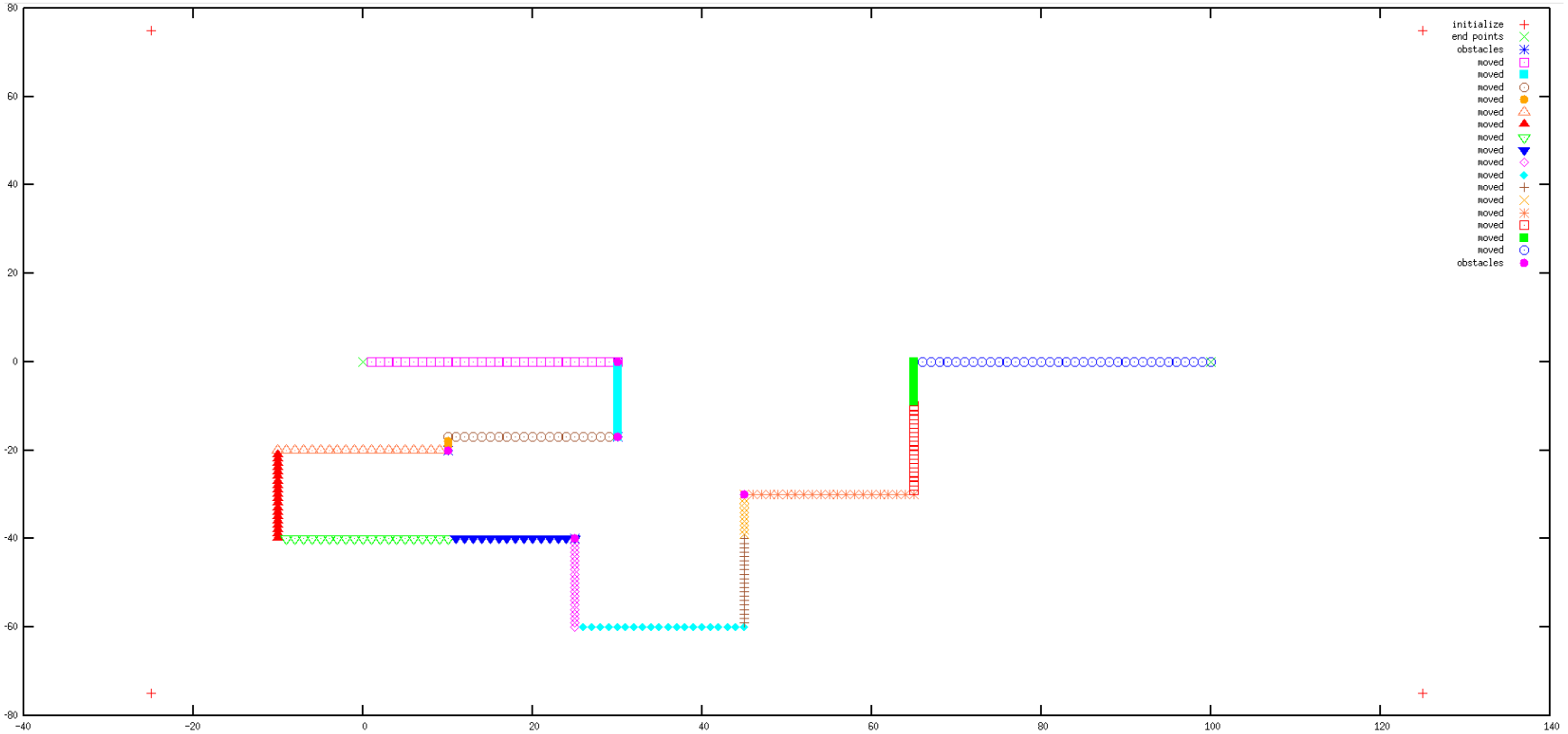
```
157             if(multiAvoid == NESTED)
158             {
159                 returntoPath(remainingX, remainingY, orientation, devPointX, testcounter);
160                 return;
161             }
162 
163             reorient(90, orientation);
164         }
165 
166         break;
167 
168     case BACKWARD :
169         reorient(180, orientation);
170         returntoPath(remainingX, remainingY, orientation, devPointX);
171         break;
172 
173     case RIGHT :
174         if (disFromDevPoint(remainingX, devPointX) > 0)
175         {   //Face forward then recall returntoPath
176             reorient(-90, orientation);
177             returntoPath(remainingX, remainingY, orientation, devPointX, testcounter);
178             return;
179         }
180 
181         if(*remainingY > 0)
182         {
183             multiAvoid = forward(*remainingY, remainingX, remainingY, orientation, AVOID, devPointX, testcounter);
184             if(multiAvoid == NESTED)
185             {
186                 returntoPath(remainingX, remainingY, orientation, devPointX, testcounter);
187                 return;
188             }
189 
190             reorient(-90, orientation);
191         }
```

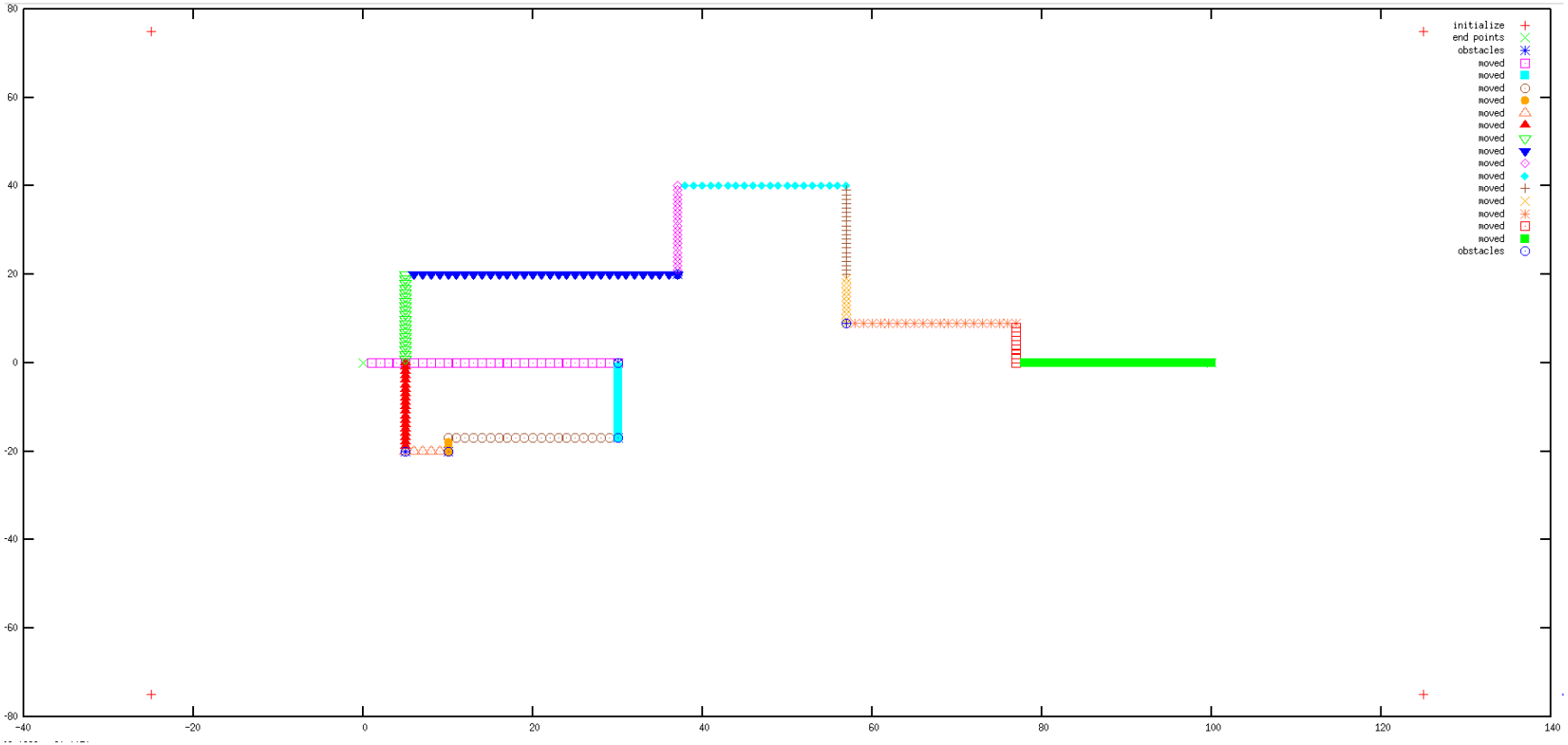
```
191             }
192             else if(*remainingY < 0)
193             {
194                 reorient(180, orientation);
195                 multiAvoid = forward(*remainingY, remainingX, remainingY, orientation, AVOID, devPointX, testcounter);
196                 if(multiAvoid == NESTED)
197                 {
198                     returntoPath(remainingX, remainingY, orientation, devPointX, testcounter);
199                     return;
200                 }
201 
202                 reorient(90, orientation);
203             }
204             else
205             {
206                 reorient(-90,orientation);
207             }
208 
209             break;
210 
211         case LEFT :
212             if (disFromDevPoint(remainingX, devPointX) > 0)
213             {   //Face forward then recall returntoPath
214                 reorient(90, orientation);
215                 returntoPath(remainingX, remainingY, orientation, devPointX, testcounter);
216                 return;
217             }
218 
219             if(*remainingY > 0)
220             {
221                 reorient(180, orientation);
222                 multiAvoid = forward(*remainingY, remainingX, remainingY, orientation, AVOID, devPointX, testcounter);
223                 if(multiAvoid == NESTED)
224                 {
225                     returntoPath(remainingX, remainingY, orientation, devPointX, testcounter);
226                     return;
227                 }
```

```
220             {
221                 reorient(180, orientation);
222                 multiAvoid = forward(*remainingY, remainingX, remainingY, orientation, AVOID, devPointX, testcounter);
223                 if(multiAvoid == NESTED)
224                 {
225                     returntoPath(remainingX, remainingY, orientation, devPointX, testcounter);
226                     return;
227                 }
228 
229                 reorient(-90, orientation);
230             }
231             else if(*remainingY < 0)
232             {
233                 multiAvoid = forward(*remainingY, remainingX, remainingY, orientation, AVOID, devPointX, testcounter);
234                 if(multiAvoid == NESTED)
235                 {
236                     returntoPath(remainingX, remainingY, orientation, devPointX, testcounter);
237                     return;
238                 }
239 
240                 reorient(90, orientation);
241             }
242             else
243             {
244                 reorient(90,orientation);
245             }
246 
247             break;
248     }
249 
250 }
251 
252 int disFromDevPoint(int *remainingX, int *devPointX)
253 {
254     return (*remainingX - *devPointX);
255 }
256 
```

Appendix H – Obstacle Navigation Demonstrations




initialize
end points
obstacles
moved
moved
moved
moved
moved
moved
moved
moved
moved
moved
moved
moved
moved
moved
moved
moved
moved
obstacles
80
60
40
20
0
-20
-40
-60
-80
-40
-20
0
20
40
60
80
100
120
140


initialize
end points
obstacles
moved
moved
moved
moved
moved
moved
moved
moved
moved
moved
moved
moved
moved
moved
moved
obstacles
80
60
40
20
0
-20
-40
-60
-80
-40
-20
0
20
40
60
80
100
120
140