# Cover Page

CS1701

Group Project Lectures & Tutorials

CS1803 – Group Project Reflection

CS1809 – Software Design

CS1810 – Software Design

Task 2 – Traffic Lights

Group B33

Tutor: Prof. Theodora Koulouri

Nathan Singh – 2376781

Page Count: 8/10 (Report)

13 pages (Source code)

# Implementation Summary

## Functional Requirements:

1. The program must display a command prompt window which the user can interact with.
2. To allow for the Swift Bot to navigate its path using colour code input using the Swift Bot's onboard camera.
3. The Swift Bot must be able to respond to traffic lights when the lights are within a range of 20cm.
4. The program must be able to process the photo taken of the traffic light and determine the colour.
5. Once pressed; the ‘A’ button must initiate the program.
6. The Swift Bot should be able to toggle its LED under lights ON and OFF as well as switch between different colours.
7. The Swift Bot should have a predefined ‘low initial speed’ at which it travels at certain intervals.
8. The Swift Bot must be able to come to a halt at certain given points.
9. The Swift Bot must be able to rotate ‘x’ number of degrees in a fixed point to change direction.
10. The Swift Bot should be able to retrace its movement.
11. The Swift Bot must be able to come to a full stop.
12. The program must be stopped when the user presses the “X” button on the Swift Bot.
13. The program must be able to create, open, read to, write to and save a text file; where a display of the execution log will be present.
14. The program must be able to keep count of the types of colours and number of times it visits traffic lights as well as be able to calculate the most frequently visited of the same.
15. The Swift Bot and program code must keep track of time to plot and coordinate certain events during the program's running time.

## Non-Functional Requirements (in accordance with functional requirements above)

2) In order to navigate its path using colour code input, the Swift Bot must take a picture of its environment via the onboard camera and follow set protocol based on the input it detects.

4) One of the methods by which the picture taken (#2) by the Swift Bot can be processed is via a pixel matrix. After this is computed each pixel colour can be independently verified; after which the colour of the traffic light can be concluded and the following sequences can be executed.

7) A predefined low initial speed can be computed using simple mathematics and trial and error to determine a suitable speed. The challenge lies in optimizing the robot’s speed to achieve faster and more efficient travel while ensuring it can correctly identify traffic lights within 20cm of them.

14) In order to keep an efficient count of most frequently visited colours during the program runtime, each time a traffic light is passed a value must be added to a tally that can be accurately displayed at the end of the program.

## Additional Requirements/Functionalities:

* In addition to the basic red-light functionality listed in the study guide, the robot will also simulate a real world-like scenario when it encounters a red light. If a vehicle/ object is present in front of it the robot will wait for 3 seconds for it to move, if it does not move the robot will change lane and continue its path.
* When the program starts, the A button must be pressed to execute it. To signal the user to press the A button, it flashes for a brief amount of time to draw the user's attention to the button that needs to be pressed. This is important as the buttons on the robot are physically small and maybe harder for certain program users (especially those with weaker eyesight, etc.) to use.

## Omitted Requirements (in accordance with functional requirements above)

10) The program does not need a special algorithm/method by which it retraces its movement. This is for a couple reasons; namely it is only required in the singular instance when a blue colour traffic light is encountered. Hence, creating an algorithm that retraces the Swift Bot’s last steps is a waste of time and computational power. It makes more sense for the inverse of the movement to be executed.

# Changes to the Algorithm and UI Design

* In Assignment 2, I describe 2 methods by which the robot can decide what colour is in front of it. The first method includes having a preset value for each colour and matching it accordingly with the input from the camera. This method has been omitted as it is prone to errors; colour values will not always match up due to variance in lighting conditions, etc.
* In the final rendition of my code, I have utilized a mix of the second method and a separate method not described in Assignment 2. Once an image is taken and stored in buffer. The RGB values of each pixel are extracted, and then accumulated into their respective variables; avgR, avgB, avgG. After this an average is displayed to the console, indicating which value is largest. The program flow continues into its allocated methods from here on.
* Implemented as an additional requirement, the A button light will flash when the program is first executed. As mentioned above, it is a means by which we can grab user attention and allow for people will slight visible impairments to also interact with the software.
* The UI has also significantly changed, in the design assignment there were not ample messages displayed in the terminal to make the user aware of the current state of the program and the operations it is undergoing at given points in time.

# Testing

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Requirement # | Input | Desired Outcome | Actual Outcome | Pass/Fail |
| 3 | N/A - start robot from distance greater than 20cm (Valid Input) | To stop within 20cm of the traffic light | Stops within 20cm of the traffic light | Pass |
| 3 | N/A - start robot from a distance less than 20cm (Invalid input) | To stop within 20cm of the traffic light | Stops as soon as traffic light is detected but reports distance incorrectly (robot is already too close to traffic light) | Fail |
| 4 | Any 3 of the valid traffic colour lights: Red, Green or Blue | Compute and identify what colour is being shown and branch into the appropriate method | Identifies colour accurately and branches into respective method | Pass |
| 4 | Colours other than Red, Green and Blue that are invalid and not recognized by the program | N/A - The colour may be valid but under poor lighting conditions, dirt on the robot camera etc. | Identifies and goes to class that resembles either Red, Green or Blue the closest. | Fail |
| 5 | Button A - valid | The robot will start moving | Robot moves | Pass |
| 5 | Button B, X, Y – invalid inputs (at the initial stage of program) | The robot will start moving | A message is printed to the terminal and the robot does not move | Fail |
| 8 | N/A - in this case the inputs are events that are pointers for the robot to carry out different actions | Robot should stop | The robot does stop for X number of seconds and continues accordingly | Pass |
| 10 | Invalid inputs in this scenario are not entered but rather the physical surface traversed upon. Example: Carpet | Robot should carry out the required movement in the blue method and retrace its movement accurately | The code for retracing is executed but due to the surface being optimal, the robot cannot move as intended | Fail |
| 12 | Button X | Program terminated and prompts user to either display log or write it to a text file | Program stops and presents 2 options to the user either show log or write to text file | Pass |
| 14 | Different traffic light colours – valid inputs | Tallys increment by 1 every time respected traffic colour is visited | Tallys are updated accordingly and will be displayed accurately | Pass |
| 14 | Invalid colour inputs | As mentioned before this event may be due to poor lighting conditions or simply if another colour is shown. It is still desirable to for a colour to be identified accurately | The program detects and decides a colour based on what is most like to its RGB color definitions and may alter the accuracy of the tally. | Fail |
| 15 | Button A | Timer to initialize and start accordingly | The timer starts when the program is executed | Pass |
| 15 | Buttons X and Y – invalid inputs | Use timer | The timer is only initialized when A is pressed hence it does not initialize and cannot compute a total run time | Fail |
| Additional Functionality #1 | User input Button A – valid input | Light starts flashing and becomes active to receive input | Button flashes for 5 seconds and prints confirmation to terminal if pressed | Pass |
| Additional Functionality #1 | User input any button except A – invalid input | Program doesn't run (as per study guide) | Button presses print to terminal, but program doesn't execute as button A has not been pressed. | Pass |
| Additional Functionality #2 | Vehicle in front of robot at red traffic light | Robot does not move until path is clear | The under lights flash yellow and red to indicate that the red process as per the study guide has been executed and is now awaiting the path ahead to be clear to proceed. | Pass |
| Additional Functionality #2 | Vehicle ahead of traffic light moves but comes to a stop again | Robot can stop in time | Robot has jumped back to lowInitSpeed(); method and will now detect the vehicle ahead as a traffic light and act accordingly | Fail |

# Planning and Monitoring progress

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ***Task*** | ***Milestones*** | ***Deadline*** | ***Completed date*** | ***Deliverable*** |
| Initial Plan | Basic flowchart | 15/12/23 | 19/12/23 | Basic flowchart and rough idea of variable names |
| Rough pseudocode | Couple of valid methods | 23/12/23 | 29/12/23 | Rough methods for each class |
| Naming variables | Suitable names chosen | 21/12/23 | 28/12/23 | Appropriately named variables which are easily implementable |
| Robot systems testing (in accordance with task) | Valid code that executes | 14/12/23 | 4/1/24 | Linked variables to different parts of robots to be used in different parts of program |
| Start Report | Complete individual sections | 3/1/24 | 27/2/24 | A mock copy of the report that isn’t fully complete |

Challenges faced:

1. Another method to compute colour had to be decided on due to inconsistencies with lighting, camera issues etc. To overcome this hurdle, I tried different methods by which colour could be determined. One of them was to locate the central pixel and use that as a fair means to determine the colour of the entire traffic light. Albeit plausible, it was like the original method where it only worked in favorable lighting conditions, etc. After trial and error, the method I saw fit was to collect all the RGB values and create averages for each RGB colour and compare them to the values of the other RGB values in the image. The largest RGB presence was decided upon as the final colour.
2. Another challenge I faced was error handling in an open environment. Given that the robot has no predefined idea of what a traffic light or colour is, it is susceptible to recognizing different objects in the real world as its target detected object. To negate this, I decreased the stopping distance, so it allows for the robot to come closer to objects. In the grand scheme of things this proved to be more time effective as well as efficient in recognizing colours.

# Source Code

import swiftbot.Button;

import swiftbot.ImageSize;

import swiftbot.SwiftBotAPI;

import java.awt.image.BufferedImage;

import java.io.BufferedWriter;

import java.io.File;

import java.io.FileWriter;

import java.io.IOException;

import java.time.Duration;

import javax.imageio.ImageIO;

public class as3 {

private static SwiftBotAPI *API* = new SwiftBotAPI();//declared as a private so it can be used w put constant redefining

//api also doesnt permit repeated calling of api, produces error

private static int[] *colourToLightUpR* = {255,0,0};

private static int[] *colourToLightUpB* = {0,255,0};

//colour format is RBG

private static int[] *colourToLightUpG* = {0,0,255};

private static int[] *colourToLightUpY* = {255,0,255};

private static int *tallyB* = 0;

private static int *tallyG* = 0;

private static int *tallyR* = 0;

private static int *lightEncounter* = 0;

private static long *startTime1* = System.*nanoTime*();

private static long *endTime1* = System.*nanoTime*();

private static boolean *isButtAPressed*; //to create desired methods

private static boolean *isButtBPressed*;

private static boolean *isButtXPressed*;

private static boolean *isButtYPressed*;

public static void main(String[] args) throws IOException{

int tallyR = 0;

int tallyG = 0;

int tallyB = 0;

*API*.enableButton(*Button*.***A***, ()->{

System.***out***.println("Button A has been pressed"); //print statement to include UI user

*isButtAPressed* = true;

});

//initialize other buttons

//B isnt intialized because it is not neeeded

*API*.enableButton(*Button*.***X***, ()->{

System.***out***.println("Button X has been pressed");

*isButtXPressed* = true;

try {

*pressedX*(*isButtAPressed*, *isButtAPressed*, *endTime1*);

} catch (IOException e) {

e.printStackTrace();

}

*API*.stopMove();

System.***out***.println("Do you want to display an execution log");

System.***out***.println("Press Y for YES AND X for no");

});

*API*.enableButton(*Button*.***Y***, ()->{

System.***out***.println("Button Y has been pressed");

*isButtYPressed* = true;

*displayExecLog*();

});

System.***out***.println("Press A to start program");

for (int i = 0; i < 5; i++) {

*API*.setButtonLight(*Button*.***A***, true); //flash buttA for 5sec

//signals which butt to press

try { //additional functionality

Thread.*sleep*(500);

} catch (InterruptedException e) {

e.printStackTrace();

}

*API*.setButtonLight(*Button*.***A***, false);

try {

Thread.*sleep*(500);

} catch(InterruptedException e) {

e.printStackTrace();

}

}

while(true) {

if(*isButtAPressed* == true) {

long startTime = System.*nanoTime*();//nano time must be divided by one million to convert to seconds

System.***out***.println("PROCEEDING");

*API*.fillUnderlights(*colourToLightUpY*);

System.***out***.println("Looking for traffic light!");

break;

}

try {

Thread.*sleep*(250);

} catch (InterruptedException e) {System.***out***.println(e);}

}

if(*isButtAPressed* == true) {

*lowInitSpeed*();

}

}

static String *chosenCol* = "";//string for colour decided

private static long *startTime*;

public static void lowInitSpeed() throws IOException{

*API*.fillUnderlights(*colourToLightUpY*);

*API*.startMove(70,70);// lower speed to ensure ultrasound accuracy

boolean continueMove = true;

double disToTrafLight = 0;

while(continueMove) {

disToTrafLight = *API*.useUltrasound(); //when object within 15cm stop and get ready to decide image colour

if (disToTrafLight < 20) {

continueMove = false;

*API*.stopMove();

System.***out***.println("Current distance to traffic light is :" + disToTrafLight + "cm. Now deciding colour...");

*decideCol*(continueMove, 0, 0, 0);

break;

}

}

}

public static void decideCol(boolean continueMove, int tallyG, int tallyB, int tallyR) throws IOException { //how to error handle if other obj other than traf light is deteced

try {

BufferedImage img = *API*.takeStill(*ImageSize*.***SQUARE\_48x48***); //small resolution for faster computation

System.***out***.println("Taking image");

ImageIO.*write*(img, "jpg", new File("/home/pi/Documents/testImage.jpg")); //where image is stored as buffer

} catch (Exception e) { // removed traditional exception handling commands and only included 'Exception e' to scan all

e.printStackTrace();

System.***out***.println("PICTURE TAKEN, PROCESSING NOW");

}

//try diff image sizes in testing to find best time and space complexity

String fn = ("/home/pi/Documents/testImage.jpg");

BufferedImage img = ImageIO.*read*(new File(fn));

int avgR = 0; //initializing colour variables

int avgG = 0; //^will change when updated from camera

int avgB = 0;

for(int x = 0;x<img.getWidth();++x)

{

for(int y=0;y<img.getHeight();++y)

{

int p = img.getRGB(y,x);

int r = (p >> 16) & 0xFF;

int g = (p >> 8) & 0xFF;

int b = p & 0xFF;

avgR += r;

avgG += g;

avgB += b;

}

}

avgR /= (img.getWidth()\*img.getHeight());

avgG /= (img.getWidth()\*img.getHeight());

avgB /= (img.getWidth()\*img.getHeight());

System.***out***.println(avgR + "," + avgG + "," + avgB);

if (avgR > avgG && avgR > avgB){

System.***out***.println("The traffic light is red");

String chosenCol = "Red";

*detectRed*(chosenCol);

}

if (avgG > avgR && avgG > avgB){

System.***out***.println("The traffic light is green");

String chosenCol = "Green";

*detectGreen*(chosenCol);

}

if (avgB > avgR && avgB > avgG){

System.***out***.println("The traffic light is blue");

String chosenCol = "Blue";

*detectBlue*(chosenCol);

}

}

//GREEN

public static void detectGreen(String chosenCol) throws IOException {

if (chosenCol.equals("Green")) { //uses equals() instead of == to compare current variable and not its reference

*API*.fillUnderlights(*colourToLightUpG*);

*tallyG* ++;

*lightEncounter* ++;

System.***out***.println("NOW IN GREEN STATE");

System.***out***.println("Passing light wihtin the next 2 seconds...");

try {

Thread.*sleep*(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

*API*.startMove(100,100);

try { //move past traffic light in 2 seconds

Thread.*sleep*(2000);

} catch (InterruptedException e) {

e.printStackTrace();

}

*API*.stopMove();

}

try {

Thread.*sleep*(500); //last step of green lights

} catch (InterruptedException e) {

e.printStackTrace();

}

*API*.fillUnderlights(*colourToLightUpY*);

//go back to low init speed

*lowInitSpeed*();

}

//BLUE

public static void detectBlue(String chosenCol) throws IOException {

if (chosenCol.equals("Blue")) {

*API*.fillUnderlights(*colourToLightUpB*);

*tallyB* ++;

*lightEncounter* ++;

System.***out***.println("NOW IN BLUE STATE");

*API*.stopMove();

try {

Thread.*sleep*(500); //study guide says to stop for 1/2 second and then blink

} catch (InterruptedException e) {

e.printStackTrace();

}

*API*.disableUnderlights();

try {

Thread.*sleep*(400);

} catch (InterruptedException e) {

e.printStackTrace();

}

*API*.fillUnderlights(*colourToLightUpB*);

try {

Thread.*sleep*(400);

} catch (InterruptedException e) {

e.printStackTrace();

}

*API*.disableUnderlights();

try {

Thread.*sleep*(400);

} catch (InterruptedException e1) {

e1.printStackTrace();

}

*API*.fillUnderlights(*colourToLightUpB*); //unconventional light flashing method

try {

Thread.*sleep*(400);

} catch (InterruptedException e11) {

e11.printStackTrace();

}

*API*.disableUnderlights();

try {

Thread.*sleep*(400);

} catch (InterruptedException e111) { //e1,e11,e111 are different exceptions being caught

e111.printStackTrace();

}

*API*.fillUnderlights(*colourToLightUpB*);

*API*.stopMove(); //get ready to turn 90 degrees to the L

try {

*API*.startMove(0, 50);

Thread.*sleep*(1035);

*API*.stopMove();

} catch (InterruptedException e) {

}

//move forward for a second

try {

*API*.startMove(75, 75);

Thread.*sleep*(1000);

*API*.stopMove();

} catch (InterruptedException e) {

}

//retrace 1 - move back

try {

*API*.startMove(-75, -75);

Thread.*sleep*(1000);

*API*.stopMove();

} catch (InterruptedException e) {

}

//retrace rotate back to angle

try {

*API*.startMove(0, -50);

Thread.*sleep*(1035);

*API*.stopMove();

} catch (InterruptedException e) {

}

}

*lowInitSpeed*();

}

//RED

public static void detectRed(String chosenCol) throws IOException { //additional functionality to stop until traffic ahead is clear

if (chosenCol.equals("Red")) {

*API*.fillUnderlights(*colourToLightUpR*);

*tallyR* ++;

*lightEncounter* ++;

System.***out***.println("NOW IN RED STATE");

*API*.stopMove();

try {

Thread.*sleep*(500);

} catch (InterruptedException e) {

e.printStackTrace();

}

boolean continueMoveR = true;

while(continueMoveR) {

double disToVeh = *API*.useUltrasound(); //when object within 15cm stop and get ready to decide image colour

if (disToVeh < 10) {

continueMoveR = false;

*API*.stopMove();

System.***out***.println("Path not clear...waiting");

*decideCol*(continueMoveR, *tallyR*, *tallyR*, *tallyR*);

break;

}

if(disToVeh >= 10) {

continueMoveR = true;

try {

Thread.*sleep*(1000); //after vehicle not detected move forward for a second and continue

} catch (InterruptedException e) {

e.printStackTrace();

}

}

*lowInitSpeed*();

}

}

}

//if x is pressed

public static void pressedX(boolean isButtYPressed, boolean isButtXPressed, long endTime1) throws IOException {

*API*.disableButton(*Button*.***X***);

*API*.stopMove();

System.***out***.println("Press Y to see execution log or X to write to text file");

*API*.enableButton(*Button*.***X***, ()-> {

System.***out***.println("You have chosen to write log to text file");

});

if(isButtYPressed == true) {

System.***out***.println("Printing execution log...");

*displayExecLog*();

}

else if(isButtXPressed == true) {

*logInTxtF*(*chosenCol*, *lightEncounter*, *lightEncounter*, *lightEncounter*, *chosenCol*);

System.***out***.println("Printed to text file");

}

}

public static void displayExecLog () { //press Y

*API*.stopMove();

long endTime = System.*nanoTime*();//divided by a billion to calculate run time in seconds

long executionTimeMs = (endTime - *startTime*) / 1000000; //longs can not hold numbers greater than x value hence convert to million and then billion

double executionTimeSec = (executionTimeMs) / 1000000000.0;

int totalVisit = *tallyR* + *tallyB* + *tallyG*;

String mostVisit = "";

int encounter = 0;

if (*tallyR* > *tallyG* && *tallyR* > *tallyB*){

mostVisit = "Red";

encounter = *tallyR*;

}

else if (*tallyG* > *tallyR* && *tallyG* > *tallyB*){

mostVisit = "Green";

encounter = *tallyG*;

}

else if (*tallyB* > *tallyG* && *tallyB* > *tallyR*){

mostVisit = "Blue";

encounter = *tallyB*;

}

else if (*tallyR* == *tallyB* && *tallyR* > *tallyG*) {

System.***out***.println("Tally Red and Tally Blue are equal and greater than Tally Green");

}

else if (*tallyR* == *tallyG* && *tallyR* > *tallyB*) {

System.***out***.println("Tally Red and Tally Green are equal and greater than Tally Blue");

}

else if (*tallyB* == *tallyG* && *tallyB* > *tallyR*) {

System.***out***.println("Tally Blue and Tally Green are equal and greater than Tally Red");

}

System.***out***.println("The number of times the SwiftBot encountered traffic lights was: " + *lightEncounter*);

System.***out***.println("The most frequent traffic light colour encountered was: " + mostVisit);

System.***out***.println("The number of times " + mostVisit + " traffic lights were encountered was: " + encounter);

System.***out***.println("The total duration of program execution is: " + executionTimeSec + " seconds");

try {

Thread.*sleep*(2000);

} catch (InterruptedException e) {

e.printStackTrace(); //2 second pause before closing program

}

System.*exit*(0);

}

public static void logInTxtF (String mostVisit, int executionTimeSec, int lightEncounter, int lightEncounter2, String encounter) { //press x

try {

BufferedWriter writer = new BufferedWriter(new FileWriter("output.txt"));

writer.write("YOOOOOO The number of times the SwiftBot encountered traffic lights was: " + lightEncounter + "\n" +

"The most frequent traffic light colour encountered was: " + mostVisit + "\n" +

"The number of times " + mostVisit + " traffic lights were encountered was: " + encounter + "\n" +

"The total duration of program execution is: " + executionTimeSec + " seconds\n");

writer.close();

} catch (IOException e) {

e.printStackTrace();

}

System.***out***.println("Written to text file");

}

}