07COC251

A449326

Computer Controlled

Car Racing

By

Andrew Armstrong

Supervisor: Dr. C. Hinde

May 2008

Abstract

The automated navigation and control of a remotely controlled car by a computer is a difficult problem to solve. The task to have a system drive a remotely-controlled car for a certain amount of laps of a track without assistance was put forward during the 2005 Congress on Evolutionary Computation (CEC), called the "Racing Cars" competition. The winning entry by Ivan Tanev still had to be assisted to get around the track successfully, requiring human assistance to even get around the track once.

The previous two years of projects done at Loughborough University by four different student tried to solve this problem. Research and investigation by each has proven the task to be difficult by the varied implementation problems found and solutions proposed by each student. A lot of progress was made however, with both implementations of simulations and image processing techniques.

This project builds on the existing projects to include sufficient rule-based analysis of the track done by image processing, to successfully navigate around and avoid, or get out of, crash situations.

Overall, the final solution proposed is a solid implementation of a reliable, but slow and simple navigation framework which should require no assistance to get around the track. In addition it provides a well structured internal model for the car and track. The system is also well structured, allowing it to be expanded into further work if deemed necessary.

Acknowledgements

These people helped tremendously directly or indirectly for various reasons, and all need to be acknowledged.

Chris Hinde – For his valued supervision of the project, expert advice and suggestions.

Chris Barnard – For ideas on the initial project work, and for sharing lab time.

Christopher Carter – For the previous years work carried out in 2007, and the use of his code for the project base.

Scott Culcheth – For the proceeding project work from 2006 and research, and for the use of some portions of his code.

Richard Mee – For the construction of the parallel port interface for the initial project.

Thank you all.

Table of Contents

Introduction

The project concept originated in 2005 from the Congress on Evolutionary Computation (CEC) *(Essex, 2005i)*, during the "Racing Cars" competition. Dr. Chris Hinde brought the project to Loughborough University in 2005, and four students have worked on it previously.

In the 2005/2006 academic year, two MComp Computer Science students undertook the project, one of which was Scott Culcheth *(Culcheth, 2006)*. Their goal was to build a system capable of driving a remote control car around a track, and in addition to extend it to include autonomous AI to navigate the track, possibly including learning to improve the performance over many laps.

However, this proved difficult as controlling the car by itself proved a challenge and no AI was added to their projects. Therefore, in 2006/2007, Christopher Carter *(2007)* took the project further by looking to implement a system capable of driving a remote control car around a track without human assistance.

This proved a success, since Carter managed to implement a simulation and a list of commands to send to the car which could get the car around the track without human assistance for one lap. With this in place it was proved automated control or AI could be added to navigate the track.

Therefore, the aim of this project is to continue the aim of controlling the car without human assistance, with the additional aim to recover from crash situations which both Culcheth and Carter detailed as an important area of future work.

Aim: *To develop and implement a system which is capable of manoeuvring a remote control car around a track while doing automated crash avoidance and recovery without human assistance.*

Project CD-ROM

Included with this text is a CD-ROM with resources related to the project. Included on the disk are:

- A copy of this document in OpenDocument (.odt), Microsoft Word (.doc), and Portable Document Format (.pdf) formats
- The program source code files and instructions for use
- The UserPort program for allowing programs access to parallel port pins
- Videos of the final car performance, and for comparison the "Ivan Assisted" CEC 2005 winner video *(Essex, 2005ii)*

Literature and Resources Review

This project continues the work of two years of previous projects. Two of the four projects are the main source of information, since they provided the base system code and between them done a great amount of investigation into the project already. Additionally, software that was used in this project, or used in the system by previous projects is reviewed.

Analysis of Previous Projects

The two projects which work was continued from were Culcheth *(2006)* and Carter *(2007)* both called "Computer Controlled Car Racing".

Scott Culcheth (2006)

The initial work in 2006 provided the base of the project and an excellent overview of the problems involved. Part of Culcheth's investigation went towards creating the physical track, deciding on the camera settings use and also the car configuration. The track was painted matte black, and the camera's height and lighting conditions were tested to gain the best results for the system. The camera itself was setup to see all the track in the best possible way. The car's choice of tyres, and motor configuration was thoroughly tested, and the results from both were kept through Carter's project.

The project additionally made a lot of progress with image analysis and recognition, analysing the usefulness of several techniques. He details the use of image thresholding, background subtraction and frame differencing, noting the uses of each to different situations. Attention was given to the run time of the techniques noting while the accuracy could be improved by additional image filters, the time used to run them would be highly impractical since the system is meant to run in real time.

The control of the car, with a unfinished example of how to have the car navigate the track, is also provided. Pulse Width Modulation (PWM) is fully explained to give precise control of the actions sent to the car. The use of PWM was improved by having left-right steering independent from the forwards-reverse pulse, to improve the turning of the car due to gliding after the forwards-reverse pulse was sent.

Finally, the report details the system Culcheth created by the use of class diagrams, making the report extremely useful when adapting parts of the system for further use. An explanation for the use of Java is also given, with respect to the interaction with the web camera and the use of object orientated code.

Christopher Carter (2007)

The project completed by Carter in 2007 made a divergence from the use of the web camera to analyse the track. Instead, effort was made to achieve a full lap of the track with no assistance by implementing a simulator. The simulator would control the car by using an input list of instructions, but work with it's own measurements to test the car for crashes.

Additional testing was done to some areas which Culcheth had lacked detail on, such as the connection of the camera to the PC and the exact sizes of the track. The implementation itself provides detail on the exact turning and distance measurements recorded and used in the simulator.

Although this project did not continue the work of the simulator, Carter also improved the system code inherited from Culcheth, and provides a good set of class diagrams and information on the system layout. This proved valuable in adapting the project for further work.

CEC Car Racing Competition

The competition was run in 2005 by the University of Essex (Essex, 2005ii). The aim of the competition was to develop the most effective system to remotely control a car around a track for 10 laps. The 2005 winner was Ivan Tanev (Essex, 2005i), who provided a system which completed 10 laps but only with a high amount of human assistance, although it was more sophisticated and able compared to other entries.

Software Review

UserPort Driver

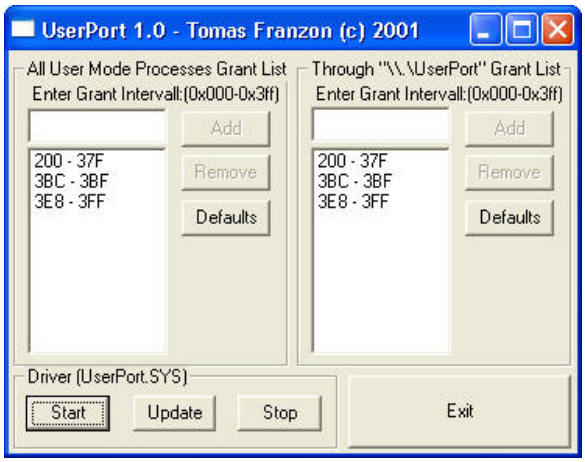


Figure 1 - UserPort 1.0 interface

The UserPort driver, available from the CEC competition website *(University of Essex, 2005i)*, unlocks parallel port addresses for use with programs on the system. As previous projects have investigated, it is recommended to unlock only the ports required due to the security risks *(Culcheth, 2006)*.

ParPort Java Package

Previous projects *(Carter, 2007)* have gained access to the parallel port by use of the ParPort Java Package, which contains the required DLL file. This was attributed originally to Juan Portillo, available on his website *(Portillo, 2005)*

Parallel Port Viewer

The Parallel Port Viewer was integrated into Carter's project *(Carter, 2007)*, and is a application to communicate to the parallel port, making use of both ParPort and UserPort. The original viewer is available on the Planet Source Code website *(Margold, 2004*), which allows direct manipulation of the parallel port pins states.

Specification

System Requirements

To complete the project aim, the car is required complete laps of a rectangular track built to the original CEC "Racing Cars" competition standard. The car and track can be viewed by the webcam setup on a gantry directly over the track. The car is controlled using the cars own remote control system, connected via parallel port to the PC running the system. The track and gantry was optimised for the system by previous projects *(Culcheth, 2006 and Oyns, 2006)*. The original hardware specifications can be found on the CEC website *(University of Essex, 2005i).* The controller was adapted by Richard Mee *(Loughborough University, 2005)*.

The projects completed in 2006 concluded with the analysis of several problems that still needed resolving to complete the core aim of track navigation. While several problems were solved, such as car and track detection, the results still couldn't match the CEC "Racing Cars" competition system winner provided by Ivan Tanev *(Doshisha University, Japan)*. A video of the winning laps is available on the CEC competition website *(University of Esex, 2005ii)*. Tanev's entry required a high amount of human intervention to succeed at navigating the track, despite having a better turning circle then this project's own car. Two large problems identified were the navigation out of crashes, and the reliability of finding the car's location and facing.

In 2007 the project *(Carter, 2007)* identified the area of simulation as an area which required work. He succeeded at building a reliable simulator and a set of instructions that could be followed by the car to do a successful lap of the track without human intervention. It was still identified that the car could not complete more then a few laps without human intervention due to the known accuracy limits of the simulation. It was still accepted that the car racing had more work to be done, since the project did not use the camera at all and used only a pre-set list of instructions.

Physical Components of the System

Track Layout

The track is the same as the design used in the 2005 CEC competition, and was constructed by Dr Chris Hinde *(Hinde, 2005)*. The design is detailed on the CEC competition website *(University of Essex, 2005)*.

The track originally used the brown finish it was completed with, but the initial 2006 project work *(Culcheth, 2006 and Oyns 2006)* discovered that using a greater contrast of colour between the track and walls would improve the detection of the car and track. The track was therefore painted matte black (which additionally avoids reflection) with white barriers. The track as used during the project is shown in figure 2.

The track layout consists of a rectangle track area, with a centre barrier dividing it in half. A chicane comes down from the centre "top" wall, and the car starts just between the chicane barrier and the centre barrier. Direction of travel around the track is clockwise.

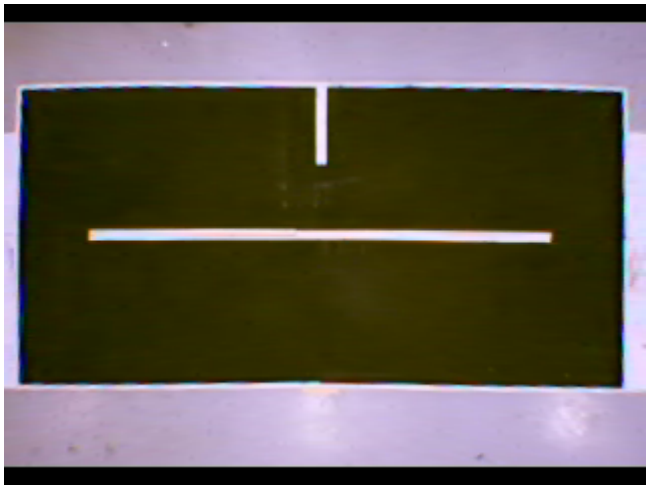


Figure 2 - Webcam photo of the track from a top-down perspective on the gantry.

The Car

The car is a standard *Nikko* brand 1/24 scale Mini Cooper Sport. The main colour is yellow and white, which contrasts well with the black track. There are optional motors and tyres, which were investigated by Culcheth *(2006)*, and were not altered for this project. The car takes 3 standard AA batteries.

The car in previous projects used a right steering bias *(Culcheth, 2006 and Carter 2007)* due to the clockwise direction of travel. This was set to neutral for the project since reverse and left travel were used frequently, and right bias would have added additional calculations to determine which direction the car would be heading.

The car is controlled using the native remove control provided, which is connected to the parallel port of the PC running the system. This takes a single 9V battery for operation. This *Nikko* range of cars can also uses the frequencies 40MHz and 27Mhz, which would allow a second car and controller to be added. Although this was not investigated during the project it is a future task to investigate.

Fig. 3: The Car

Fig. 4: The controller connected to the parallel port.

The Camera

The camera used is a standard wide-angle webcam, connected by USB to the PC running the system. The wide-angle functionality, when set to letterbox mode as seen in figure 2, allows the entire track to be in focus if at a reasonable height on the gantry. Investigation into other camera settings were performed in previous years *(Culcheth, 2006)*, with letterbox mode being the preferred choice.

Development Software

Java SDK and Java Media Framework (JMF)

The project has in previous years used Java 1.6.0 and the Java Media

Framework 2.1.1. During the first project, Culcheth *(2006)* details this was because of the usefulness of the JMF to interact with the web camera. The use of Java also allowed generally quick implementation of classes and features such as the required threading of the camera and running command processes, and is familiar. In addition there is a large amount of available examples and code samples for the language available, one of which was used for Culcheth's original project base *(Bull, 2004).* The use of Java has been continued to make use of the previous projects work, since this project makes use of two projects previous systems code *(Culcheth, 2006 and Carter, 2007)*.

Notepad++

The use of a design studio was not necessary and evaluated in previous projects *(Culcheth, 2006)*, where the use of an development IDE was not suitable due to the small amount classes used in the system. Notepad++ *(HO, 2008)* was chosen since it allowed tabbed file browsing, syntax highlighting, shortcuts for compilation and was very familiar.

Approach Taken

The 2006 projects *(Culcheth, 2006 and Oyns, 2006)* laid out several targets for the project, the main one being a similar aim to that of the CEC competition, to have the system perform unassisted car laps. The 2007 project investigated (Carter, 2007) approached this problem by creating a simulator instead of continuing image processing and web camera work.

After a talk with Dr Chris Hinde and Chris Barnard in preliminary meetings, it was decided to take forward the original design of using the web camera to provide a system that could navigate the track. Specifically, I took on board the problem of crash analysis and recovery, which I incorporated into my aim.

While the simulation approach *(Carter, 2007)* proved successful at accepting a list of actions and executing them effectively, it had difficulty going around the track more then twice. The main approach therefore was to accept the simulation was useful but to not expand on its functionality, instead making use of the image analysis techniques in the 2006 projects *(Culcheth, 2006).*

The approach of live image analysis lends itself to being perfect for either simple rules to govern car movement or the use of artificial intelligence

techniques to allow the system to learn. Out of these two options, the rules based approach was chosen, to both simplify the coding and to make it reliable to test. This was the approach chosen by Culcheth *(2006)*, from his initial work into the area of tracking and navigation.

While general rules could be difficult to apply to a complex track, the system is only required to navigate a relatively simple course. Additional rules to govern the crash situations and general problems with moving around obstacles would allow more complex rules or artificial intelligence algorithms to be added in future work on this project.

To allow the system to perceive the track, the use of an internal model to store the locations of the wall and obstacles would be required, as well as to store the cars location and facing. A similar storage of an internal route planning system would be needed, to allow the car to find its way accurately around the track. Without these, the overhead from checking the camera for all the information every update would stop the program working effectively *(Culcheth, 2006)*.

The system was created in Java, allowing the use of previous projects as a base for the GUI and camera systems. Changing the language would have been counter-productive and lose the existing systems implementations and code for reuse.

Known System Deviants

Both previous years have identified several inconsistent variables *(Culcheth, 2006 and Cater, 2007)*. These need to be taken into account by the system to maintain the reliability, so reasonable allowances are made for:

- Changes in lighting conditions
- Battery life
- Tyre grip and track surface
- System latency between requesting an image and processing it
- Processing overhead from other processes running on the PC, and the general speed of the PC itself

Out of these issues, the latency and speed of the PC were significant in the previous project *(Carter, 2007)*, which produce a lack of synchronisation between

the system knowledge and the real-world events. This was taken into account by the use of a delay in the navigation loop to allow the general overhead to be less of a problem, and to allow the car to stop moving before picking up its position. The rules were also altered to reflect these choices.

Lighting conditions affect the web camera heavily if the lights are dim, which are easily solved by having the lights on when the system runs, and have it taken into account by the image processing. Battery life can affect the distance of each pulse, although affected the system less due to the use of rules to determine behaviour. However if the car ran down the batteries, the movements might be too small and so navigation would prove impossible. Therefore, batteries were replaced frequently so to not hamper the cars movement.

Design

Design Methodology

For the project, inspiration on how to implement the required features was taken from the past projects. After analysis of the previous years, it was obvious iterative design and testing was the most appropriate for the system. This allowed the design to be planned and prototyped quickly, by keeping the parts of the design generally separate or only relying on the previously constructed parts. Testing was done as the project progressed, with earlier systems being tested for compatibility with newer ones and integration of features happening when they were added. Different options were evaluated before implementation, as detailed in the Evaluation of Coding Methods section. The iterative testing and design changes are detailed in the Testing section.

Project Code Base

As identified in the aim of the project, there will be no further work made on simulating the track since a great amount was completed by Carter *(2007)*. However, the program improvements to the class design and GUI made by Carter made it worthwhile to further expand his project and keep his existing simulation code in unaltered.

Since Carter had removed several important parts of Internal Model and Image Processing, it is necessary to re-implement them. The work done by Culcheth *(2006)* includes a lot on modelling and image processing, and much of this is reused where possible.

Implementation Order

The use of iterative design requires separating tasks into stages of incremental development. Earlier stages are required for later stages to be implemented, so the order of these are important. The implementation order used in the project is outlined in the table shown in figure 5.

No.	Stage	Sub-Stages	New code / modified from
1	GUI redesign	i. Plan GUI changes to make to existing GUI ii. Resize simulation, split GUI in half, resize existing panels to fit iii. Implement new buttons and placeholder panels	Modified from Carter *(2007)* project work.
2	Engine implementation	-	Partially modified code from Carter *(2007).*
3	Simulation Separation	i. Remove all GUI control functionality from the simulation classes, move to Engine ii. Rename simulation classes to better segregate code	Modified most of the work done by Carter *(2007)*, to separate the functionality from new code.
4	Static camera addition	-	Uses a modified copy of existing live camera code *(Carter, 2007)*.
5	Image Processing	i. Implement image processing classes and controls from Culcheth's work ii. Evaluate the most appropriate way to finding the track iii. Evaluate the most appropriate way to find the car when it is on the track iv. Evaluate the most appropriate way to find the car's facing	Partially implemented from code in Culcheth *(2006)* project. New code for the car facing needs to be evaluated.
6	Internal Model	i. Setup model structures and functions for the track ii. Add image algorithms to search for the track barriers and obstacles iii. Setup car structure iv. Add image algorithms to search for the car, and find it's facing v. Add algorithm to determine if a location is in a barrier or wall	New code, although inspiration taken from Culcheth's *(2006)* internal model.
8	Driving AI	i. Implement new class to handle the driving loop to control actions sent to the car ii. Integrate car control via. parallel port	Parallel port code from existing project *(Carter, 2007)*
9	Navigation AI	i. Implement control class to handle navigation rules ii. Integrate internal model updates into class as needed information	New code
7	Track Route Planning System	i. Investigate suitable solutions to finding routes around the track ii. Implement solution, adding in interfaces with the navigation AI	New code

11	Navigation Rules	Add rules to determined best new action to take considering information available i. Require the rules to move towards next waypoint ii. Take into account crash situations	New code
12	Driving PWM code	i. Add full PWM in a thread loop to take into account navigation rule decision ii. Test and tune PWM timings	PWM design from existing projects *(Culcheth 2006, Carter 2007)*
13	Final testing of crash recovery and laps	i. Final code and system testing ii. Testing and evaluation of performance with laps and crash recovery	New code

Figure 5 – Implementation Order Table

Class Interactions

The different sub-systems were designs in their own classes to implement the required functionality. Since the program is based upon past simulation code by Carter *(2007)*, this was put into it's own simulation section early on in the project, and all new program functionality run under a different class hierarchy. This also simplified the class diagram. The different systems are outlined in figure 5 below.

As noted in figure 6, the system developed by Carter is not shown in full, with only the entry class being referenced. The entire simulation system was separated from the rest of the program and left unaltered. Since the simulation is not being developed during this project, it would have taken up unnecessary space to explain its full functionality. Details of its implementation can be found in the project report *(Carter, 2007)*.

The new hierarchy allows a clear set of communications between the navigation and modelling classes, with much shared functionality coming from the vision system.

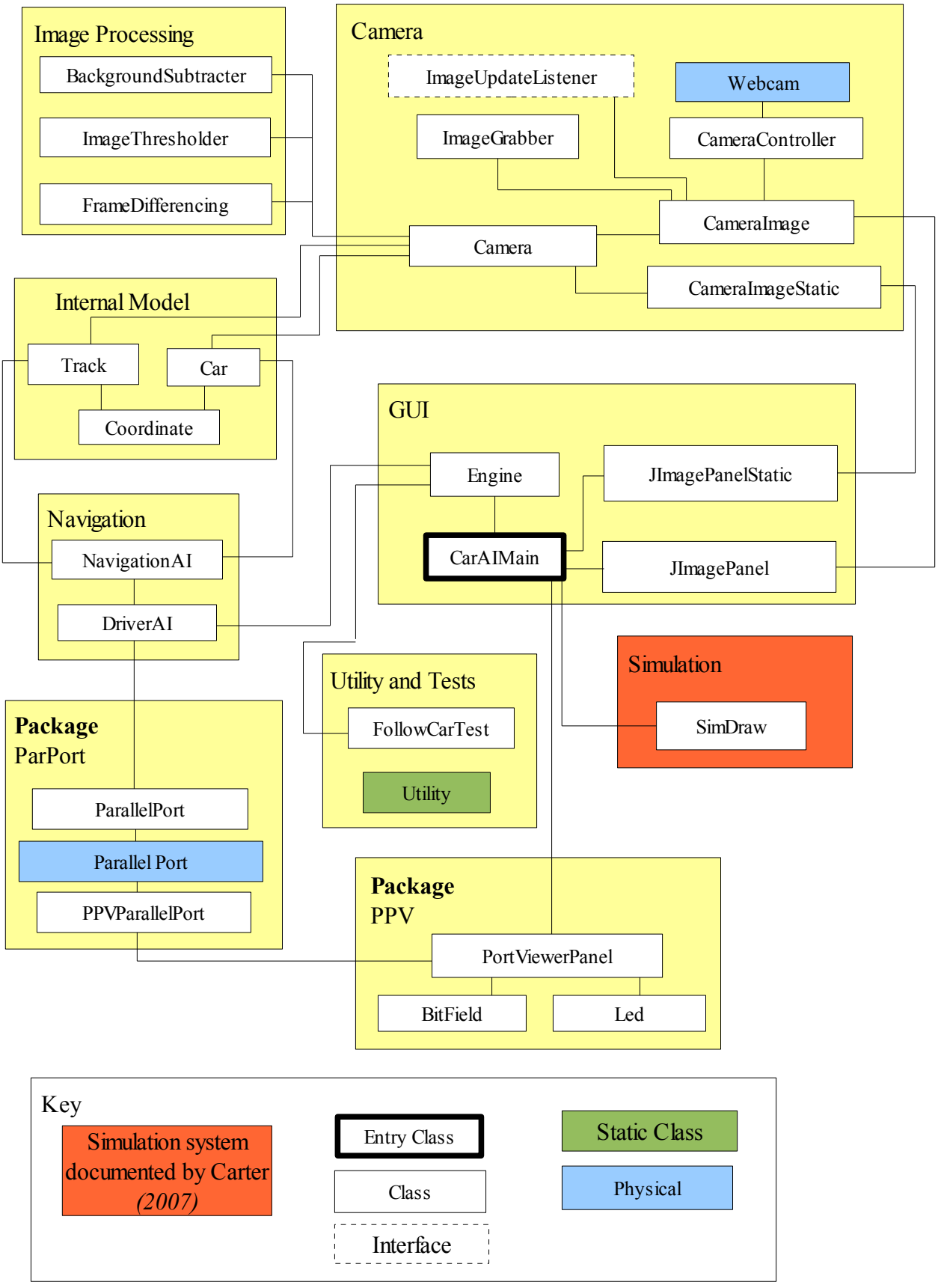


Figure 6 – Class Interaction Diagram.

User Interface Design

The user interface was originally inherited from Carter's *(Carter, 2007)* project shown in figure 7, and so was designed around the simulation of the track with no image processing needed. This was modified to add the necessary functionality for image processing and navigation AI.

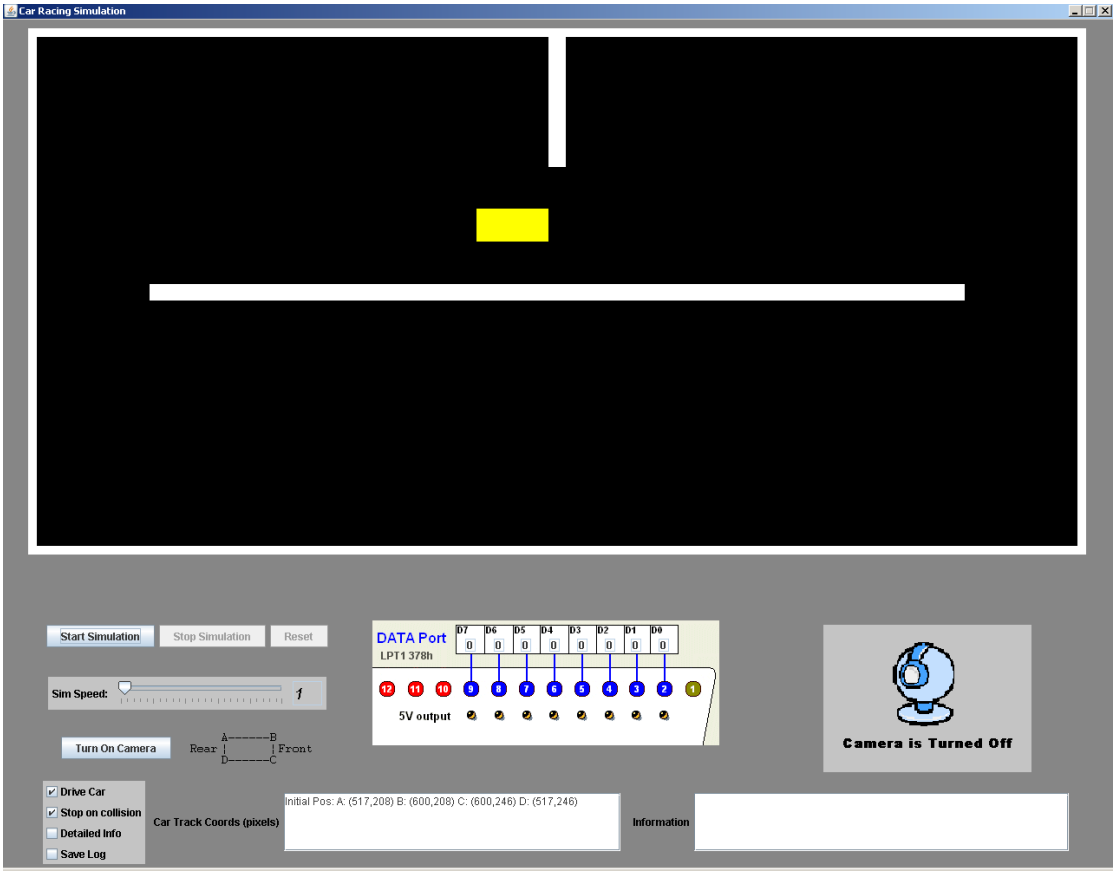


Figure 7 – Initial GUI state from Carter's *(2007)* project

The plan to change the interface was based around adding additional controls, and therefore necessitating the resizing of the simulation. It was decided to partition the GUI into two halves, and further divide it up into specific container panels. The initial plan for this is shown in figure 8.

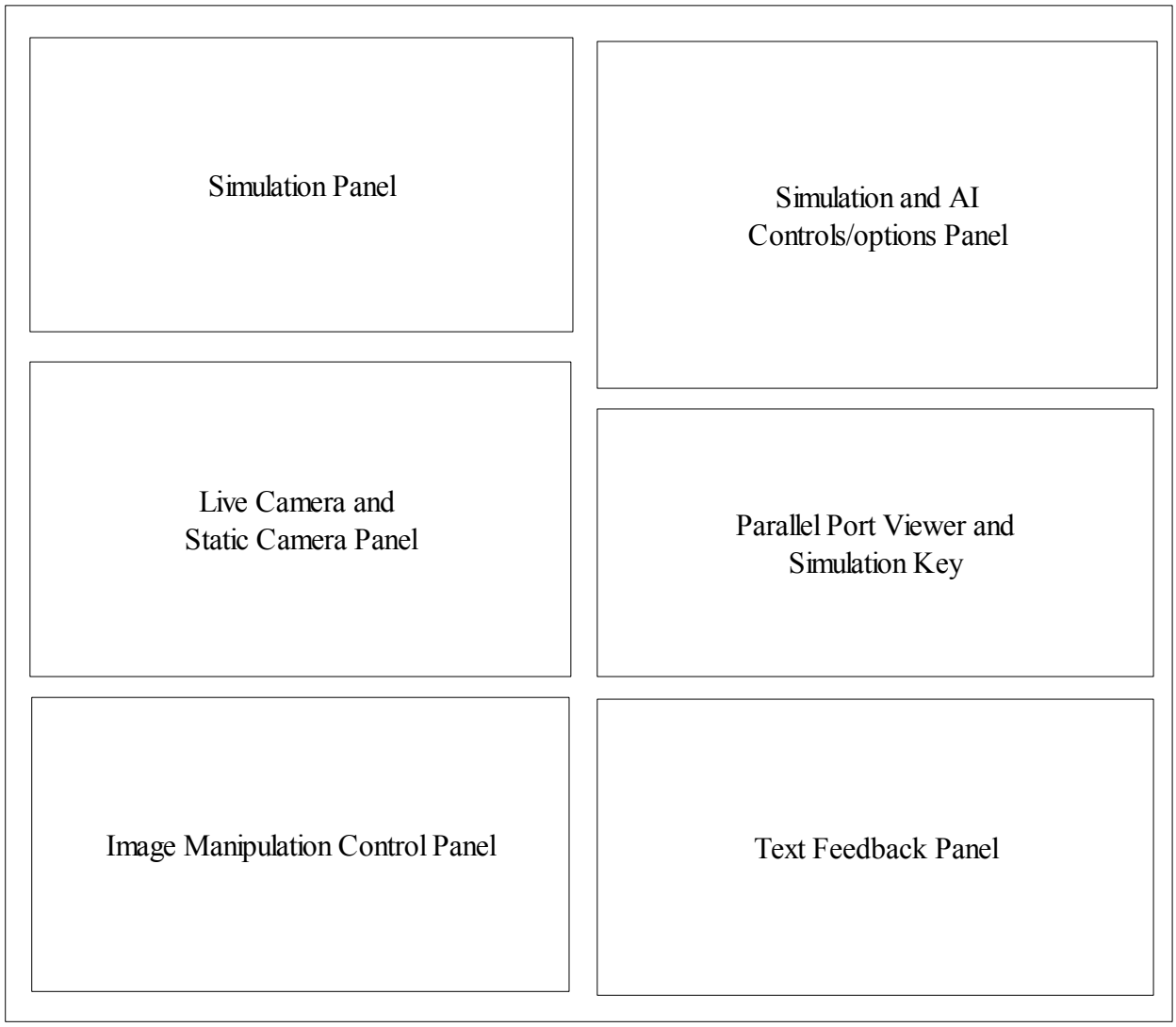


Figure 8 – GUI design mockup

The original code had the camera resized, and the simulation track taking up the major part of the GUI. To allow more features to be added, the simulation was resized to half the original, and the GUI was split into two halves. The left half held the simulation image, two camera panels (one was purely for debugging while one is the live image), and image processing controls. The right held the simulation controls, AI controls and debugging panels. The final GUI design is shown below in figure 9. More detailed information on the final interface controls are described in detail in Appendix D.

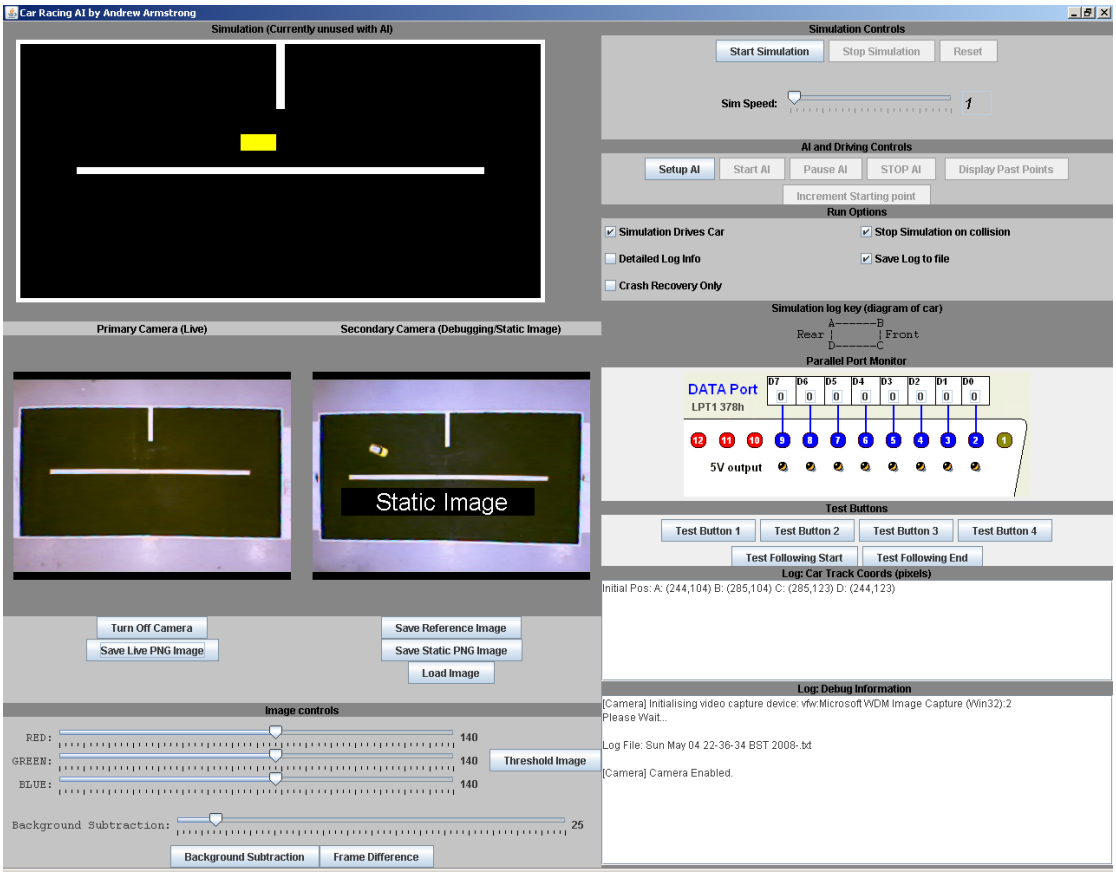


Figure 9 – Final GUI

Evaluation of Coding Methods

As noted in the previous project *(Carter, 2007)*, it was necessary to evaluate all the available methods to code the project since no standard solution to the given problems. However, several problems have existing solutions explored in previous projects *(Culcheth, 2006 and Carter, 2007)*, which is noted and referenced. In these cases, the evaluation was made much easier. The sections are in order of implementation, and also evaluate the design and structure decisions of the system.

Engine Implementation and Simulation Separation

It was decided immediately on starting to alter the functionality of the system provided by Carter's *(2007)* project code. The simulation was the core of the system and to work on additional interfaces needed segregating from the project code. To best achieve this, the class originally called DrawSim was renamed SimDraw, and a new Engine class was added to handle GUI actions, such as button presses and slider changes. Additionally, the entry class Simulation was renamed CarAIMain, although the functionality was relatively unchanged. The main addition to the CarAIMain class was logging functions, including saving logs to a file.

The simulation classes then were all renamed to properly segregate them from the rest of the system. The prefix "Sim" was chosen since several classes were already named with the prefix. The details of name changes are listed in figure 10. Apart from these changes, and the aforementioned resizing of the simulation display, the simulation system was left as it was.

Original Class Name	New Class Name
Simulation	CarAIMain
Action	SimAction
ActionHandler	SimActionHandler
Car2D	SimCar2D
Car	SimCar
CarControl	SimCarControl
DrawSim	SimDraw
Navigator	SimNavigator
TestCrash	SimTestCrash
TrackArea	SimTrackArea

Figure 10 – Simulation Classes Name Changes

Static Camera Addition

It was noted that a requirement of the GUI was to add a functional location to store static camera images. This was to aid debugging of the main camera while not overwriting the independently updating image.

This was simply implemented using a copy of the JImagePanel class, called JImagePanelStatic, and the CameraImage class, called CameraImageStatic. Both were very similar to the live counterparts, however, the CameraImageStatic class only allowed updates from within the program and had all the camera related functionality removed.

Image Processing

Previously, Culcheth *(2006)* had investigated several methods of car and track identification, which were image thresholding, background subtraction and frame differencing. The methods were re-implemented with some changes into the program and then evaluated.

Image Thresholding

A threshold can be applied to the webcam image to detect the track, as stated by Culcheth *(2006)*. This provides a binary image of black (empty) or green (full) pixels. The track is black, so this was detected by applying threshold values of red = 140, green = 140, blue = 140. In good lighting conditions this found the track area, and if the threshold needed adjusting, the sliders Culcheth *(2006)* had implemented were added to the new program. The results of

thresholding the track can be seen in figure 11, where it correlates to Culcheth's analysis and easily makes the track seen.

Figure 11 – Image Thresholding applied to empty track

Frame Differencing

The movement of the car needs to be detected and frame differencing is the method to detect changes between images. The original implementation by Culcheth *(2006)* was implemented and refined for use in the program.

Originally, Culcheth used frame differencing to detect the the changes between two frames where the car was already present. This caused some error when the care barely moved or had crashed, and no car was detected on the track. To solve this problem, frame differencing was used on images which had thresholding applied to them already, the original of which had the track detected with no car present. This is shown in figure 12 and 13.

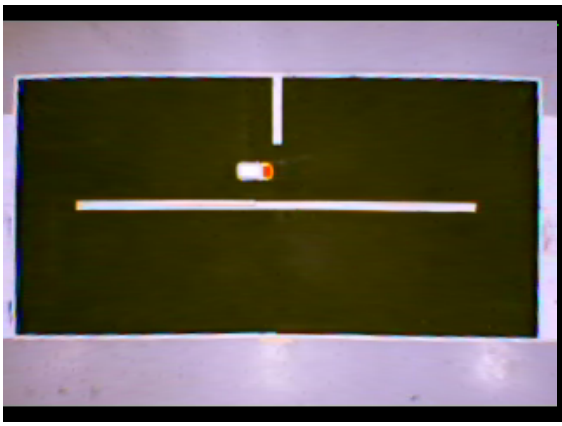


Figure 12 – Live image to be tested

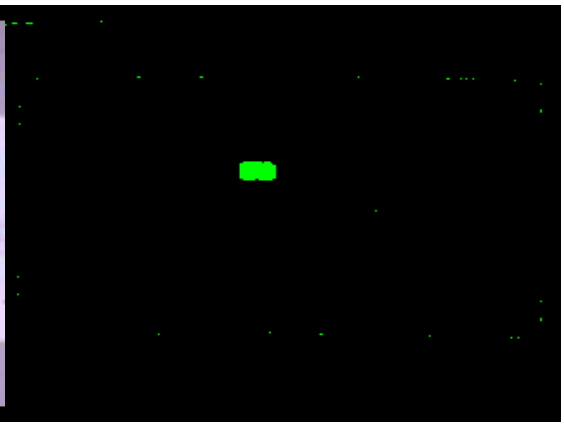


Figure 13 – Frame difference between current image after thresholding, and previously stored threshold image of the track

Background Subtraction

This is a special technique of frame differencing, and was investigated using Culcheth's original implementation. Background subtraction will check the difference between two live image frames and detect changes in pixels by a certain amount. However, obviously, lack of movement by the car meant a background subtracted image did not provide any feedback. This was tested with frame differencing and similar problems occurred.

Solution Chosen

Since the results from frame differencing two threshold images was very satisfactory, there was no need to use background subtraction to detect the differences between a past reference image and the current live image. This however was kept in the project for debugging, although is unused in the final implementation.

The final solution therefore used image thresholding to detect the track, and then this stored image and frame differencing to detect the location of the car. The major advantage of this method over background subtraction was the detection of the stationary car in all tests. This solution can also detect any obstacles on the track, although the system does not take them into account a future implementation could take advantage of the threshold images.

In addition, the frame differencing and thresholding suffered less from the

effects of camera movement and poor or changing lighting conditions then background subtraction, as noted by Culcheth *(2006)*.

Internal Model

The storage of the track internally is done to reduce overhead of processing, since it doesn't need to gather track information each frame. In addition, the car is difficult to analyse as a collection of image pixels, so a representation of it is required. This internal model in Culcheth's *(2006)* original implementation was partially successful at representing and finding the track. However, it was not full proof and was prone to some errors which should be avoided, since the internal barriers of the track were inaccurate.

Track Modelling

Rectangle Representation

Individual obstacles could be modelled as rectangles or shapes, for each of the walls or areas on the track. This is advantageous if the objects have definite edges, and if modelled as rectangles use up very little space in memory. They are also easy to use for collision detection since the area defined as obstacle is the entire area.

Accuracy is lost on exact edges of objects however, due to camera angles and non-perfectly square obstacles. This usually is not a problem due to the car wanting to avoid obstacles entirely, so additional buffer space around an obstacle can be a welcome addition.

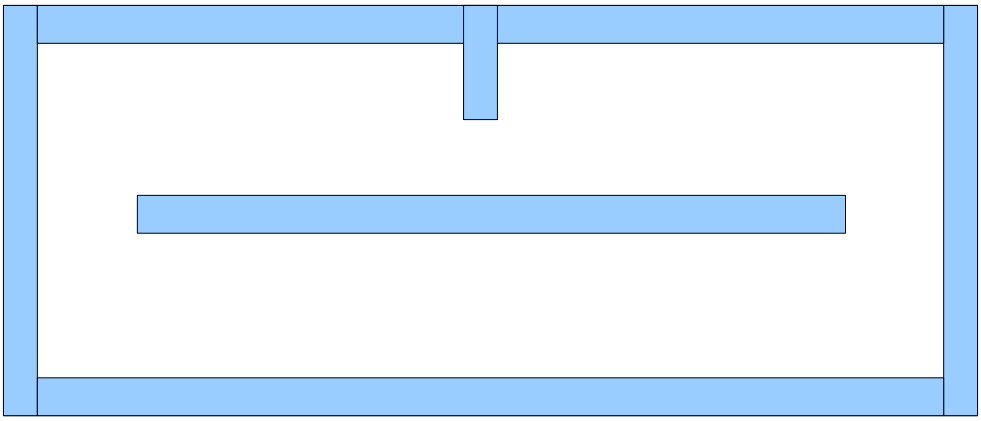


Figure 14 – Example Diagram of Individual Objects

Object Area Limit Representation

Similar to modelling individual objects, the limits of the obstacles could be modelled. This is much more advantageous for the walls, since they only have one side and do not require a width. For obstacles on the track it is less useful.

This method also relies on the camera being perfectly aligned with the track horizontally, since it allows no angles for the obstacles. This can be acceptable since, like with rectangle object representation overlapping parts of the valid track, the car will seek to avoid obstacles so the small amounts of valid track which are set to an obstacle will just be ignored.

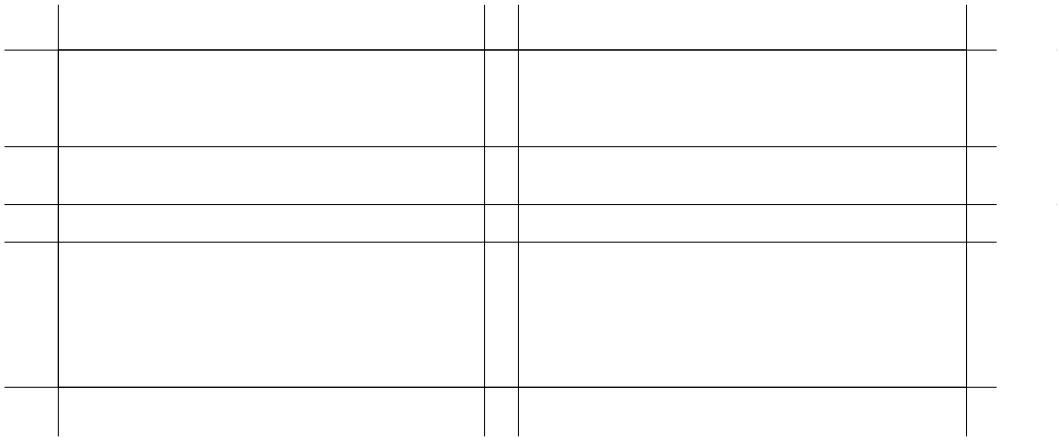


Figure 15 – Example diagram with limits applied to represent the walls and obstacles

Corner Point Representation

A more complex solution to modelling the track could be to find the corner points of an object to store, providing accuracy if the camera is at an angle, and usually better accuracy along lines for only having the obstacle represented and none of the track overlap. This method is difficult to apply, since it requires more complex collision detection since lines may be at an angle, but is more accurate overall.

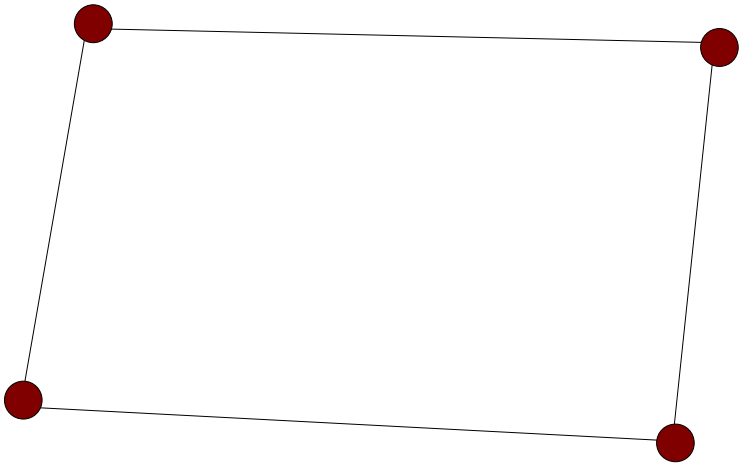


Figure 16 – Example of track walls possible representation by corner points

Chosen Solution

All three of the methods described can accurately portray the track, although the corner point representation is the most accurate. However, it also is the most difficult to implement reliably, since the camera is prone to providing poor data to find the corner points of obstacles and walls and the algorithms to implement the system are troublesome, as shown in Culcheth's project *(2006)*. The collision detection also is harder, and slower to perform then using right angle representations.

Therefore a mixture of obstacle area limits and rectangle object representation were chosen. The obstacle area limits were used to represent the track and internal walls. Internal walls representation was added since the track area might contain areas of the wall, and the internal walls can more accurately portray areas the car cannot go. However, both are required since the can could possibly hit the wall area and if that was the only representation, the car finding algorithm might fail. as seen in the comparison figure 16 below.

Figure 17 - Comparison of Track (in red) and Walls (in blue).

The boundaries of the track are the extreme points where there is no more valid track area. This uses a simple scanline algorithm, the horizontal version is available in the code sample figure 36, in Appendix A. The final result of the track limit representation is shown in Figure 18.

A special note must be made that by using the more accurate Pan and Scan camera mode, it appears as if track is available at the top and bottom of the image. This can stop the left and right track limits appearing correctly, since there would be no line made up of 100% black pixels. To sort this problem, the top and bottom limits were acquired first then used for the left and right limits.

Figure 18 – Track limit representation

In contrast, the walls are determined to start when there is no overlapping

wall with the track, by doing a scan of horizontal or verticle lines again, but with an 80% tolerance for track area, meaning it requires more then 80% of a scan line to be track for it to be considered part of the track. This is shown in figure 19.

Figure 19 – Internal Wall representation

Both of these models were created in the Track class, with all relevant functions to determine their location implemented as methods. In addition, the track limits were incorporated as an optimisation to further image analysis of the track, since we know that anything outside the limits is useless information. All other image analysis done after the track is found uses the track limits to speed up efficiency.

The chicane and barrier were represented as rectangles. To find them, the areas of black which represented them on the threshold image were estimated by checking the track limits. The largest limits of these areas were determined, and stored. The final representation of these obstacles can be seen in figures 20 and 21 below.

Figure 20 – Centre barrier detected **Figure 21 – Chicane detected**

Car Modelling

Point and Facing Angle Representation

A standard way to model the location of something is to take its most centre point and store it as a location. Because this doesn't show its facing, additionally an angle coordinate or the location of the front of the object must also be stored. This has the ease of allowing the system simple calculations between the centre point and any location it has to reach, however it doesn't allow for the precision detection of crash situations when the corner of the car might be stuck on some obstacle.

Car Corner/Rectangle Representation

As investigated by previous projects *(Culcheth, 2006 and Carter, 2007)*, the representation of each corner of the car is possible. This adds additional accuracy over the standard model, since collision detection is improved. However, this is difficult to model as detection of the corners is not perfect as noted by Culcheth *(2006)*. The simulation included in Carter's project uses this representation, but does no image analysis so is able to make use of the exact precision of the simulation.

Chosen Solution

The use of a single centre point was determined to be the best way to represent the car. This was to simplify the rule based navigation system and collision detection, as well as the navigation around the track route.

The method used to find this centre point and therefore location of the car

was based on a frame differenced, threshold image of the car being picked up as green as shown in figure 13.

This image was scanned, using the previously obtained track limits to speed up the search. Each pixel which was green had the 6x6 pixel area around it searched for green. The highest count was determined to be part of the car. Around this chosen point, the coordinates of all the green in 34x34 pixel area are averaged, and this new average coordinate should be the car centre. Figure 37 of Appendix A lists the initial car search algorithm.

After the car had been found once this was optimised so that it searched for the nearest collection of 20 pixels in a 6x6 pixel area, checking first where the last recorded car position was. Figure 22 below shows the location of the car as a red dot.

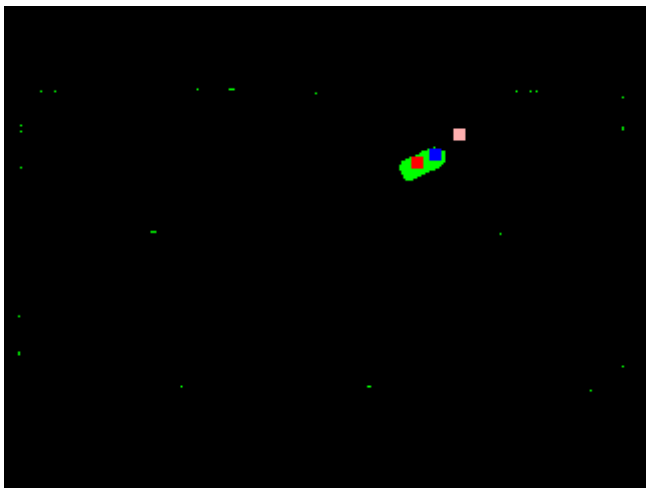


Figure 22 – Location of the car (red dot), front of the car (blue dot) and next waypoint (pink dot) are shown on the debug image.

Once this was found, the car facing must be determined. The method chosen was to search for a specific colour attached to the front of the car. Since the car was yellow, the colour red was chosen. This required the use of the live camera image unaltered, and a new algorithm that detects the average location of red pixels within a 26x26 area around the centre of the car was implemented. This is location that is found is shown as blue on figure 21. Additionally, the previous method used by Culcheth (2006) to detect the cars location by extrapolating from the previous location was used as a backup to this method. The problem was that reverse commands caused it to think the back of the car was the front, so it was used as last resort by the system.

The choice of what red object to use on the front of the car was highly tested, and is noted in the Testing section later on.

Driving AI

The core DrivingAI class was written as an interface between the decision making NavigationAI class and the parallel port commands. It would handle the start, stop, pausing of the AI system and use appropriate PWM code to cause the car to move.

The base class was written in a similar implementation to Carter's *(2007)* original DrawSim thread. This thread was started or stopped as dictated by the users of the system, or if the code was in a condition to end. There was the possibility of making it a simpler single threaded system, but this renders the camera unusable since it requires CPU time to receive new images for the system to use. The actions performed in the threaded loop can be seen in Appendix B, which shows a flow diagram of the loops main actions.

The controls added to allow the thread to run were Start AI, Pause AI, and STOP AI. Their location is detailed in Appendix D.

Car Control Mechanism

The control of the car was fully investigated in previous projects *(Culcheth 2006, Carter, 2007)*, and so did not need further work. The main control is provided by interfacing with a parallel port class created by Culcheth, which interacts directly with the remote control used for the car.

The controller can be sent on/off signals to command the car so it moves forward or backwards, and turns left or right. The signals directly map to pins on the parallel port, shown in figure 23 below.

	Byte (binary)	Coded Byte (0x denotes hex)	Parallel Port Pin(s) activated by byte	Instruction
Car 1	0000 0001	0x01	Data Pin 0	Left
	0000 0010	0x02	Data Pin 1	Right
	0000 0100	0x04	Data Pin 2	Forwards
	0000 0101	0x05	Data Pins 0 and 2	Forwards, Left
	0000 0110	0x06	Data Pins 1 and 2	Forwards, Right
	0000 1000	0x08	Data Pin 3	Backwards
	0000 1001	0x09	Data Pins 0 and 3	Backwards, Left
	0000 1010	0x0A	Data Pins 1 and 3	Backwards, Right
Car 2	As above, shifted 4 bits left	Range 0x10 to 0xA0	Data Pins 4 to 7	As above
Both	0000 0000	0x00	None	Off (Stop)
Instructions for both cars can be sent together (e.g. 0010 0101 means Car 1: Forwards & Left, Car2: Right)				

Figure 23 – Table of parallel port pins, against actions. *(Source: Carter, 2007)*

The car is controlled using Pulse Width Modulation (PWM) by these controlled. This turns on the control for a specified time, then have it off for a specified time. Using PWM allows fine tuned control of the car, and for the purposes of the design, adds reliability and control at the sacrifice of speed. Figure 24 shows an example PWM execution.

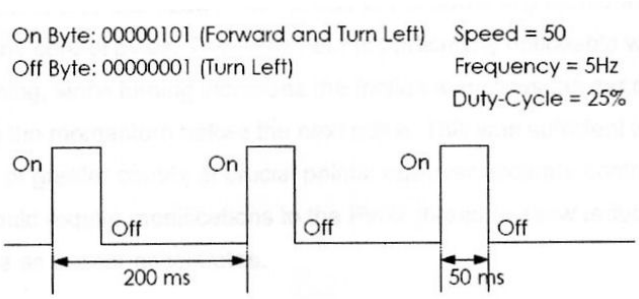


Figure 24: Example PWM execution *(Source: Culcheth, 2006)*

In addition Carter *(Carter, 2007)* implemented a parallel port viewer GUI to show what commands are being sent to what pins. This was kept in the program to allow debugging of commands being sent to the car if needed.

Navigation AI

The base class to handle navigation decision making first required the implementation of prototype decision making before the route planning was implemented. In addition, the class was created to interface with the models of the car and track. These were added when the class was created, with setup, initialization and update functions that the DriverAI class could operate with. Public variables were used to store the action that was chosen to be performed so the DriverAI could operate the parallel port. This segregation of decision making and operation allowed much cleaner interaction between classes instead of using a single huge class to do all the work.

The navigation rules themselves were implemented and evaluated after the route planning, and are described further on in this section. Additionally for debugging, the last 100 located car points were stored once the car and track classes were integrated, and are available to show on the GUI as the button "Display Past Points".

Track Route Planning System

There are several methods for determining the right direction to go around the track. Investigations were made into three potential methods, noting that the system did not need to learn and that the actual method of route navigation only had to be reliable enough for the rules to interact with.

Track Regions

Dividing the track up into regions which need to be traversed in order is a simple method to show the car a new place to get to. However, the method to determine the best route between each location still requires more work. Since the regions must cover all of the track, the usual way is to divide the track into quarters. This doesn't take into account the turning circle or control of the car, and it would be very difficult to optimise the route by any degree.

Simulated Route

There is the possibility of using Carter's *(2007)* simulation route code, and having the car analyse and use the route where possible. However, this is problematic since the list of instructions is very long and exact, meaning if the rules missed a turn it might be difficult to follow the route any further. While it is

good to have an optimised route for not crashing into the walls, it is difficult for the car to follow it using anything but the instructions themselves.

Waypoints

Setting up a series of waypoints allows the car to race towards a point then, once it has reached it, move on to the next one. The system also allows the manual placement of waypoints, simplifying the route choosing solution. As long as the system can detect when a waypoint is passed, forward movement should always be the main aim since the car essentially is following breadcrumbs around the track.

Chosen Solution

This decision was made to go with Waypoints after discussion with Dr Chris Hinde during during meetings. The situation requires only basic route planning, and no optimisation during the race itself. Since the track is static, and waypoints are easily added to the existing model system using the Coordinates class, it is the most simple to implement.

Iterative testing described in the Testing section further on shows that the placement and frequency of the waypoints was very important. The final waypoints chosen for the system to use are shown in figure 25 below.

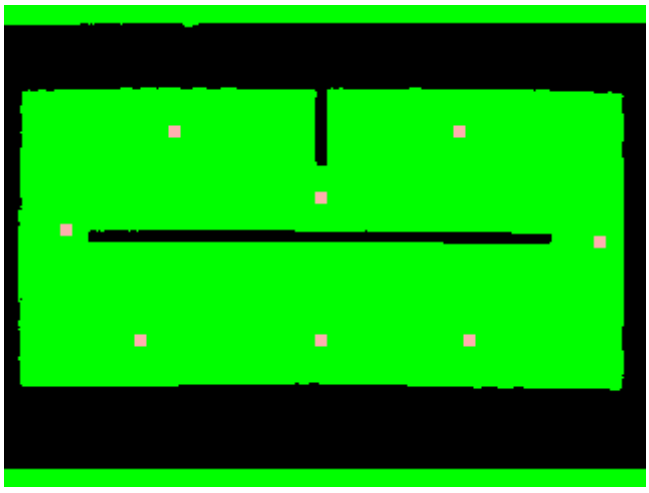


Figure 25 – Final waypoints in pink

The waypoints were stored in an array of coordinates. The Track class stored the required methods to find the initial waypoints locations, based on the track walls and obstacles, and to decide if the car had reached the current waypoint yet. This was done by a distance check from the waypoint to the cars

location every navigation update.

Navigation Rules Implementation

The navigation rules were determined to be the best method to implement navigation AI. While learning methods were investigated by Carter *(2007)*, he remarked that any learning system to be implemented needed several pieces of work, such as image analysis, to be achieved. Therefore, it was determined that learning AI would be out of the scope of the project. Instead, rule-based AI with a simple set of rules would be a workable solution to solving the aim.

Collision and Crash Detection

To detect walls and the obstacles, there are a few methods of analysis to find them around the car. However, many of these were investigated already by Culcheth *(2006)*. His recommended solution was to use extrapolation to find what was around the car.

Extrapolation is a simple method at determining if a obstacle is in a certain direction is to extrapolate a certain distance, or up to a certain distance, checking if points on the line hit an obstacle. This is relatively quick to perform, and can usually provide reliable results. Appendix A lists the algorithm used for this, and Appendix C lists the angles that are checked for obstacles. Each pixel is checked against the barriers, chicane and walls by the Track class.

Once this is completed, the navigation rules have most of the information required to determine the next action. The final piece of information is the location and direction to the current waypoint, so the application of rules can aim to get towards that point. This is detailed in the Track Route Planning section previously.

Navigation Rules

The navigation rules themselves deal with the aim of heading directly into the next waypoint. It always should try to make progress directly towards the location, turning as needed.

If the car is not facing the waypoint location, the set of rules generally will try to do a 3-point-turn driving manoeuvre, allowing the car to be turned fully around if they are facing the opposite direction. Since this would take many

different iterations of the rules, to achieve this without infinite loops the number of rules are limited. The general aim, since the direction around the track is clockwise, is also to reverse left, or go forwards right as much as possible.

The full rules in the code are listed in figure 38 of appendix A. Generally, the rules below in figure 26 are followed, although the code takes into account situations when the rule's result cannot be performed by taking into account if it can go that direction.

Rule Condition	Rule Result
Facing a Wall	
Car is facing the opposite direction to the waypoint	Go backwards, left
Car is facing the waypoint	Go backwards, left or right
Car is to the right of the waypoint	Go backwards, right (to turn forwards left later)
Car is to the left of the waypoint	Go backwards, left (to turn forwards right later)
Standard Navigation	
Car is facing the opposite direction to the waypoint, but more to the right	If we can reverse, go backwards right, else go forwards left
Car is facing the opposite direction to the waypoint, but more to the left	If we can reverse, go backwards left, else go forwards right
Car is facing waypoint	Move straight forwards
Car is facing to the right of the waypoint	Move forwards, left
Car is facing to the left of the waypoint	Move forwards, right
Final resort	
All other rules fall through	Reverse backwards, left

Figure 26 – Table of navigation rules

One problem encountered was that if a point is seen to be behind a barrier, the rules cannot cope with navigating around the barrier since it always attempts to go directly into the point. As noted in the Testing section, additional waypoints were added to achieve better results when navigating the track.

PWM Control Implementation

While the parallel port control had been implemented previously, the actual

length of the pulses used required some evaluation once the rules were sorted. After testing as noted in the Testing section, the final values for the pulses were best determined and are shown in figure 27.

Movement Pattern	Speed
Forwards, left or forwards, right	150 milliseconds
Forwards, straight	Half the forward milliseconds. Total: 75 milliseconds
Reverse (any direction)	+50 Milliseconds on top of forward speed. Total: 200 milliseconds

Figure 27 – Table of PWM action speeds

Update and Processing Speed

For a variety of reasons, the update and processing speed used by the system was set to a recurring interval of 700 Milliseconds. This was to achieve the fastest possible update time, but with these points in mind:

- The lab computer system was tested for speed and the processing took on average 200 milliseconds to do required work, but could take longer, up to 300 milliseconds in some cases.

- At least 100 milliseconds were required to send a sufficient pulse to the car to produce required movements.

- Additional time was required to slow down this movement or have it stop entirely, since as noted during the car modelling, the front of the car might blur at high speeds, making the front of the car difficult to find accurately due to motion blur on the camera.

- The aim of the system was not to produce a fast result but an accurate one. While faster speeds could be achieved, the above points would render it less accurate or prone to mistakes.

Therefore, after testing as noted in the Testing section, the 700 millisecond time with the overhead of 300 additional milliseconds allowed a new action to be chosen roughly every 1 second. In practice, this is frequent enough so navigation of the entire track takes no more then two minutes. To allow the system to make faster decisions without mistakes, improvements to the car facing algorithm, and

an upgrade to the lab computers would be necessary.

Testing

Incremental White Box Testing

Work on testing the design of the system started in an incremental feature by feature basis, since the system was designed around incremental addition of features to the existing program by Carter *(2007)*.

As noted in the design outline, each part of the system had several different possible solutions. Generally each were tested for appropriateness until the most satisfactory solution was obtained.

The early design and testing resolved around making the GUI suitable for additional controls and resizing the existing simulation display. Particular attention was made to separating the integration of GUI code from the simulation, and put into it's own engine class.

With this, necessary methods that would require work later on were prototyped with empty data and methods. This included the image processing and modelling classes.

During the incremental design of features, there were several complex sections to work on. The first complex feature to test was the image processing to gather the location of the car. While the track was easy to pick out and a solution was easily found, with appropriate image analysis work done by Culcheth *(2006)* added to the system, the system design required the cars location and facing to always be accurately known for navigation. Previous projects had only got a simple set of algorithms to find the car and it's facing from the past location, which would not work when tested since reversing caused it to record the facing as entirely the opposite way and the accuracy was poor.

To accurately find the car, it was determined the image threshold and frame differencing method was sound, but accuracy was hindered since the car itself had black windows and featured. The method used was to find the average point of all the area detected as the car, and so with the black features being detected as track, the centre of the car moved depending on which was it was facing and which windows were visible to the camera. This was solved through the application of paper to cover the black features, improving the accuracy. Initially, this was a flat piece of paper stuck onto the roof of the car. After it was

found that at certain angles, the car appeared to be much bigger then usual, so applying the paper as a cover over the car and it's black features allowed better accuracy, as shown in figure 28 and figure 29 below.

Figure 29 – White Paper applied to car

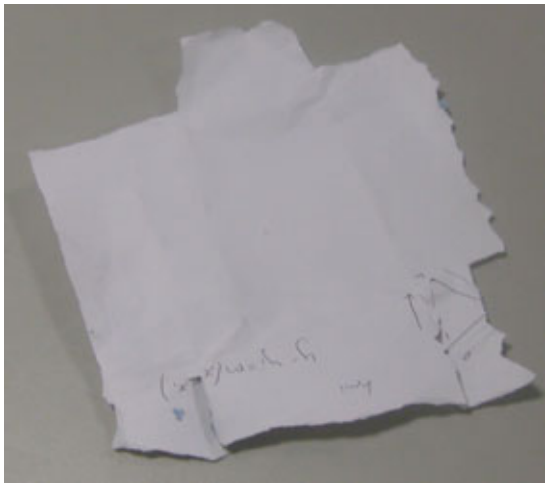


Figure 28 – White paper

Additionally, several iterations of testing went into finding the cars facing. Firstly testing was done on the possible solutions until picking up a colour on the front of the car was chosen to be the most suitable method. Then, a large amount of testing went into making this both accurate and reliable. The main problem encountered was the fact the initial testing with a Duplo Lego brick proved a problem when light directly reflected off it's shiny surface. The brick is pictured in figure 30 below, with some ambient light reflecting off it's surface despite not being positioned directly under a light.

Figure 30 – Duplo Lego block, note the shiny surface

A matte surface was required, big enough to be detected by the camera. Using a red marker on paper, with two different pressures, provided a matte surface to use. The final solution was to have one put underneath the other, since the larger piece of paper was too shiny in the middle, but provided an accurate edge to the smaller piece which was fully matte. Figure 31 and 32 shows the final paper used, and applied to the front of the car.

Figure 31 – Matte Red Paper

Figure 32 – Red paper applied to the front of the car

The final iteration of testing was on the control of the car and rules used for navigation. The basic controls used to command the car were available in the project already from the work by Culcheth *(2006)* and Carter *(2007)*. However, the rules and timings for navigation were not complete in any previous project, and so had to be tested thoroughly. The basic pulse width modulation was used, but it was noted over testing that the reverse movement of the car provided less

thrust then forwards motion for the same amount of pulse time. This meant the pulse time given for reverse commands was doubled, to allow a reasonable amount of movement from every choice of forwards or backwards.

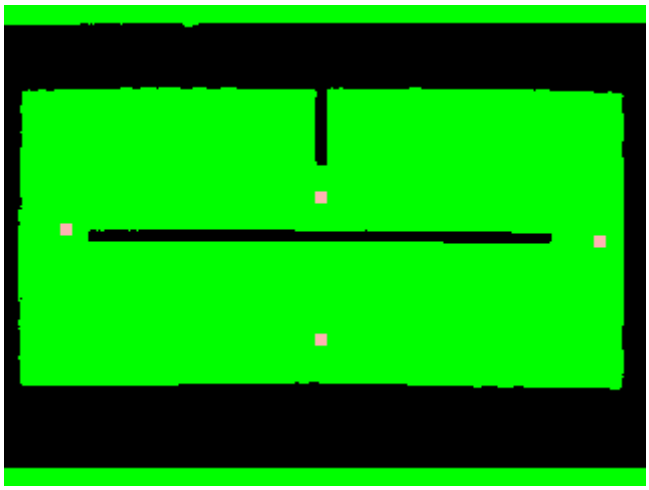
In addition, the decision of what action to take was first attempted every frame, which locked up the program requiring it to be slowed down to roughly every second. This in fact benefited the design since tracking the car was inaccurate when moving fast as the image blurred, making the car appear larger then normal and the detection of red was made more difficult. The delay allowed each pulse to finish before a new one was decided.

Finally, the last complex iterative testing was on the rules used to navigate the track and recover from crashes. These took some time to accurately sort out. Initially, the rules worked on the basis of knowing the number of pixels (and thus roughly the distance) to the nearest obstacle directly in front, behind, left and right of the car. These were partially accurate, especially at determining whether to reverse or move forward, but proved poor at judging if it could turn left or right and not hit anything, resulting in cases where it repeatedly tried to move into an obstacle it could avoid.

This was solved by the additional detection lines at angles roughly equivalent to where the car would be when going left or right. With these available, new conditions based on them, such as if there was an obstacle forwards and to the left of the car, were added, so it could not go in that direction. Appendix C shows the angles used for this collision detection.

The last iterative testing came from the chosen method of route planning, the waypoint system. This was based entirely on the manual creation of waypoints that the car had to navigate. At first, four waypoints were used, at the four main gaps in the track. Assuming a clockwise track direction, the number identifying them incremented until the last point, which was where the car started from. The first point was always right of the starting position in all the choice of waypoints. These initial waypoints are seen in figure 33 below.

Figure 33 – Initial waypoints.



The initial four waypoints did function in some key ways but were inefficient. The car usually managed to navigate without crashing to the first point, but due to the small gap between the track and the middle barrier, sometimes got stuck when turning into the first point, causing it to reverse, and try and then go through the barrier. It rarely completed a full track lap.

Therefore, more navigation points were added, to make the car follow a similar path found by Carter *(2007)* to be an optimal one for navigating the track, with the possibility of no crashes. An additional five points were added to the top and bottom areas of the track, to allow the car to follow a more optimal path and to not direct themselves into an obstacle after reaching a point. In addition, while tuning the location of the waypoints, the area which the car was detected to have reached a waypoint was increased and decreased by various amounts, improving the accuracy. The final waypoints when fully tested are displayed in figure 25.

Global System Testing

With the system at a functional state with all components implemented separately, the system as a whole required testing to see if it completed several necessary tests to complete the aim of the project.

The testing aimed to complete what was originally set out in the aim; to successfully navigate the track without human intervention. A set decision to achieve 10 track laps was the core aim since this was the guidelines setup in the CEC competition and also researched as a good target in previous projects *(Culcheth, 2006)*. Additionally, there are several tests to check the navigation when manually put in a crash position.

For the track laps, the system required an additional report on how many laps it had completed, which was implemented into the waypoint system. The crash recovery required that the system have a way to choose the next waypoint to reach. This meant that when the system is started, the AI could be on any part of the track and so long as the user has chosen an appropriate waypoint to move to next, it will do it's best to do so. In addition, an option to have it stop when it is facing or has reached the point in question was added, speeding up the testing. This option was called "Crash Test Only" and added to the options panel.

Battery Charge Conditions

As noted in previous projects *(Culcheth, 2006 and Carter, 2007)*, the performance of the car deteriorates when the batteries start to lose charge. Since this was a known problem, tests were done with rechargeable batteries which had less then 15 minutes of use. An additional check on the performance while at low battery was done, and navigation did suffer since the rules were tested on well charged batteries.

Test Conditions

The laps were all started from the initial start position shown in figure 34, which is marked on the track so the same position could be used each time.

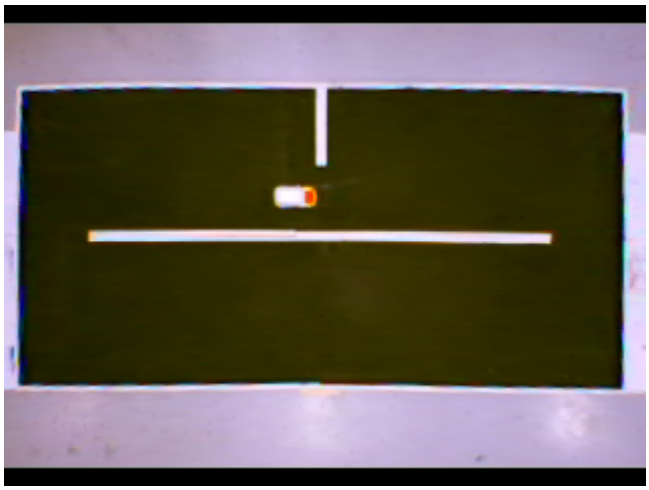


Figure 34 – Car in start position

The crash tests were done in the top left section of the track. This was deemed the most complex area to navigate after some initial investigation, and having the same area tested allowed comparisons to be made directly between the test cases. The car always aims during the crash tests to face or go over the car's starting location directly between the chicane and the barrier, and due to the addition of a "Crash Test Only" option will stop when this is achieved.

Testing Results

The tests and the results from them are detailed in table shown in figure 35 below. All the crash tests had recorded shots of the start and end position recorded in Appendix E. A recordings of one of each test was made, and are available on the project CD.

Each test was run 3 times, allowing for any possible errors to be spotted if one run of the test failed while the other passed.

Test	Result	Notes
Lap Tests		
Completion of 1 full track lap	All completed Successfully	No intervention was made, lap completed successfully.
Completion of 10 laps	All completed Successfully	No intervention was required to complete all 10 laps.
Crash Tests		
Facing Top Left Corner Test	All completed Successfully	Correctly faced towards waypoint
Facing Left Wall	All completed Successfully	Correctly faced towards waypoint
Facing Top Wall	All completed Successfully	Correctly faced towards waypoint
Facing Barrier	All completed Successfully	Correctly faced towards waypoint
Facing Chicane	All completed Successfully	Correctly faced towards waypoint
Facing Chicane and Wall Corner	All Failed	Stuck trying to reverse through barrier, with chicane directly in front of it. Did not succeed at facing the waypoint

Figure 35 – Test results table

Testing Evaluation

While most of the tests completed successfully, a problem occurs with one crash test, leading the system into an infinite loop. This problem obviously was not found during the iterative testing, and shows that the crash recovery, while it works for most of the general cases, can fail with some specific ones.

However, the test case used, facing the corner between the wall and the chicane, is unlikely to happen during an actual race due to the direction of travel. When the log is checked, the reason for the flaw is shown to be the navigation rules executing a reverse action since it decides the chicane directly in front of it is in the way. It in fact cannot reverse but it is the last given rule, and so is executed regardless. Error checking for this condition should have been added, to at least stop the system when it cannot navigate according to the rules given.

Overall however, the tests can be considered a success since apart from in some specific areas of the track, the car can navigate around successfully and get out of known crashes.

Conclusions

Project Successes

The most significant success in the project is the fulfilment of the aim to have the car traverse the track without human assistance, achieving the aim of 10 laps successfully. While the previous project *(Carter, 2007)* achieved the this aim successfully, the recovery from various crash positions, and avoidance of crashing if it happens, has proved to be a major accomplishment. Now the car can perform 10 laps of the track, or more, without human assistance. Compared to the 2005 winning entry into the CEC event by Ivan Tanev *(Essex, 2005ii)*, which uses a car with a tighter turning circle, it navigates correctly making good use of reverse and is a good solution to the problems put forwards regarding navigation.

While the car, under testing, did rarely get caught in an infinite loop of recurring instructions while testing the crash routines, the fact that it achieved 10 laps with no human assistance and additionally could start at nearly any location (navigating out of crash situations in the mean time) and navigate around makes this project very successful in archiving its core aim.

Additionally, the program implements several accurate image analysis techniques and this data is put to good use. The success at finding the cars location and facing for each new decision was not accomplished in previous projects and therefore is the first year to achieve such accuracy.

The project also provides a good system for building future improvements on to. The simulation created by Carter *(2007)* is still integrated into the program, which could be taken forwards. Small modifications to the design mean that more systems could be added to improve the system and implement more required features as needed.

Project Improvements

While the project achieves it's aims, there were some possible improvements when reviewed.

The waypoint system could have been implemented in a more reliable way, since it required several tests to fine tune the performance of the manually placed

locations. Sometimes the navigation still suffers if it misses a waypoint, and has to turn around to get near enough to it, meaning the car faces the wrong direction for the waypoint after it. If the system was implemented earlier, a set of rules to govern if a waypoint has been "passed" near enough would be beneficial, allowing the car to continue forwards rather then backtracking to a point it barely missed.

The crash recovery could have been perfected with the addition of more rules to the navigation system. This would have likely solved the failed test case and improve the systems reliability, at the expense of a lot more iterative testing time.

The navigation system itself also runs at a required low speed to retain accuracy. Updates are done every second or so, making the car movement slower then necessary. If the system had additional testing and functionality to detect movement in progress and prediction of where the car would end up, more frequent updates could provide a constant stream of movement rather then small pulses of it.

Personal Achievements

The project has provided a great deal of opportunity to learn several new techniques in a large Java project including the use of Threads, class interactions, and object orientated design, as well as further improving previous knowledge of the language. Additionally the knowledge of image processing and using models to store information, and on the control of the car by PWM was gained.

Future Work

Control Improvements

The current PWM timing is functionally slow, since it waits for a command to finish until analysing the car to perform the next command. This delay adds a long amount of time onto lap times, and although very reliable, is not fast. It could be improved greatly by the use of constant speed to move around the track, and the use of more frequent updates to control the car more effectively.

Alternative Track Layouts

Additional track layouts could be created, with an aim to have the internal model adapt to any new track layout presented to it with a general algorithm, since the current modelling can only apply to the specific track within this project.

Navigation Improvements

The use of manually placed waypoints, while reliable are not optimised for speed. There could be huge improvements to lap times by finding a suitable way to have the car navigate around the most optimal path.

Multiple Cars

A second car could be added since there are two *Nikko* cars in the range, which have different frequencies of 40Mhz and 27Mhz. The parallel port has support for a second car, and if image processing is applied to determine which car is which, the system could find the car which it is controlling to navigate the track correctly, avoiding the other car as needed.

Completion of Simulation

The simulation started by Carter *(2007)* is incomplete, but still functional in the system. This could be improved with the addition of reverse actions. The simulation then could be used to test for the best track navigation or to simulate the track and navigation within it without the need of the car itself.

General Improvements

The user interface could be improved with more clear controls, and the removal of unused sections such as the background subtracter which isn't used in

the system. The choice of the current waypoint to get to could be implemented in a drop down box, since currently a button must be pressed to cycle them. Since the waypoint system is static, allowing the GUI to be used to set or move the location of them would be highly useful as well.

In addition, speed controls would be useful for controlling the update speed of the AI, especially for testing.

Final Conclusions

With navigation of the track without human assistance, including recovery from crashes, the project is a great success. The system has a very reliable internal model and image processing techniques, and makes use of good navigation rules with a simple waypoint system.

Additional development should result in optimal navigation of the car around the track, allowing fast lap times and other improvements such as the addition of a second car to race against.

References

Barnard, C Loughborough University, 2007-2008
Personal Communication.

Bull, D. 2004
Java Colour Tracker
Available at: http://www.uk-dave.com/projects/java/colour_tracker.html
Consulted on: 02/05/08

Carter, C. Loughborough University, 2007
Computer Controlled Car Racing [2007 project]

Culcheth, S. Loughborough University, 2006
Computer Controlled Car Racing [2006 project]

HO, D, 2008
Notepad++
Website: http://notepad-plus.sourceforge.net/uk/site.htm
Consulted on: 04/05/08

Hinde, C. Dept. of Computer Science, Loughborough University, 2007-2008
Personal Communication and initial project creation

Margold, M, 2004
Download source of Parallel Port Viewer Java application source code
Website: http://www.planet-source-code.com/vb/scripts/ShowCode.asp?txtCodeId=4309&lngWId=2
Consulted On: 04/05/08

Portillo, J. 2005
ParPort Java package
Available at: http://www.geocities.com/Juanga69/parport/
Consulted On: 04/05/08

Mee, R. Dept. of Computer Science, Loughborough University, 2005
Parallel Port Controller Construction

Sun Microsystems, 2004
Java Media Framework Version 2.1.1e
Website: http://java.sun.com/products/java-media/jmf/index.jsp
Consulted on: 03/05/08

Sun Microsystems, 2006
Java Platform Standard Edition 6 API Specification
Website: http://java.sun.com/javase/6/docs/api/
Consulted on: 03/05/08

Tanev, I. Doshisha University (Japan), 2005
Winner of 2005 CEC Racing Cars Competition
Website: *http://isd-si.doshisha.ac.jp/itanev/*
Consulted on: 04/05/08

University of Essex, 2005(i)
CEC 2005 Car Racing Competition
Website: http://algoval.essex.ac.uk/cec2005/race/race.html
Consulted on: 02/05/08

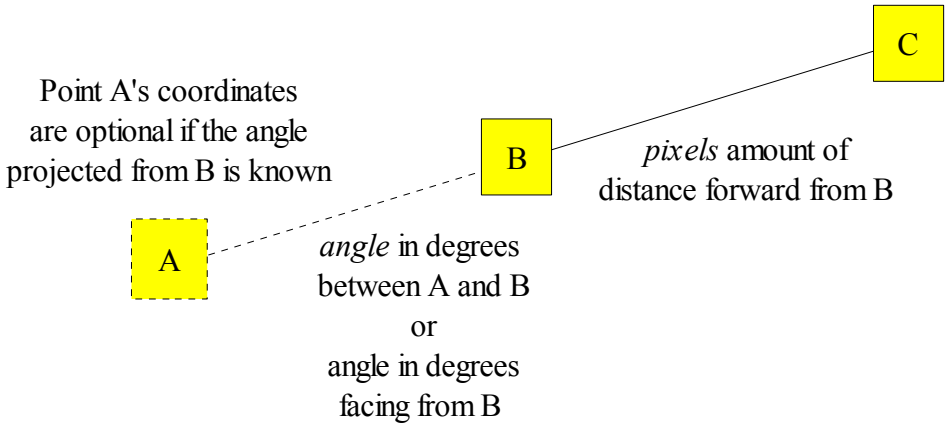
University of Essex, 2005(ii)
Video of winning entry of 2005 CEC Racing Cars Competition (provided by Tanev, I)
Available at: http://algoval.essex.ac.uk/cec2005/race/race.html
Consulted on: 02/05/08

Appendices

Appendix A: Key Code and Formulae

Projected Point

To get a projected point along a line, given an angle or two points this algorithm is used.



$$C.x = B.x + (pixels * \cos(angle))$$
$$C.y = B.y + (pixels * \sin(angle))$$

Key Code Samples

Horizontal Scanline Algorithm

Note the Vertical version is exactly the same, but reverses the X and Y inputs.

```
private boolean scanHorizontalLineForColor(int y, int minX, int maxX, int colorMatch,
BufferedImage image, int percent, int ignoreMinX, int ignoreMaxX)
{
    // For the % check
    double totalOfColour = 0.0, totalPixels = 0.0;

    // Error checking
    if(y >= image.getHeight()) y = image.getHeight() - 1;
    if(y < 0) y = 0;

    // Look between these co-ordinates
    for(int x = minX; x < maxX; x++)
    {
        // Cannot be within the boundried point
        if(x < ignoreMinX || x > ignoreMaxX)
        {
            // Check color - add up pixels
            totalPixels++;
            if(image.getRGB(x,y) == colorMatch)
            {
                totalOfColour++;
            }
        }
    }
    if(totalOfColour > 0)
    {
        // Do we have the % work? (integer division)
        double percentLine = (totalOfColour / totalPixels) * 100;
        if(percentLine >= percent)
        {
            return true;
        }
        else
        {
            return false;
        }
    }
    // Not found on this line
    return false;
}
```

Figure 36: Horizontal Scanline Algorithm. Note logging functions have been removed to improve code clarity.

Find Initial Car Location

```
// getCarPosition
// Will get the position of the car wherever it is on the track, returns true if successful, else false
// This doesn't need the last car coordinates.
public boolean getCarPosition(BufferedImage liveImage, BufferedImage image) {
    // Require good images
    if(liveImage == null || image == null || liveImage.getWidth() == 0 || image.getWidth() == 0)
    {
        return false;
    }
    if(liveImage.getWidth() != image.getWidth())
    {
        return false;
    }
    // Now search the image. We for now work on a x=0, y=0 to the max loop.
    int xCar = 0, yCar = 0, matches = 0;
    // This is the area size (2 * countPixelAreaSize * 2 * countPixelAreaSize total pixels) around a full pixel we look to see if there is enough to consider the car being part of the area
    int countPixelAreaSize = 3;
    // This is the area we use to average and find the centre of the car location. We find quickly a tiny location which should be it, and average all nearby points of relevant "fullness" to get the general centre
    // Experiment with this number if it isn't entirely accurate all the time.
    int averageLocationAreaSize = 17;
    // This should be set to the "minimum" area that can be considered part of the car. For performance, set lower.
    int minimumValidPoints = 20;
    int highestmatches = 0;

    // Loop the pixels
    for (int x = trackMinX; x < trackMaxX; x++)
    {
        for (int y = trackMinY; y < trackMaxY; y++)
        {
            // Get the RGB values here. If full, we will search around
            if(image.getRGB(x,y) == MATCH_FULL_REGION)
            {
                // Check the pixels in this area (+/- 6 from coordinates)
                matches = countPixelsInArea(x, y, countPixelAreaSize, MATCH_FULL_REGION, image);

                // Has it found a large enough area?
                if(matches > highestmatches)
                {
                    // This is a place we want to investigate further!
                    highestmatches = matches;
                    xCar = x;
                    yCar = y;
                }
            }
        }
```

```
    // Do we have something which has at least the minimum number of valid points?
    if(highestmatches > minimumValidPoints)
    {
        // Average out
        Coordinate xyAverage = findAverageColorLocationInArea(xCar, yCar, averageLocationAreaSize,
MATCH_FULL_REGION, image);
        // Now we have this car's location find it's facing
        double facing = findCarFacing(xyAverage.x, xyAverage.y, liveImage, image);

        // Errors?
        if(facing < 0)
        {
            // Cannot be negative, we have a problem. We simply don't know the facing!
            parentCarAIMain.logDetail("[Car] getCarPosition. Error finding facing we cannot get
out of, please retry! Error occured.");
            return false;
        }
        // Set since no errors, we have the last location of the car
        lastCarCoords.x = xyAverage.x;
        lastCarCoords.y = xyAverage.y;

        // Got a centre location for our car, hopefully!

        // Draw the car
        camera.drawBlock(xyAverage.x, xyAverage.y, image);
        // Put in debug location
        parentCarAIMain.staticCamImg.image.setImage(image);
        return true;
    }
    // Put in debug location
    parentCarAIMain.staticCamImg.image.setImage(image);
    return false;
}
```

Figure 37 – getCarPosition. Note some logging has been removed to clarify the code.

Navigation Decision

```
// We need to decide what to do when navigating. This takes the decision and passes the variables to directionChosen and forwardBackwardChosen.
public boolean navigateDecision() {
    // We have the car's location and facing

    // Add the new car loaction into our waypoint array
    addToPastLocationsArray(car.lastCarCoords);

    // If we get stuck we'll stop after 10 seconds of being stuck - not much distance moved!
    double pastDistance = distanceTravelledRecently();

    // Debug: Report distance
    parentCarAIMain.logDetail("[Navigation AI] navigateDecision(). pastDistance [" + pastDistance
+ "] pastLocationsArrayIndex [" + pastLocationsArrayIndex + "]");

    if(pastDistance > 0 && pastDistance < 5)
    {
        // Error encounted, don't do anything...
        parentCarAIMain.logDetail("[Navigation AI] navigateDecision(). Error! No progress made!
pastDistance is [" + pastDistance + "] pastLocationsArrayIndex [" + pastLocationsArrayIndex + "]");
        return false;
    }

    // Check where it is in relation to walls.
    // We use angles based off the car facing to the right.
    //      BL (210)      Left (270)     FL (330)
    //      Back (180)                   Forward (0)
    //      BR (150)      Right (90)     FR (30)
    // Add the calculated car facing to get these angles (doesn't matter if it goes over 360 of course).
    // Maximum returned value is 150 pixels, which is all we need
    // Note on car sizes: we are around
    // Additional; also need back/forward left/right to make sure we can turn in those directions - we use angles of 30 degrees.
    int wallFrontDistance = (int)getDistanceToNearestObstacle(car.lastCarCoords, car.lastCarFacing
 + 0);
    int wallFrontRightDistance = (int)getDistanceToNearestObstacle(car.lastCarCoords, car.
lastCarFacing + 30);
    int wallRightDistance = (int)getDistanceToNearestObstacle(car.lastCarCoords, car.lastCarFacing
 + 90);
    int wallBackRightDistance = (int)getDistanceToNearestObstacle(car.lastCarCoords, car.
lastCarFacing + 150);
    int wallBackDistance = (int)getDistanceToNearestObstacle(car.lastCarCoords, car.lastCarFacing
+ 180);
    int wallBackLeftDistance = (int)getDistanceToNearestObstacle(car.lastCarCoords, car.
lastCarFacing + 210);
    int wallLeftDistance = (int)getDistanceToNearestObstacle(car.lastCarCoords, car.lastCarFacing
+ 270);
    int wallFrontLeftDistance = (int)getDistanceToNearestObstacle(car.lastCarCoords, car.
lastCarFacing + 330);

    // Need angle to the next waypoint
    double waypointAngle = Utility.calcAngle(car.lastCarCoords, track.getCurrentWaypointLocation
());
```

```
        // We now need to decide the relative limits of left/right so we can decide which way to turn.
        // waypointAngle to waypointAngle + 180 == We are facing to the right, so need to turn left
        // waypointAngle to waypointAngle - 180 == We are facing to the left, so need to turn right
        // We also have the fact we can overlap with "0" or "360", it will loop around (so we we have "right" as from "190 - 370" it will really take 190 - 360, then 0-10 too.
        double rightAngleLimit = waypointAngle + 180;
        double leftAngleLimit = waypointAngle - 180;

        // Check these distances, they determine what we want to do...
        parentCarAIMain.log("[Navigation AI] navigateDecision(). Distances to obstacles. F [" + wallFrontDistance + "] FR [" + wallFrontRightDistance + "] R [" + wallRightDistance + "] BR [" + wallBackRightDistance + "] B [" + wallBackDistance + "] BL [" + wallBackLeftDistance + "] L [" + wallLeftDistance + "] FL [" + wallFrontLeftDistance + "]. Car facing [" + car.lastCarFacing + "] Current Waypoint [" + track.getCurrentWaypoint() + "] and waypointAngle [" + waypointAngle + "]");

        // Error checking
        if(wallFrontDistance <= 30 && wallBackDistance <= 30)
        {
            parentCarAIMain.log("[Navigation AI] navigateDecision() Wall in front and behind us, cannot move!");
        }

        // The usedCorrectFacing flag is used for crash recovery - if we do move forwards, we must be facing the point (more or less)
        usedCorrectFacing = false;

        // ********************************
        // FACING A WALL
        // ********************************

        // Facing a wall...
        // This is the biggest barrier to progress, of course!
        if(wallFrontDistance <= 30)
        {
            parentCarAIMain.log("[Navigation AI] navigateDecision() Wall in front of us");

            // We want to reverse, but what direction?

            // If the waypoint is entirely the wrong direction, we always turn to the RIGHT since the track is based on going around almost entirely right
            if(getAreOppositeFacing(car.lastCarFacing, waypointAngle, rightAngleLimit, leftAngleLimit))
            {
                // Reverse LEFT, since this will make us able to turn RIGHT more
                forwardBackwardChosen = BACKWARD;
                directionChosen = LEFT;
                parentCarAIMain.log("[Navigation AI] navigateDecision() Wall in front of us. We are facing wrong way, so reversing left.");
                return true;
            }
            // If we are actually facing right at the target, we need to turn away from the given wall
            else if(getAreCorrectFacing(car.lastCarFacing, waypointAngle, rightAngleLimit, leftAngleLimit))
            {
```

```
            parentCarAIMain.log("[Navigation AI] navigateDecision() Wall in front of us. We are
facing waypoint, avoiding the wall by reversing.");
            // Which side has more space? Move backwards and opposite to that direction
            forwardBackwardChosen = BACKWARD;
            if(wallRightDistance > wallLeftDistance)
            {
                directionChosen = LEFT;
            }
            else
            {
                directionChosen = RIGHT;
            }
            return true;
        }
        else if(getAreRightFacing(car.lastCarFacing, waypointAngle, rightAngleLimit,
leftAngleLimit))
        {
            parentCarAIMain.log("[Navigation AI] navigateDecision() Wall in front of us. We are
on right side of waypoint. Turning back/right to turn left later.");
            // For now, if we are on the right side we should just reverse right (since this gets us more "left")
            forwardBackwardChosen = BACKWARD;
            directionChosen = RIGHT;
            return true;
        }
        else
        {
            parentCarAIMain.log("[Navigation AI] navigateDecision() Wall in front of us. We are
on left side of waypoint. Turning back/left to turn right later.");
            // Opposite if we are on the left side, we want to turn left now to get into a position where in the future we can turn right
            forwardBackwardChosen = BACKWARD;
            directionChosen = LEFT;
            return true;
        }
    }

    // ********************************
    // STANDARD NAVIGATION
    // ********************************

    // This occurs if there are no given direct obstacles to our progress

    // If the waypoint is entirely the wrong direction, we always turn to the RIGHT since the track is based on going around almost entirely right
    if(getAreOppositeFacing(car.lastCarFacing, waypointAngle, rightAngleLimit, leftAngleLimit))
    {
        // It is on the right of us
        if(getAreRightFacing(car.lastCarFacing, waypointAngle, rightAngleLimit, leftAngleLimit))
        {
            if(wallBackDistance > 30)
            {
                // Reverse opposite location
                forwardBackwardChosen = BACKWARD;
```

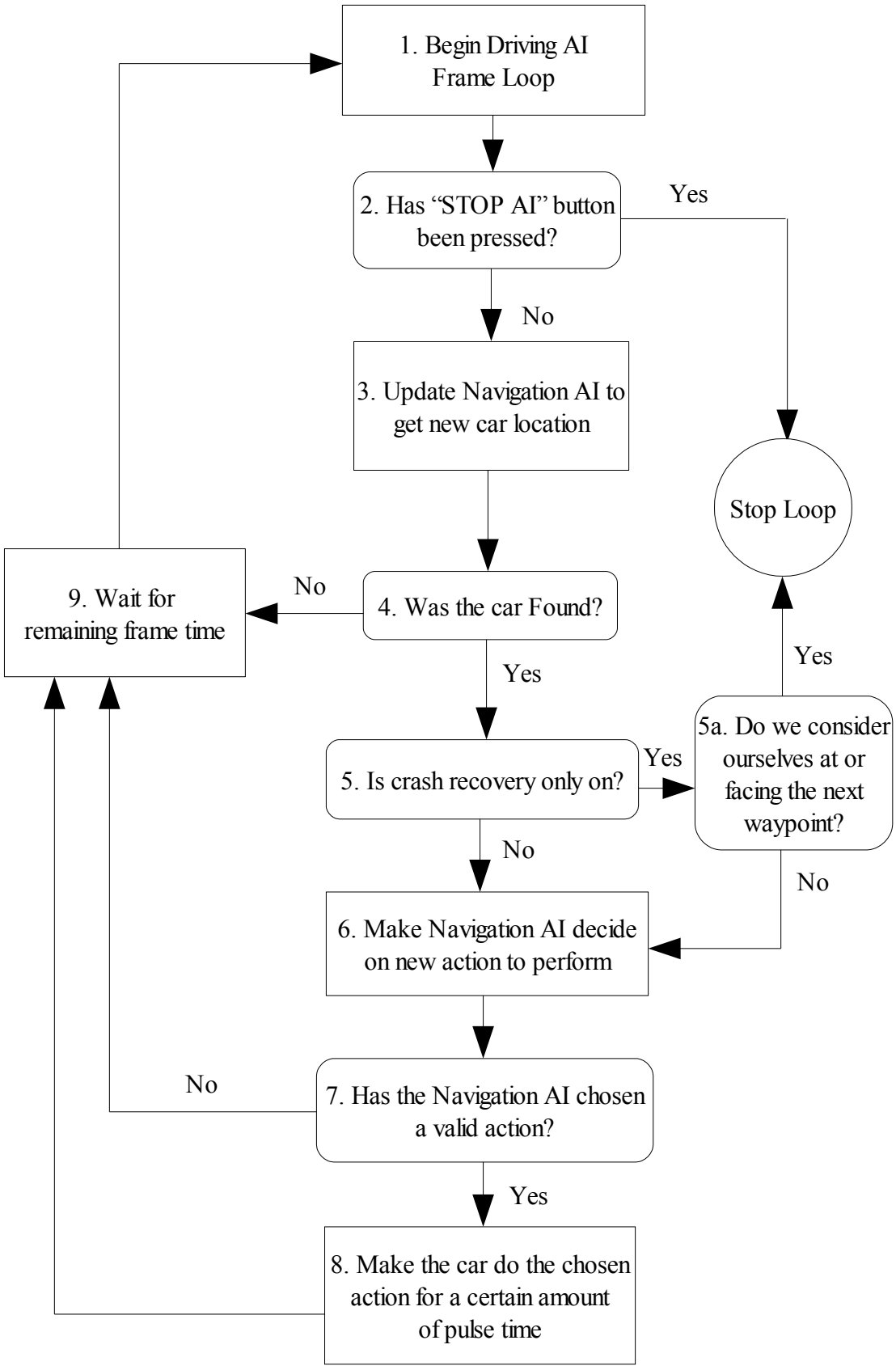
```
                directionChosen = RIGHT;
                parentCarAIMain.log("[Navigation AI] navigateDecision() No obstacle. We are
facing wrong way, it's on the left side, reverse back right.");
                return true;
            }
            else
            {
                // Move forwards
                forwardBackwardChosen = FORWARD;
                directionChosen = LEFT;
                parentCarAIMain.log("[Navigation AI] navigateDecision() No obstacle. We are
facing wrong way, it's on right side, forward left.");
                return true;
            }
        }
        else
        {
            if(wallBackDistance > 30)
            {
                // Reverse opposite location
                forwardBackwardChosen = BACKWARD;
                directionChosen = LEFT;
                parentCarAIMain.log("[Navigation AI] navigateDecision() No obstacle. We are
facing wrong way, it's on left side, reverse back left.");
                return true;
            }
            else
            {
                // Move forwards
                forwardBackwardChosen = FORWARD;
                directionChosen = RIGHT;
                parentCarAIMain.log("[Navigation AI] navigateDecision() No obstacle. We are
facing wrong way, it's on left side, forward right.");
                return true;
            }
        }
    }
    // If we are actually facing right at the target, we need to turn away from the given wall
    if(wallFrontLeftDistance > 12 && wallFrontRightDistance > 12 && getAreCorrectFacing(car.
lastCarFacing, waypointAngle, rightAngleLimit, leftAngleLimit))
    {
        // This would be the best one to do of course!
        usedCorrectFacing = true;
        parentCarAIMain.log("[Navigation AI] navigateDecision() No obstacle. We are facing
waypoint, move forward only.");
        forwardBackwardChosen = FORWARD;
        directionChosen = STRAIGHT;
        return true;
    }
    if(wallFrontLeftDistance > 15 && wallFrontRightDistance > 12 && getAreRightFacing(car.
lastCarFacing, waypointAngle, rightAngleLimit, leftAngleLimit))
```

```
    {
        parentCarAIMain.log("[Navigation AI] navigateDecision() No obstacle. We are on right side
of waypoint. Turning left.");
        forwardBackwardChosen = FORWARD;
        directionChosen = LEFT;
        return true;
    }

    if(wallFrontRightDistance > 15 && wallFrontLeftDistance > 12)
    {
        parentCarAIMain.log("[Navigation AI] navigateDecision() No obstacle. We are on left side
of waypoint. Turning right.");
        forwardBackwardChosen = FORWARD;
        directionChosen = RIGHT;
        return true;
    }
    else
    {
        // Move backwards-left
        parentCarAIMain.log("[Navigation AI] navigateDecision() No obstacle. We are on left side
of waypoint. Cannot turn right, so going back-left as last resort.");
        forwardBackwardChosen = BACKWARD;
        directionChosen = LEFT;
        return true;
    }
}
```

Figure 38 – navigateDecision - Navigation decision code and rules

Appendix B: Driving AI Loop Flow Diagram



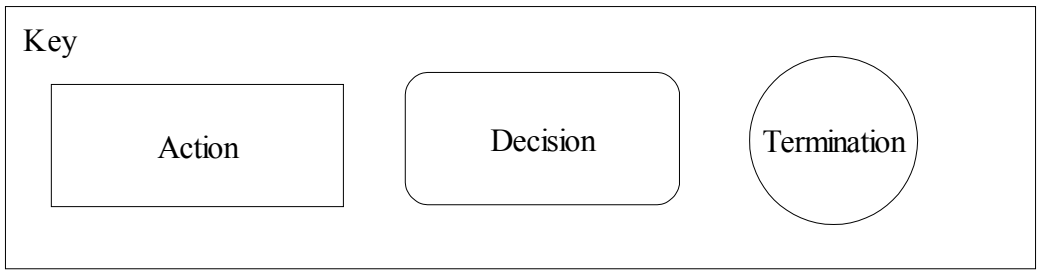


Figure 39 – Driver AI Loop flow diagram

Appendix C: Object Detection Angles

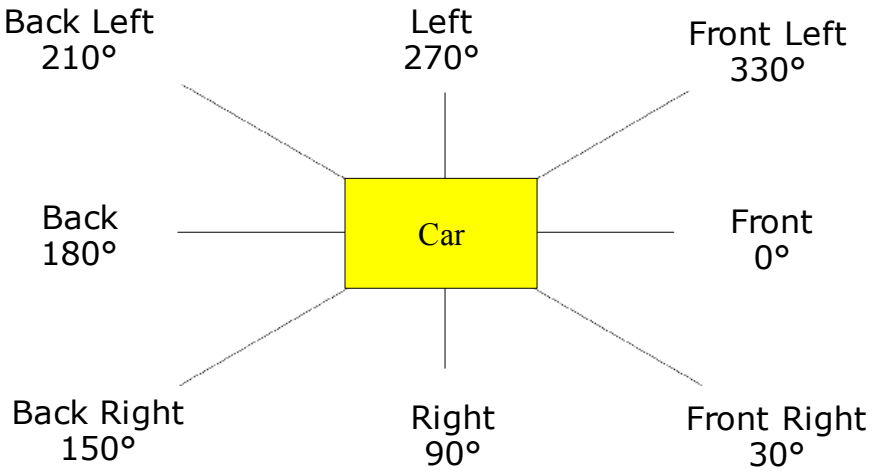


Figure 40 – Diagram to show the angles which detect barriers in the path of the car

Appendix D: GUI Functionality

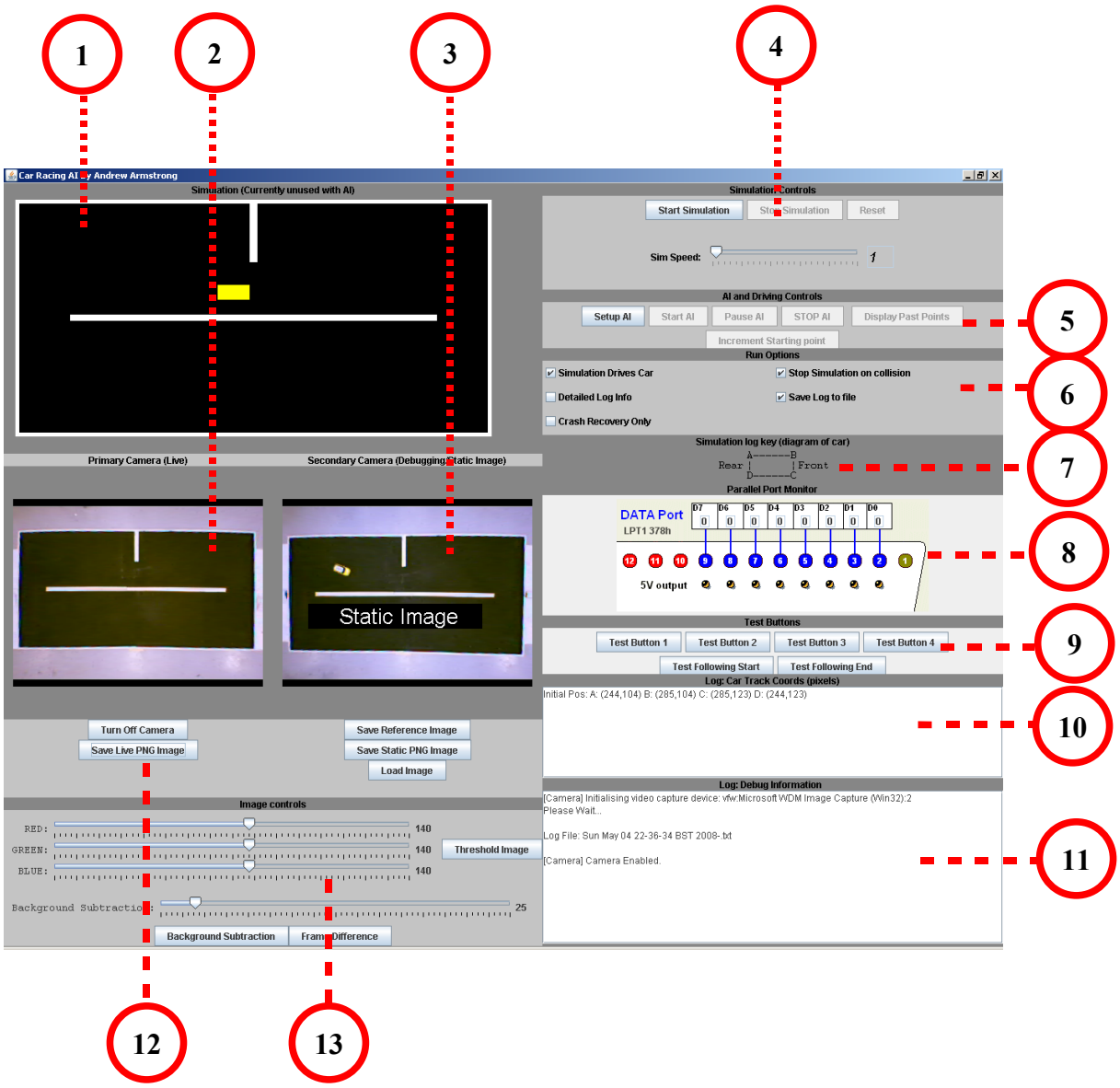


Figure 41 - Final GUI with numbered identification

1. Simulation Interface – Scale is 1px:4mm
2. Primary Camera viewing panel (Live camera)
3. Secondary Camera viewing panel (Static camera / debugging image)
4. Simulation controls. These are set to enabled or disabled depending on the simulation state.
    a) Start Simulation – Starts the simulation
    b) Stop Simulation – Stops a running simulation
    c) Reset – Resets the simulation that is running
5. AI Controls. These are set to enabled or disabled depending on what the AI state is.
    a) Setup AI – Sets the AI up ready to run and finds the track
    b) Start AI – Starts the AI, so it finds the car and proceeds to navigate
    c) Pause AI – Pauses the AI loop. It can be resumed by pressing the button again (it is renamed "Resume AI" when it is pressed)
    d) STOP AI – Stops the running AI immediately
    e) Display Past Points – Displays the last 100 tracked locations the car has been found at.
    f) Increment Starting Point – Increments the waypoint that the car should aim to get to
6. Run Options. These provide a set of options for the simulation and AI.
    a) Simulation Drives Car – If turned on, the simulation will run the instructions on the car controller in addition to the simulator.
    b) Stop Simulation on Collision – Will stops the simulation if it goes into a barrier when turned on.
    c) Detailed Log Info – Provides in depth logging information to the debug panel.
    d) Save Log to file – When turned on, will save the log entries to a log file.
    e) Crash Recovery Only – When turned on, once the AI faces or reaches the next waypoint, the AI will stop running.
7. Simulation Key. Logging will output A, B, C, and D locations of the car, which are shown in this diagram.
8. Parallel Port Data Pin Monitor. Shows the current parallel port actions.
9. Test buttons. These cause debug and test functions to be called.
10. Car Track Coords Log. Car location coordinates are displayed in pixels here.
11. Debug Information Log. Shows debug information on what the system is

doing.

12. Camera Control Buttons Panel.
    a) Turn On/Off Camera – Turns the live web camera feed on or off.
    b) Save Live PNG Image – Saves the current live webcam frame to a PNG file.
    c) Save Reference Image – Saves a copy of the current static camera image for use in the image controls
    d) Save Static PNG Image – Saves the current static camera frame to a PNG file.
    e) Load Image – Loads a file from the computer and puts it in the static image location.
13. Image Controls
    a) Threshold Sliders – Allows the control of what threshold RGB settings are used.
    b) Threshold Image – Does a sample threshold analysis of the current live webcam image frame
    c) Background Subtraction Slider – Allows the control of the image difference needed for background subtraction.
    d) Background Subtraction – Performs a sample background subtraction between the current static image and a previously stored image (from the "Save Reference Image" button)
    e) Frame Difference – Performs a single frame difference between the current static image and a previously stored image (from the "Save Reference Image" button)

Appendix E: Crash Testing Results

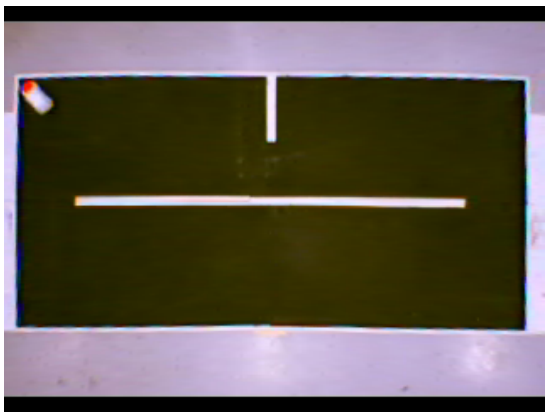


Figure 42 – Corner Test - Start

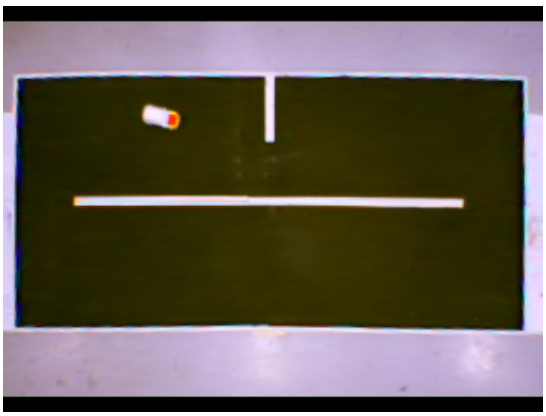


Figure 43 – Corner Test - End

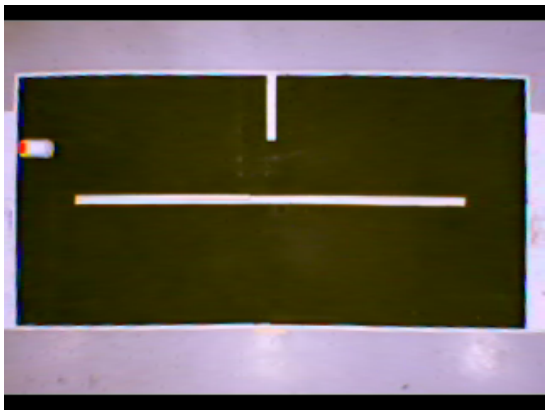


Figure 44 - Facing Left Wall Test - Start

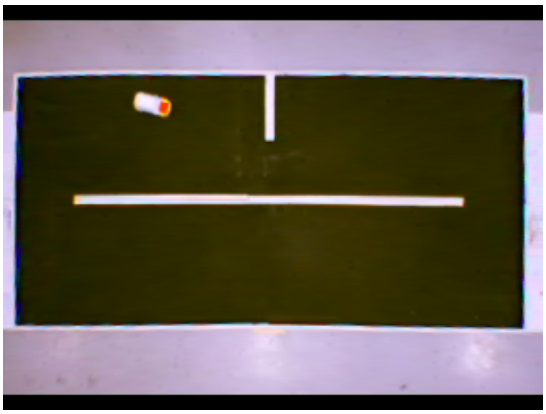


Figure 45 – Facing Left Wall Test - End

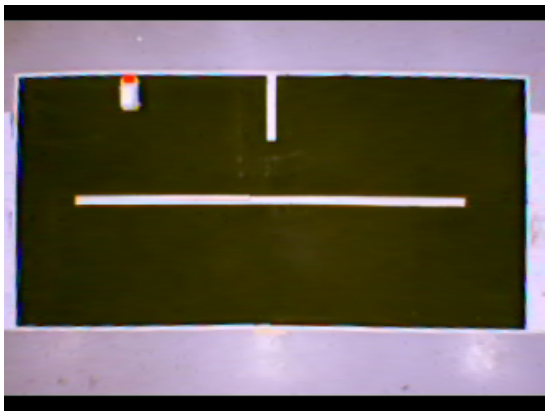


Figure 46 – Facing Top Wall Test - Start

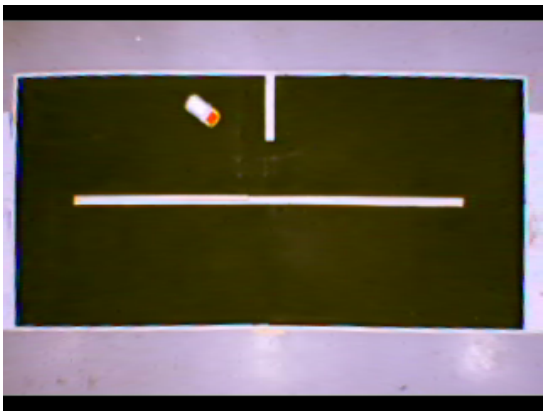


Figure 47 – Facing Top Wall Test - End

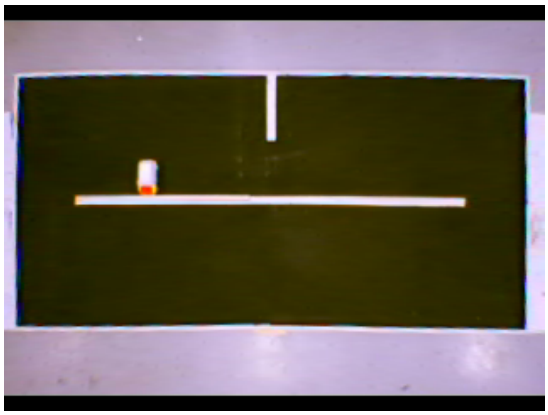


Figure 48 – Facing Barrier Test - Start

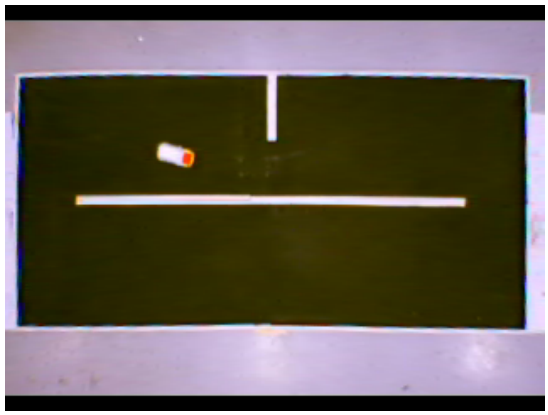


Figure 49 – Facing Barrier Test - End

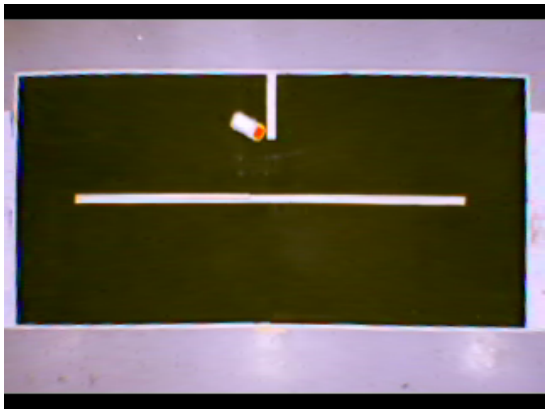


Figure 50 – Facing Chicane Test - Start

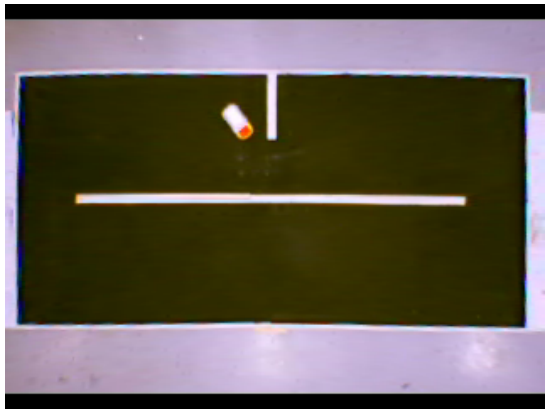


Figure 51 – Facing Chicane Test - End

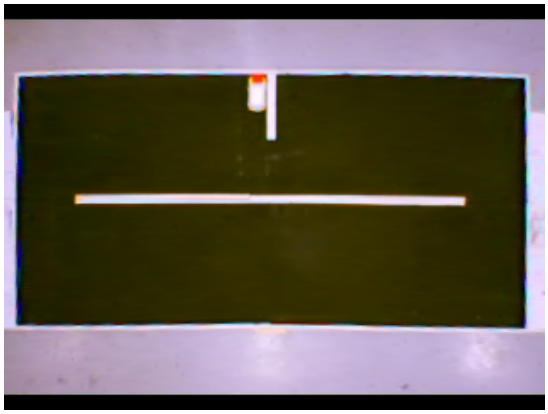


Figure 52 – Corner of Chicane Test - Start

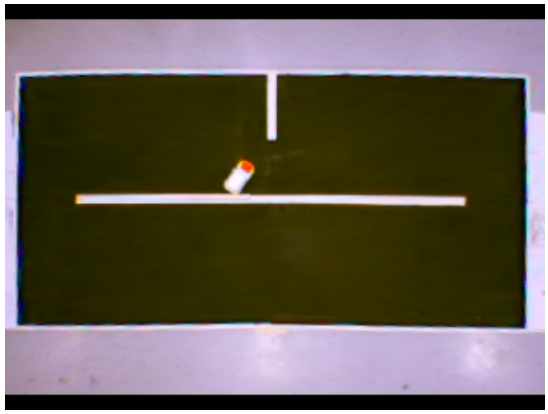


Figure 53 – Corner of Chicane Text - End