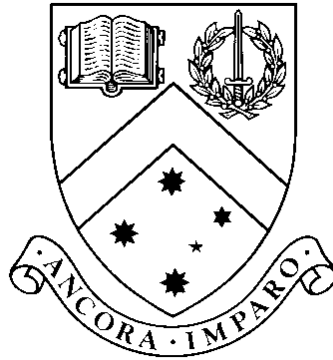


School of Computer Science and Software Engineering
Monash University



Bachelor of Digital Systems (Honours), Clayton Campus
Honours Thesis – 2002

A “smarter” computer controlled model car
November 4, 2002

Supervisors: Dr. Lloyd Allison, Dr. Ronald Pose

Abstract

Autonomous robots have made their impact felt in industry for the past 20–30 years. More recently, the technology has been applied to vehicles in an attempt to create the autonomous car. Although hardware and software has been steadily advancing there have still been very few car projects to achieve complete autonomy and none to achieve commercial viability.

The Monash computer controlled car (CCC), is a radio controlled model car, partly inspired by its larger road-going cousins. It makes use of both off-the-shelf and specially designed hardware to visualise the “race track” and communicate with a host computer that runs the control software for the car.

This project adds new hardware and software components to the existing architecture of the CCC to increase its reliability and performance. Attempts are made to integrate technology from the latest successful autonomous projects in a modular software design. A variety of new algorithms and techniques are adapted and tested for performance and suitability on this platform, with results compared to previous methods. The best results show an improvement in track detection under noise conditions. Some practical limitations uncovered in the hardware are also discussed.

Contents

1	Introduction	1
2	Background & Design	3
2.1	Motivations & Applications	3
2.2	Previous Research	4
2.2.1	Fully Autonomous Vehicles	4
2.2.2	Semi-autonomous Vehicles	7
2.2.3	Previous work at Monash on the car	8
2.3	Initial Project Goals	8
2.4	Achievements	9
2.5	Design Methodology	9
2.5.1	Car Architecture	10
2.5.2	Video Capture Method	14
2.5.3	X Display Method	14
2.5.4	Hardware Speed Detector	15
2.5.5	Track Design	18
2.5.6	Image Analysis Library	22
2.5.7	Edge Detectors	23
2.5.8	CMVision	24
2.5.9	Minimum Message Length	24
2.5.10	Motion Library	26
3	Testing Methodology	28
3.1	Software Architecture	28
3.1.1	Threads	28
3.1.2	Classes and functions	29
3.2	Video Capture Method	29
3.3	X Display Method	29
3.4	Hardware Speed Detector	30
3.5	Track Design	30
3.6	Image Analysis Filters	31
3.7	Edge Detection	31
3.8	CMVision	32
3.9	Minimum Message Length	32
3.10	Motion Library Testing	32

4	Discussion of Results	34
4.1	Architecture & execution speed	34
4.1.1	Single Process	34
4.1.2	Multiple processes	34
4.1.3	Multiple threads	35
4.1.4	Pipelining	35
4.1.5	Summary	36
4.2	Capture frame rate	37
4.2.1	Using the read system call	37
4.2.2	Using memory mapped buffers	37
4.2.3	Summary	37
4.3	X Display Method	38
4.4	Hardware Speed Detector	38
4.4.1	Accuracy & Effectiveness	38
4.4.2	LED output	39
4.4.3	Software integration effectiveness	39
4.5	Image Analysis Software	40
4.5.1	Edge Detectors	40
4.5.2	Filtering	42
4.5.3	Combining Filters	45
4.6	CMVision	47
4.7	Minimum Message Length	47
4.8	Motion Library Testing	48
5	Conclusions	50
5.1	Image Capture	50
5.2	Motion Library	50
5.3	MML	51
5.4	CMVision	51
6	Future Research	52
6.1	Serial adapter	52
6.2	New Model Car	52
6.3	Adaptive edge locker	53
6.4	Speed Detector Hardware	53
6.5	Forward Motion Planning	53
6.6	On board filter hardware	53
6.7	Scaling to multiple processors	53
6.8	RALPH	54
A	Software Notes	55
A.1	README	55
A.2	Application Programming Interface	56
B	Screen Captures	66

C Tables & Figures **67**

C.1 Car Motion instructions 67

C.2 Motion Library Errors 68

C.3 Speed Sensor 68

List of Figures

2.1	Logical path from the PC to the car	10
2.2	Logical path from the car to the PC	10
2.3	Threaded architecture with 2 simultaneous processes	11
2.4	Threaded architecture with 3 simultaneous threads	12
2.5	Threaded architecture with 2 simultaneous threads	13
2.6	The basic building blocks of the software	13
2.7	Design schematic of the 555 timer	18
2.8	Speed detector circuit	19
2.9	Comparison of various surfaces under reflection	20
2.10	Single square right angled turn	20
2.11	Four square right angled turn	21
2.12	2.5 square 45° turn	21
2.13	Image captured in YUV format	25
2.14	RGB Image after segmentation	25
3.1	Connection diagram for speed detector test	30
4.1	Final architecture for software threads	36
4.2	Telemetry information displayed to the screen	39
4.3	LED output Board	40
4.4	Results using an averaging filter	43
4.5	Average image histogram	44
4.6	Results using a 3 bit filter	44
4.7	Results using the monochrome filter	46
4.8	Results testing the CMVision library	48
4.9	MML detector working on a noisy image	49
B.1	Further comparison of surface reflection	66
B.2	Further comparison of surface reflection	66
C.1	Speed detector circuit concept	68

List of Tables

2.1	Comparison of different detector types	16
3.1	List of classes and threads	29
4.1	Comparison of application performance	37
4.2	Comparison of application performance with capture methods . . .	38
4.3	Performance of the differentiator edge detector	41
4.4	Performance of the area weighted edge detector	41
4.5	Performance of the averaging filter	42
4.6	Performance of the 3 bit filter	45
4.7	Performance of the nonlinear filter	46
4.8	Performance of the CMVision library	47
4.9	Performance of the MML detector	48
C.1	Motion instructions accepted by the remote	67
C.2	Measured errors for the motion library	68

Acknowledgments

I would like to thank the following people:

Gary Evans for his extensive advice and assistance with hardware, as well as the use of his lab for construction.

David Arnold for his invaluable technical support and assistance with Linux and PC hardware.

My supervisors, Lloyd Allison and Ronald Pose for their suggestions and motivation, particularly Lloyd for the idea to try MML.

My friends and family for their support and guidance this year.

Dedication

To my fiancée Margaret, for putting up with me working on this.

Chapter 1

Introduction

As the field of robotics has matured over the last few decades, many technological advances discovered have become cheaper and more widely available for application on new robotic projects. An important application for robotic automation to everyday life is the development of intelligent or autonomous vehicles. A great variety of possible uses exist for self-driving or self-navigating vehicles, ranging from passenger to military use. Complete autonomy, or full computer control of these vehicles, is obviously a very useful goal, but one that has proven problematic so far.

The bulk of previous research into autonomous vehicles has focused on individual components useful in an autonomous vehicle architecture, such as speed and distance sensors, intelligent lateral controllers and object detection algorithms. Research from other fields, such as image processing, also applies. The few publicised projects that have made significant progress toward full autonomy have been largely successful, although usually in a constrained environment. To date there has never been a project to achieve complete and robust autonomy in all road environments, however the achievement of this is approaching rapidly.

The Monash Computer Controlled Car (CCC) aims to provide full autonomy, however in a different manner to most larger projects. It is a remote controlled model car, which uses a miniature camera and analog TV transmitter along with custom hardware and software running on a separate PC to visualise and navigate an arbitrary race track. The intent of the car is to model the behaviour of race cars and as such must stay on the race track, avoiding possible collisions, navigating at as high a speed as possible.

Previous configurations of the car have allowed it to follow a track marked by strips of black material. Autonomous navigation around this by the CCC has often been slow and difficult, sometimes resulting in aborted laps of the track due to noise, track conditions, algorithmic complexity and difficulty in making the car respond appropriately to control. This thesis presents the results of research completed this year into extending and enhancing the capabilities of the CCC.

This project attempts firstly to design new software platform and control algorithm, employing a new set of image filters based on those used by recent successful projects. These filters are designed from the ground up to work with the CCC, in order to increase the potential speed around a track. Secondly, the design for

a hardware speed detector is presented. Finally, the effects of investigation into specifying the track by different means, and the implementation of a new motion library to take advantage of this are presented. Other constraints affecting the car are explored to make the features more effective and more useful.

In the following chapters, several facets of this project are re-iterated from the perspectives of design, followed by testing and results. Chapter 2 begins with a summary of previous research into the area of autonomous cars by other groups, as well as deeper background on the motivations of and previous research on the Monash CCC. Following this, the initial configuration of the hardware and software is presented and the design of the new components and details of their aims within the project are fully explained in the latter sections of Chapter 2. Chapter 3 presents the motivations, methods and design of the experiments performed to test aspects of the CCC. The experimentation results are shown and discussed in depth in chapter 4. Chapter 5 contains conclusions and hypotheses arising from the results and chapter 6 recommends future work and research to be conducted on the Monash CCC.

Chapter 2

Background & Design

The following chapter presents the design of this project. Initially, the research context and selected background information on previous research in the field is presented, then the initial project aims and a brief summary of the achievements of this project is shown. This is followed by sections detailing the exact design process of each major component of this project. Specifically relevant background and references are also included in the design of relevant components.

2.1 Motivations & Applications

There are many motivations and applications for autonomous vehicles. Some are discussed and others implied in the following sections outlining this work and the work of others in the field. Generally, the trend is toward producing products of benefit to the lives of the end-user, who is in this case the average driver. Projects tend to work on producing a car that can either drive itself, relieving the end users of the need to navigate their own vehicles, or that can monitor the road situation while the user is driving the vehicle, so that if anything should go wrong, the vehicle is able to take corrective action, or at the very least raise awareness of the problem. Many of these projects will be presented in following sections.

Cars that can do this stand to benefit the community by potentially lowering the costs of road infrastructure and government spending, preventing vehicle-related deaths, increasing driver confidence and decreasing road congestion, both as a result of increased confidence and as a feature of some of the velocity control systems mentioned below. This sort of technology also has ramifications to the automotive sport industry, with the potential to improve safety on the race-track, and even introduce new forms of motor sport.

The Monash CCC is an example of a hobby and sport based toy that has the potential to develop technology in the field for far more serious applications. Although the CCC may never drive on actual highways, compete in motor sports or navigate complex buildings or underground caverns, the vision, navigation and motion technology is essentially the same as most other projects. From this standpoint, the CCC has good potential as an academic testbed for the development of this type of technology.

2.2 Previous Research

This section covers work done on many well known autonomous vehicles. They form a theoretical basis for the work conducted in this thesis and while some are on a much different scale, with impacts in vastly different areas, their existence is significant in contextualising this project. Also covered are projects that do not constitute autonomous vehicles in themselves but, from the perspective of this project, have made a relevant contribution to research in the field.

2.2.1 Fully Autonomous Vehicles

Fully autonomous vehicles, those not requiring human or external input to function, are not commonplace on roads today. In fact, the 100% fully autonomous commercial road vehicle is still some way off in terms of research. These vehicles are ideally able to take care of all their own path planning, low-level navigation, collision avoidance etc., leaving the human in a supervisory role only (for safety purposes). All interaction would be ideally only through very high-level commands to the vehicle. While this is not yet seen in the commercial vehicle industry, it is emerging in scientific research in the laboratory, or uninhabited test fields. Such vehicles have even made it out to road-testing stages for several extended tours of freeways around the world.

Carnegie Mellon University

Carnegie Mellon University's robotics institute¹, lead by Charles Thorpe, has been among the forefront of robotics advancement in the last decade. They have developed and documented a vast number of complete autonomous platforms as testbeds for many autonomous systems developed at Carnegie Mellon. Researchers at the Robotics Institute have been responsible for the Aibo, Millibot (Navarro-Serment, Grabowski, Paredis and Kholsa 1999), Pluto (Martin 1998) and many other useful projects. Of interest to this project is the NAVLAB division, who build robot cars, trucks and busses capable of autonomous driving and also driver assistance.

In particular the "No Hands Across America" tour, which employed the RALPH² computer program (Pomerleau 1995), is of interest to this project. During the tour, two scientists traveled across America in a passenger vehicle fitted out with equipment which controlled the lateral position, allowing them to drive with only their feet to control the speed. RALPH is a program that uses video images of the road surface in front of the vehicle to determine its own position relative to the road and then calculate the appropriate steering direction, keeping the vehicle on the road. RALPH is able to adapt quickly to changes in the appearance of lanes on the road, and the types of lane markings, with little or no a priori model the road surface and features. Pomerleau (1995) showed that RALPH is able to identify and adapt quickly to changes in the road surface type with a high degree of robustness.

¹<http://www.ri.cmu.edu/>

²Rapidly Adapting Lateral Position Handler

In this project, there is a higher degree of control available over the “road” (racetrack) surface than that available in the RALPH trials and therefore a stronger a priori model of the track can be built and exploited to reduce processor overhead. It is on this point that RALPH differs from this project and previous implementations on the Monash CCC (see later sections), as RALPH is built for generality whereas the Monash CCC is aimed at specific conditions (see Tung (2000)). Overall, RALPH is a very useful project that addresses the problem of autonomous vehicle flexibility, but uses very general methods that are costly, in terms of processing time, to implement in this project. The only changes in lane features can be expected to be found during the course of a test on the Monash CCC by our car only under critical lighting conditions. It is expected that with investigation these conditions can be adapted too, or even ignored depending on the robustness of the final algorithm. The lane detection method by hypothesis fitting used by RALPH may be implemented in the Monash CCC later in this project.

Another approach which has implications in autonomous navigation is found in the ALVINN³ (Pomerleau 1992) and MANIAC⁴ (Jochem, Pomerleau and Thorpe 1993) systems, both of which form part of a project to use neural networks, trained by observing real drivers, to develop a model for human-like driving. NAVLAB is also currently applying research from autonomous vehicles to driver assistance and collision warning systems, which may be integrated with existing automotive hardware without a great deal of modification (Thorpe, Duggins, Gowdy, MacLachlan, Mertz, Siegel, Suppe, Wang, and Yata 2002, Wang and Thorpe 2002, Duggins, McNeil, Mertz, Thorpe and Yata 2001). Based on PC hardware available to this project, a neural network could slow the frame rate for the car below acceptable levels and thus will not be explored further by this project.

Universität der Bundeswehr – VaMoRs

In the mid 1990’s, the VaMoRs vehicle (Graefe and Kuhnert 1991, Dickmanns, Behringer, Hildebrandt, Maurer, Thomanek and Schiehlen 1994) was the fastest autonomous vehicle in the world, able to drive at speeds limited only by the vehicle itself. Under strict conditions this 5 ton van has been clocked at up to 130km/h. The restrictions on this vehicle are that it works well on freeways but suffers a massive performance drop on unpainted roads and does not have the ability to drive on roads with other vehicles or non-static obstacles present.

The VaMoRs vehicle uses a single monochrome camera to capture images of the road surface, focusing only on regions of interest (Graefe and Kuhnert 1991, Broggi 1995b). This allows extremely fast processing of the images forming a robust basis for road detection. However this approach was found to be unsuccessful both in critical shadow conditions and when imperfections in the road surface were detected (Kluge and Thorpe 1990), making it ideal for freeway driving but of less use in urban or outback environments.

Since VaMoRs, both the Carnegie Mellon Robotics Institute (Thorpe and Herbert 1997) and other commercial researchers (Franke and Gavrilla 1999, Matthies

³Autonomous Land Vehicle in a Neural Net

⁴Multiple ALVINN Networks in Autonomous Control

2000) have built systems capable of detecting obstacles and other vehicles (and in one case traffic signals – see 2.2.2). The system that will be developed for this project can afford to be much less general – refer also to Bruton (1999) for a discussion on this, and therefore forego much of the processing needed on road detection in favour of doing other important tasks.

Università di Parma – The ARGO

Under the framework of a European project called Prometheus, several vehicles and systems (see following sections) were developed from the late 1980's to the mid 1990's with the goal of researching autonomous vehicle design and hardware implementation. Out of this came the GOLD system (Bertozzi and Broggi 1998) and the ARGO vehicle (Broggi, Bertozzi, Fascioli and Conti 1999b, Broggi, Bertozzi and Fascioli 1999a). The ARGO, using the GOLD system for parallel stereo vision, was taken on a 2000 mile tour of Italy in June of 1998, with excellent results. The “*MileMiglia in Automatica*”⁵ tour saw the ARGO vehicle drive itself on freeways around Italy from Parma to Firenze, through Milano and Bologna at an average speed of 88km/h and a top speed of 123km/h, driving itself over 90% of the time. This tour highlights to us the possibility of autonomous driving (or at least “supervised driving”) in the near future.

The limitation of the ARGO is that the vehicle only drove on flat (of slowly changing) freeways during it's Italian tour. The system in use specifies that flat roads are an assumption for correct operation of the vehicle. This is similar to the VaMoRs vehicle, in that it makes assumptions of road quality specifications. The concept of specifying a minimum road quality is one that will be utilised in this project also. Technologies from Prometheus such as ARGO, GOLD and MOB-LAB have shown proven robustness and accuracy in the field implemented on real working projects. Some of the algorithms are directly applicable to the Monash CCC, and could be implemented given the right host architecture.

MOB-LAB

Another vehicle developed under the Prometheus umbrella was the MOB-LAB vehicle (Broggi 1995a, Broggi 1996). Developed by some of the same people as the ARGO, MOB-LAB was an early attempt at real-time autonomous drive processing. Several interesting algorithms, such as the transform techniques employed to remove perspective from images, came from MOB-LAB and were later seen in work done for the ARGO. Such algorithms might be useful in enhancing the accuracy of the Monash CCC's vision systems, as they have been shown to be robust to shadow problems that other vehicles, such as VaMoRs faced. It may be however, that the low cost massively parallel PAPRICA architecture used by MOB-LAB is not comparable at all to PC hardware in terms of performance on these applications, making it impossible to eventually use these algorithms on the current Intel-based host PCs.

⁵<http://www.argo.ce.unipr.it/ARGO/english/>

2.2.2 Semi-autonomous Vehicles

Semi autonomous vehicles, including supervised semi-autonomous vehicles, individual components and vehicle technology, make up an area most widely used in the commercial automobile industry. Increasing focus in recent years on so-called “intelligent systems” for autonomous vehicles has seen a boom in the research of these systems by vehicle manufacturers. Several key research papers and technologies in the field of autonomous vehicle control have come out of this. Jaguar have developed a radar based system for AICC (Tribe 1996)(see below) and worked with manufacturers such as Porsche, Mercedes and major Japanese car manufacturers on the Prometheus group of projects.

Passenger Vehicles & AICC

Autonomous Intelligent Cruise Control (AICC) is a system that is being developed by many organisations across the world for use in cars in the next decade (Richardson, Ward, Fairclough and Graham 1996). What the system entails is basically an evolutionary step from regular vehicle cruise control. It involves transferring control of both the throttle and brake systems of the vehicle to a computer on board the vehicle, in a similar manner to cruise control. Where the AICC based systems differ is that the computer has also the ability to set the optimum speed of the cruise control, rather than having this set by the driver. This means that the vehicle will travel as per usual cruise control conditions until it encounters other traffic, when the AICC matches the target vehicle speed to that of the surrounding traffic. This is a much more modest and realistic task than full autonomy and has been achieved quite well by several companies (Tribe 1996).

Mercedes-Benz – The VAMP

One step beyond AICC and RALPH is the ability of the car to fully control the throttle and brakes as well as steering, rather than simply one or the other. AICC involves software that sits between the user and the existing functions of most automobiles, doing what the user normally does for themselves, usually with the aid of a radar, while RALPH does the same for steering systems.

The idea of integrating full computer control and urban environment analysis with existing passenger car technology was taken on by some researchers for Daimler-Chrysler (Franke and Gavrilla 1999). Not only did the system developed have autonomous “Intelligent Stop & Go” capabilities but could also recognise and classify a range of traffic signals and other vehicles. This system is by far the most advanced autonomous driving project to date with many promising features. The modified Mercedes-Benz saloon car (named VAMP) has been shown to react to pedestrians and act to avoid collision (as featured on “*Tomorrow’s World*” ©BBC, Autumn 1998). Many of the algorithms are computationally expensive but the system has been tuned well to work in real time on three 200MHz Power PCs running in parallel

The Monash CCC currently operates on a single-host setup where all the software is run on a single processor on one machine. Although parts of the code

are optimised with multi threading (see Tung (2000)) the current hardware configuration limits the PC to a single processor. This means that the parallelism required to run software of the magnitude of the Daimler research is not possible while keeping up the necessary frame rate. The VAMP is however the most fully-featured autonomous vehicle currently available and promises to bring the technology into commercial use very soon.

2.2.3 Previous work at Monash on the car

Computer Controlled Model Car – 1999

Originally, the 1999 CCC analysed 188 pixels of every video frame in order to achieve track detection in $2.8ms$ (Tung 2000). The speed of the vehicle using this method was approximately $0.4m/s$ (Bruton 1999). This was achieved in conjunction with a path planning algorithm and parent / child architecture.

The 1999 project also implemented an optical flow algorithm (Quenot, Pakleza and Kowalewski 1997) for velocity measurement based on the movement of objects from one frame to another. This algorithm only slowed the track identification by a negligible amount, however the resulting resolution was shown to be too low to be useful for the CCC. This algorithm will be replaced with a hardware device in this project.

Computer Controlled Car – 2000

A multi-threaded approach in C++ implemented by Tung (2000) yielded an improvement in track detection speed, using a system dubbed “autoline.” The system was robust enough to allow the car to travel at faster speeds. However, this speed increase was not applicable to cornering and in the worst case the algorithm degenerated to the equivalent of the original algorithm used by Bruton (1999).

Autoline was the final result of the research in 2000 into track detection. The algorithm was built on an intermediate system of what were known as “autonodes.” These autonodes are markers generated by the algorithm and placed automatically on the video image where the algorithm detects a pair of edges which might be the racetrack. Autoline then uses these nodes to calculate certain navigational information about the hypothetical track which had been located.

2.3 Initial Project Goals

Some initial goals at the outset of this project were

- To explore factors influencing the performance of the CCC, both of individual components and the overall system, and to define limiting factors and improvements.
- To increase the navigational speeds and lap times of the CCC
- To test computationally inexpensive image analysis algorithms for the CCC real-time vision systems.

- To design, build and implement a hardware speed detection device.

2.4 Achievements

The major achievements of this project include:

- Increasing the video capture frame rate to above 26 fps using memory mapping and MIT-SHM display.
- Designing a totally new architecture for the CCC control software as an extensible piece of software with a modular architecture with a well defined API that allows different algorithms and routines to be tested on the platform.
- Switching the car software over to use the new Video4Linux 2 API, which replaces the redundant V4L1. This API is cleaner and has better device support and is soon to become the default video capture API for the Linux Kernel.
- Making the software portable across different hardware platforms, analysis techniques and motion command sets with minimal code re-writing required, by separating out the elements where possible.
- Implemented a large number of utility analysis and motion functions in C and C++ (See the API – Appendix A.2).
- Designing and building a custom hardware speed detector for use with an optical encoder wheel.
- Implementing and testing a new set of edge detectors, based on by the work of Tim Bruton in 1999, and testing a different approach to edge detection with a different algorithm.
- Testing a variety of new analysis algorithms, such as fast color segmentation, minimum message length, averaging filters and color thresholding to determine optimal techniques.
- Defining a framework for the “track” that clearly states what the track is physically, allowing a priori information on it to be used effectively.
- Performing some minor modifications to the CCC hardware to allow easy charging and running off a power cable.

2.5 Design Methodology

This section discusses details on the design of components for the CCC. These components include hardware built for the car as well as software developed for the project. Following sections discuss changes to these components that were made during the span of this project. The relationship of components to the goals of the project are discussed and testing methods and results follow in further chapters.

2.5.1 Car Architecture

Physical

The architecture that existed prior to the commencement of this project consisted of the model car transmitting analog TV signals back to a video capture card on the host PC, via an analog transmitter. The PC sent control signals back to the car from the serial port, into a device that interfaced with the RF remote controller. This is shown in Figures 2.1 and 2.2.

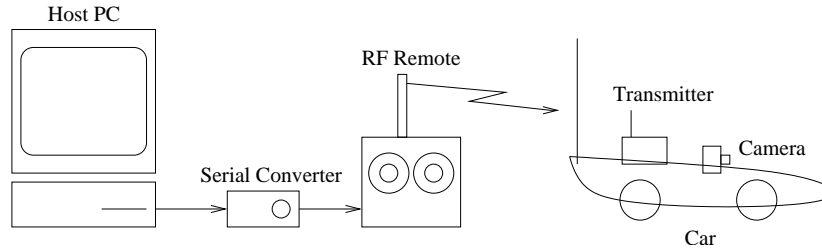


Figure 2.1: Logical path from the PC to the car

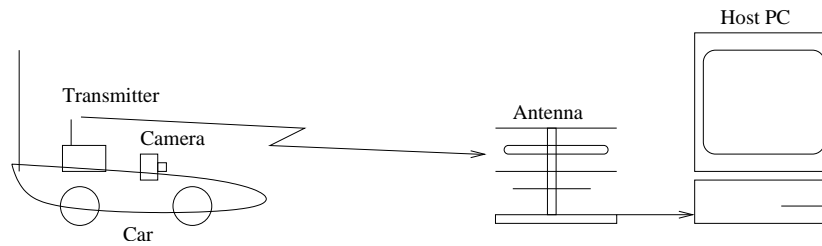


Figure 2.2: Logical path from the car to the PC

Table C.1 lists the ASCII and hexadecimal values of instruction bytes for the serial port adapter, which controls the movements of the CCC by translating these control bytes into voltages on the car's handheld remote.

Software

The software submitted for the CCC in 2000 no longer exists so a new implementation of the software was required before work could commence on this project. The architecture of that software was described by Tung (2000) as a multi-threaded architecture. The use of threads minimises context switching overhead and memory usage significantly over multi-processing using the fork system call (Muys 2000), so the multi-threading strategy was considered for the new implementation. The following pages show the design details of the new implementation.

To run more than one simultaneous set of executing procedures, it is possible to use either the multiple process technique via the fork system call, or multiple threads (lightweight processes) via the POSIX threads interface (Balakrishna

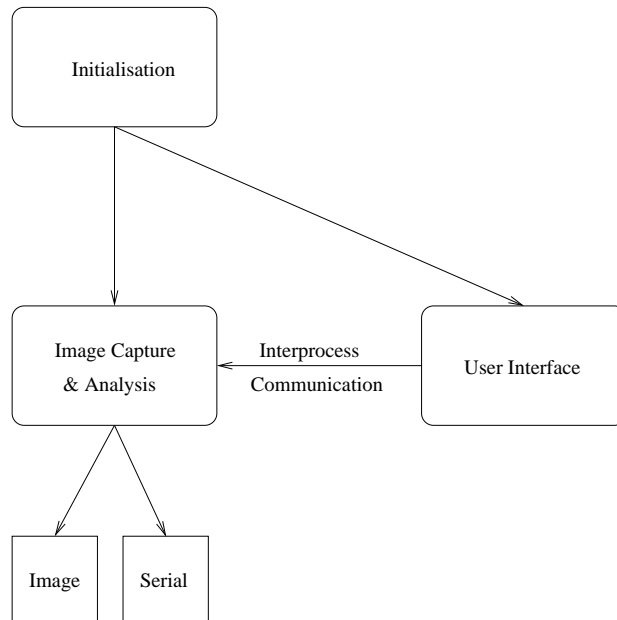


Figure 2.3: Threaded architecture with 2 simultaneous processes

2000). Threads are much more efficient than processes, incurring less overhead and requiring no wasteful IPC process calls (Robbins 2000). They are however slightly more difficult to program, therefore both approaches were tested to verify the results for this application.

The arrangement of the multiple executing blocks of code, or threads, is of paramount importance to the efficiency of the code. It is important that any process which waits for a long time for certain events be implemented as a separate thread, so it can wait in the background while other threads use CPU time. The sample architectures tested use 3 threads, 2 threads, and 2 processes. It was thought that a pipeline might be suited to the task to achieve maximum throughput and parallelism, but this would not be possible to implement correctly, as the problem cannot be broken down into equally time-expensive task blocks. This would mean the need for large buffers or large amounts of synchronisation between blocks resulting in greater overheads, reducing the efficiency of the code (Buschmann, Meunier, Rohnert, Sommerlad and Stal 1996).

Figure 2.3 shows the initial configuration considered for design, after it was determined that a single process would not be adequate. Under this arrangement, two heavyweight processes are created using the UNIX fork command. The first of these runs an infinite loop, capturing video data and processing these frames, before calculating the next required move and sending the output to the CCC via the serial port. The remaining process is a user interface window which waits for input from the user and sends this to the other process to be dealt with.

Figure 2.4 shows the second architecture considered. Being more parallel than the previous architecture it was thought that this would provide greater efficiency and image throughput. This proposed architecture utilised POSIX threads to

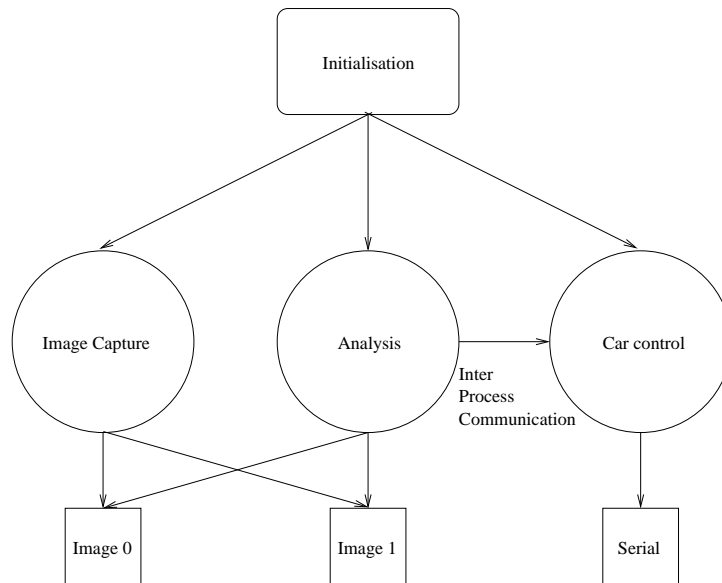


Figure 2.4: Threaded architecture with 3 simultaneous threads

provide multiple concurrent executions. Included in this design was a double buffered capture arrangement, to take advantage of splitting the threads by the data they were to handle. Under this scheme, images in one buffer would be analysed while the next image was being captured and copied into the second buffer.

A final multi-threaded architecture considered was a simple, streamlined architecture. It was designed by breaking the program up into separate processes not based on the data to be handled by each, as in Figure 2.4, but rather on logical boundaries of the types of operations to be performed. The software must first provide a user interface to the program and secondly perform low-level processing. It was decided that the third candidate architecture would be split up on this boundary, with one process to control the input and output window and the serial port, with a second process to control the image processing and decision making. The resultant design is shown in Figure 2.5. Each of these figures show some form of Interprocess Communication, which is implemented differently for each version. Process to process communication uses pipes and thread to thread communication is implemented with shared variables and X Events.

The final implementation was designed so that the main functions of the software (video capture, image analysis, path planning, motion control and profiling) could be replaced as future condition dictate by new functions or classes. A logical diagram of how this process can be seen from a high level perspective is shown in Figure 2.6.

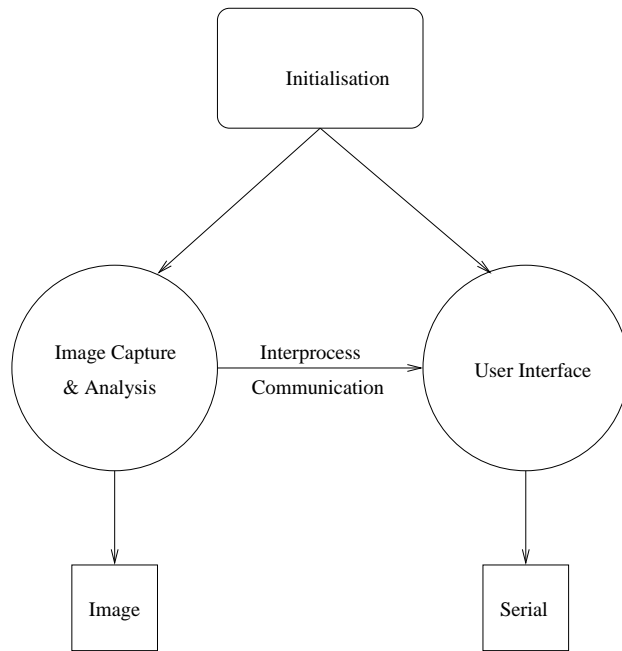


Figure 2.5: Threaded architecture with 2 simultaneous threads

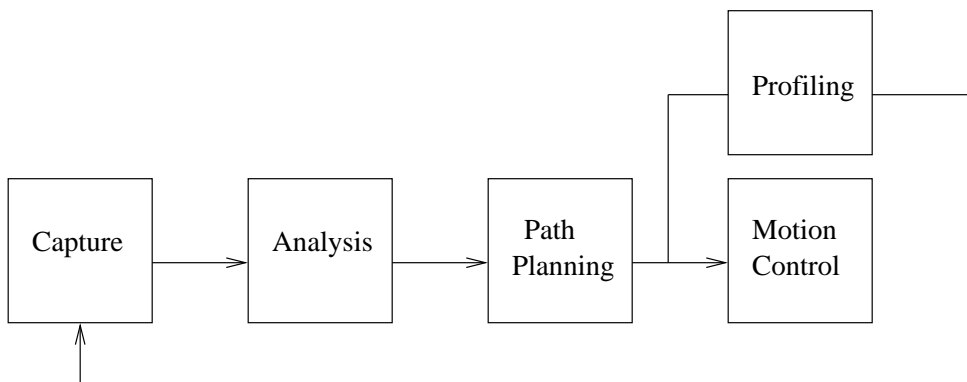


Figure 2.6: The basic building blocks of the software

2.5.2 Video Capture Method

UNIX based operating systems allow several different methods for data to be read from devices or files, which are treated in a similar manner. The most common method is to use the read system call, which reads bytes from a previously opened file descriptor, from the current file location, into a memory variable. The second method uses the mmap system call to map a file or device (in this case a device) to a location in memory. Bytes are then read off the file or device as if they were already in physical memory.

Linux, upon which the operating software for the CCC runs, is an example of a UNIX implementation and therefore also provides these methods to the CCC software. It is therefore possible to either read data from the video card, or map the driver's data buffers (fast RAM buffers) to a memory location and analyse them directly. The advantage of the read call is the simplicity of programming the capture, but it has the disadvantage of requiring extra copying of data through memory. Both methods were tested for speed and the optimum method chosen.

Other manipulations on the video card were done through the Video 4 Linux 2 (V4L2) Application Programming Interface (API). The features provided by the API include the ability to turn on streaming capture, change the channel, select an input and specify the capture format and dimensions. V4L2 is replacing Video 4 Linux (V4L) as the standard video capture API, with a cleaner interface, richer functions and better support for hardware. It will be included standard with the next kernel release. For those reasons, it was decided to port the software over to use V4L2 while it was being designed.

2.5.3 X Display Method

For illustration and debugging purposes, and to prove that the CCC is performing as it is designed, processed video frames must be displayed on the screen of the host PC. As a process that can make intensive use of memory bandwidth, this should be optimised as far as possible. The standard X library provides a function to put an X image into a window, called XPutImage. For this reason, data for display will be stored in X images in order to take advantage of the built in functions. The limitation of this function is that it uses the standard X network protocol to copy data from the server memory to the client memory, which incurs a lot of unnecessary overhead when the client and server are on the same machine.

The MIT Shared memory extension uses System V shared memory and inter-process communication to avoid the overhead of passing images through the X protocol, which provides some speed increases for large amounts of image data (Corbet and Packard 1991). The extension works in the same way as the built in function, transferring the same data types, but using shared memory. It is expected that coding all image displays to use the MIT-SHM extension will increase the frame rate of the program.

2.5.4 Hardware Speed Detector

The issue of knowing the actual velocity of the CCC at any point, as opposed to the current command velocity, has been of some concern since the inception of the car. There are two main reasons for this. Firstly, the car has only four command velocities available to it – stop, go, go faster and go backward. This does not provide a way to accurately specify the velocity at which the car is actually traveling. Also, working with only these alternatives does not give good accuracy for calculating the position the car will finish a move at. Therefore, a way to describe the car’s motion with better resolution was required. This lack of information can be solved in a variety of different ways, of which working with an actual measured speed is the most convenient.

A secondary issue is that the car takes a finite amount of time to respond to commands issued by the host PC. It is not known, nor is it efficient to determine, the time since the last command was issued. From the time of a command being sent from the serial port to the time the car begins to respond is unknown but more importantly there is an even larger delay in the time the car takes to finish increasing or decreasing velocity. Thirdly, it is not known when to apply a braking force. To get the car to drive in a smooth manner, higher speeds and therefore good braking is required. Therefore the actual velocity, due to these factors, will be quite different from the assumed velocity based solely on the previous command.

Measuring the Speed

An attempt to implement a speed detector using an optical flow algorithm (Horn and Schunck 1981) was made early in the development of the CCC, but was found to be unsuccessful (Bruton 1999) due to the speed of the host computer used and the complexity of the algorithm. To avoid any of these problems, it was decided that a hardware speed detector should be built to determine the feasibility of using such a device, firstly to measure speed and secondly to supply the measured speed to the control software in order to improve the path planning.

The design of such a device raises some critical questions. Firstly, the speed must be measured with reasonable accuracy. It has been specified that this should be to within 10% of the actual speed of the car to be of use in planning real-time motion commands. Secondly, the measured speed should be somehow transmitted to the host PC. There are two basic manners for achieving this. One is to display the speed on some sort of visual display device in the field of view of the camera and use image analysis software to read the speed from the video image. The other is to send a pulse of some sort, which is readily available from the output of most speed detectors, to the host PC via the audio channel of the analog transmitter. Some extra software would have to be written in the latter case to extract the frequency of this pulse from the signal and to remove any noise from it.

In terms of physical measurement devices available there are two main classes of device which can be used – magnetic and optical. Magnetic sensors (Gilbert 2001, Law 1998) have been used in ABS systems, fuel systems and car valve timing systems for many years and are a proven technology. These include variable reluctance sensors, Hall effect sensors and magneto-resistive sensors, which elimi-

nate the problem of early variable reluctance type sensors being inaccurate at low speeds (Gilbert 2001).

Optical sensors are also widely used and are available from vendors such as Hewlett-Packard. They are much more accurate than magnetic sensors and their convenience makes them perfect for use with the CCC, in fact Monash already uses them for third year micro-mouse projects. Optical sensors can be fitted to smaller, more confined areas such as the rear differential cavity of the CCC. For these reasons an optical sensor was chosen for the speed detector. Table 2.1 compares the possible choices.

Detector Type	Advantages	Disadvantages
Variable Reluctance	Very cheap, easy to build	Problems at low speed
Magneto-Resistive	Accurate at low speeds	Bulky, costs more
Optical	Cheap, precise and robust	Fragile encoder

Table 2.1: Comparison of different detector types

The Hewlett-Packard HEDS 9000 & HEDS 512 detector/encoder wheel combination was selected as it was widely available for only a few dollars. The only potential problem with it compared to other detectors is the fragility of the encoder wheel. The detection is however very robust to knocking and poor mounting.

All of the candidates for speed encoding technology have a similar encoding method. The output of all sensors is a square-wave pulse with a frequency proportional to the speed of the encoder wheel rotation. This is immediately useful for transmission via the audio channel but cannot be embedded in the video signal without further processing. It was decided that since the infrastructure was already in place for video transmission, a simple thresholding routine could be built to detect the speed off visually-mounted LEDs. Visual embedding was therefore chosen as the method of transmitting the speed signal, by encoding it onto 8 LEDs and thus sending an 8 bit speed number.

Displaying the Speed

The choice to embed the speed data in the video display prescribed the use of an LED scheme, using either a fixed number of LEDs to stand for bits or a number of 7 segment LEDs. As the output did not have to be immediately human readable, the “cheapest” assignment, both in terms of hardware devices and computational complexity, was to assign the output in pure binary to 8 LEDs, visible from the camera.

To encode the variable frequency pulse to an 8 bit number, a translation circuit had to be designed. The design consisted of a counter acting as a pulse accumulator, latched for display and reset at regular intervals. This is shown in figure C.1.

The output of the optical encoder, to be displayed, is a square wave pulse whose period varies proportionally to the number of rotations of the disc. There are 512 slots in the encoder, producing 512 pulses per rotation at a diameter of

2.5cm. This translates to distance traveled as follows:

$$\begin{aligned} \text{pulses/rotation} &= 512 \\ \text{diameter} &= 2.5\text{cm} \end{aligned}$$

$$c = \pi d = 2.5\pi = 7.85\text{cm} \quad (2.1)$$

$$\text{pulses/cm} = \frac{512}{7.85} = 65.19 \quad (2.2)$$

$$\text{cm/pulse} = \frac{7.85}{512} = 0.015 \quad (2.3)$$

Equation 2.2 shows that there are approximately 65 pulses generated per centimeter of motion. It would be useful to be able to read off the number of pulse generated per second, which could be used to efficiently determine the number of cm traveled in the last second.

The design of this circuit uses a 555 timer to generate an evenly spaced clock pulse with a period of one second. A 12 bit binary ripple counter is then used to with the encoder output as the clock input, to count the number of pulses since the last reset, taken from the 555 timer. This same timer signal is used to clock an 8 bit latch chip, storing most of the count value and clearing the counter. The latched data is displayed on 8 LEDs, showing the number of pulses received in the last second as a binary number.

The Latch chosen was a 74LS374 8 bit rising edge triggered flip flop chip. The counter was a 74HC4040 12 stage binary ripple counter with active high reset. Due to the fact that the flip flops had to be triggered at the same time as the counter was cleared, there existed the potential for a race condition to be entered, under which it was indeterminate as to whether the flip flop would latch the actual count, or some rubbish value as the counter was being cleared. To avoid this, which was discovered during testing, a 50ms multiple delay chip was added to the circuit, between the timer output and the counter reset.

The timer signal was designed to clear the counter (active high) and trigger the flip flops in the same signal (low to high transition). This meant that at on any rising edge of the timer output, the count would be cleared and stay cleared until the timer output fell again, upon which the count would resume on each detector pulse. The requirement therefore, was a signal with a low duty cycle with a known length of logic low pulse output. As mentioned, 1 second was the desired period for the count, so the length of time for which the timer output remained logic low should be 1 second. Duty cycle was unimportant, as the flip flops would be latched at the beginning of the logic high “waiting period,” but for low latency it should be as low as possible. The values of R_a , R_b and C in Figure 2.7 were chosen such that R_a was small with respect to R_b , but would be implemented with a trim pot for maximum flexibility. C was chosen arbitrarily as $4.7\mu F$. The frequency was later changed to 5 Hz, as latency was becoming a problem. The derivation for component values is shown below.

$$\text{freq} = \frac{1.44}{(R_a + 2R_b) \times C} \quad (2.4)$$

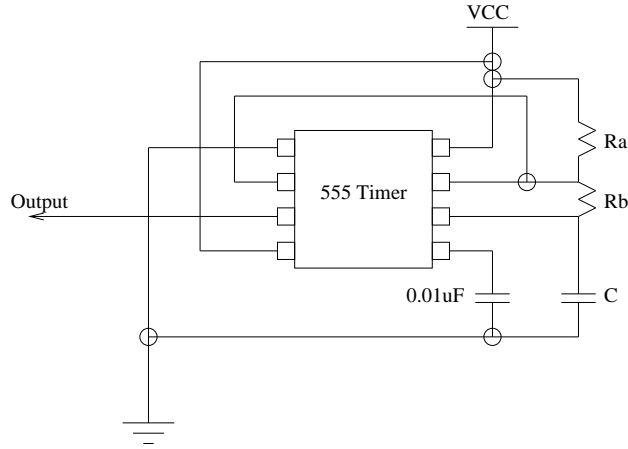


Figure 2.7: Design schematic of the 555 timer

$$5 = \frac{1.44}{(R_a + 2R_b) \times 4.7 \times 10^{-6}} \quad (2.5)$$

$$R_a + 2R_b = \frac{1.44}{5 \times 4.7 \times 10^{-6}} = 61.277k\Omega \quad (2.6)$$

Therefore $R_b = 60.4k\Omega$ and $R_a \approx 1k\Omega$. In order to get the polarity of the signal correct, an inverter was required. See Figure 2.8 for a picture of the actual circuit constructed.

Reading the Speed

A specially designed board with 8 LEDs was designed and mounted on the front of the car. This will display a binary indication of the detected speed. In order to be of any use, this speed must be read into the program. To this end, a reader function was designed to scan through a look up table which was initialised with the expected locations of the LEDs in the video display. Scanning from least significant LED to most significant, the function adds 2^i , where i is the LED number, to a running total of the current speed if the intensity of pixel i is above a threshold value.

2.5.5 Track Design

Surface

A large amount of difficulty in extracting the location of the track from an image lies in the fact that the track is difficult to see. The main cause of this is noise such as reflection of light from the surface. From the position of the car's camera, reflections become extreme. This is shown in Figure 2.9. It can be seen that some surfaces suffer from this more than others due to their reflective nature.

Of the three surfaces shown, it is not the surface with the highest color contrast that makes the white tape strips easier to see, but the surface with the highest

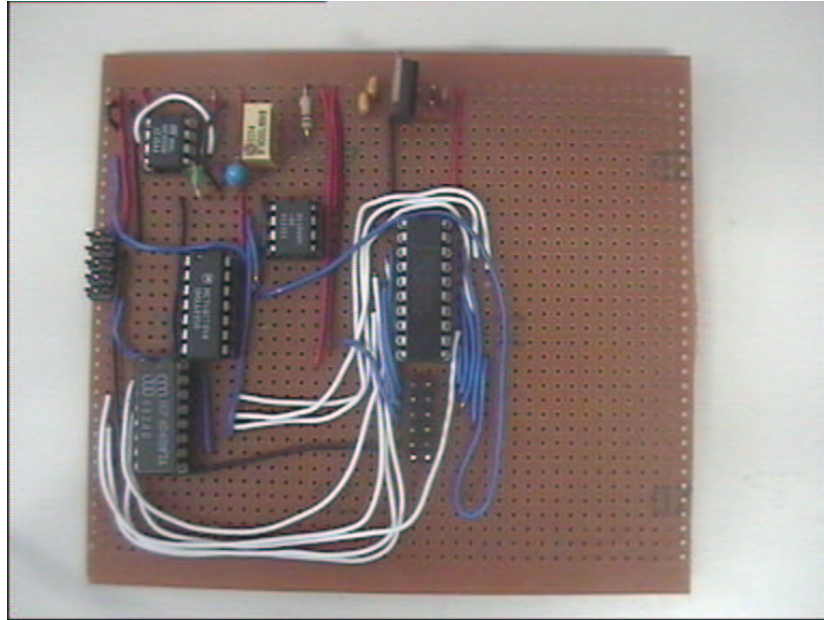


Figure 2.8: A photo of the speed detector circuit constructed, including connectors for a LED output board and power supply.

reflective contrast. This point is very important and is exploited in other stages of the software design for this project. Although the matte green surface shows good reflective contrast, it remains difficult initially to extract the track from this surface. From other angles however, where reflections are not so critical, the black surface is much better in terms of contrast (as shown in Appendix B).

It was decided that black should be used for the track color, in order to achieve maximum color contrast and that reflections should be filtered if possible where they occur. Several kinds of carpet tiles were sourced for this purpose, some being matte black in color, which minimised surface reflections. The edges of these tiles were highlighted with tape as shown in Figure 2.9 to maximise the probability of locating a track edge.

Shape

The CCC is only able to perform a limited subset of the moves that a regular car can perform. It is able to turn the front wheels a set angle to the left or right, or hold them pointing straight ahead. Due to this limitation, it is unreasonable to allow that the track will ever form a line that the CCC cannot physically navigate. Given that the surface would be constructed from carpet tiles, as described above, it is convenient for each section of track to be composed of a limited number of standard shaped pieces.

By testing the minimum turning radius of the car it was determined that the car cannot turn 90° in a 1 by 1 square, as shown in Figure 2.10. Scaling this to the next number of squares, a 2 by 2 square is needed. This is clearly a waste of



Figure 2.9: Comparison of various surfaces under reflection

space and carpet tiles, as the car can turn in a much smaller distance than four squares, as in Figure 2.11.

The most economical solution, using the minimum number of shapes, to this was to cut carpet tiles into squares and triangles to form any arbitrary path, arranged into 45° turns as shown in Figure 2.12. The CCC is guaranteed to be capable of navigating a path made in this fashion.

This has the advantage that the CCC can navigate the track no matter how it is arranged out of these two set pieces and that there are only two different types of pieces to construct, which can be used to create any shape of track out of 45° turns. This also guarantees the width from edge to edge and consistency of track

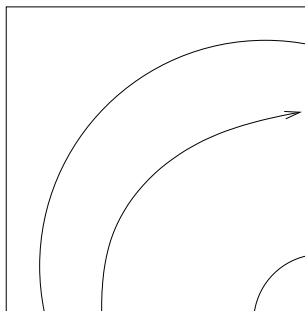


Figure 2.10: A Right angled turn implemented on a single 500mm square. The total area taken up by this turn is 25 cm^2 .

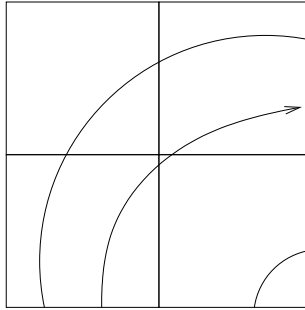


Figure 2.11: A Right angled turn implemented on four 500mm squares. The total area taken up by this turn is one metre².

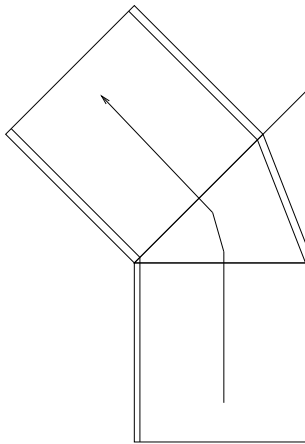


Figure 2.12: A two shape combination of track pieces forming a 45° turn. The total area taken up by this turn is 62.5cm².

surface color and reflectiveness. Previous work on the CCC has assumed some these factors to be fixed, when it was possible for variations to occur contrary to the prior beliefs (Bruton 1999).

2.5.6 Image Analysis Library

Extracting the track from the captured images required the combination of several different techniques. The process of extraction is made difficult by limited reception during movement resulting in transmission noise, strong reflections from the track surface resulting in detection errors and ambiguities introduced by the presence of objects near the track borders. To circumvent these issues in this project, a number of image filters were designed, each targeted at eliminating certain problems with the image as received by the host PC. These filters form the CCC image analysis library.

Random reception noise

In terms of magnitude, reception noise usually results in the greatest problems in the captured images. This is due to the fact that reception noise is random white noise and may occur in any location and at any intensity and hue. It is impossible to filter this noise out totally. Although random noise results in a loss of useful information at the pixel over which it occurs, there is usually still enough information in the image to acquire the track in the presence of noise.

As the ratio of noise to pixels in a frame increases, the probability of detecting the track in that frame decreases. due to this, the filter for reception noise could not be designed to remove the noise, but in some way lessen the effect before the track could be reconstructed. The approaches taken to filtering the images are presented below.

Camera Noise

Another source of noise, albeit softer noise, was the CCD camera itself. The camera has an automatic gain control, called the auto iris, which changes the gain to suit higher and lower lighting conditions. This creates the problem that colors can appear different depending on the amount of light falling on the camera at any given time, which is not easily detectable or controllable. The camera also introduces some quantisation noise, which can be filtered more easily than random noise, but is still a lossy sort of noise that cannot be totally recovered.

Pixel block average filter

Inspired by similar filters produced by Bruton (1999), this takes an image and divides it into a number of sub-image pixel blocks of arbitrary (decided at compile time) width and height. All pixels in the block are assigned the mean value of the block. This algorithm is certainly slower than many of the others, as the mean must be calculated ($O(n)$ for each pixel block) before each pixel can then be assigned the mean value for that block, which also has $O(n)$ time complexity.

3 bit filter

The new “3 bit” filter works by assigning each color channel per pixel either the maximum if the current value is above a threshold or the minimum value if the current value is below that threshold. The total range of values could be expressed optimally in 3 bits, hence the name assigned to this filter. Colorisation of all pixels in a given scan line can be achieved in $O(n)$ time complexity.

Monochrome filter

Similar to the 3 bit filter, the monochrome filter thresholds the pixel value⁶ at 50% of maximum intensity and assigns the pixel either black or white. As above, the process takes $O(n)$ time complexity to run.

Nonlinear Filter

This filter analyses an arbitrary sized sub-image pixel block in a similar manner to the averaging filter. Only the value furthest from the mean is modified for each block. The algorithm assigns the pixel who’s value is furthest from the mean to be equal to the mean value of the pixel block. Distance from the mean is derived in equation 2.7.

$$Distance_{j\mu} = \sum_{i=1}^3 |pixel_j.channel_n - mu.channel_n| \quad (2.7)$$

2.5.7 Edge Detectors

Edge detectors perform the role of locating the position where the track boundary meets the floor. An assumption when using these detection algorithms is that the image has been filtered to an appropriate standard prior to edge detection. This eliminates the need for repeating processing unnecessarily.

Differentiator

The differentiating edge detector works by scanning one line of pixels and calculating the difference from $pixel_i$ to $pixel_{i+1}$. The greatest difference (above a certain threshold) is considered to be the best edge, that is the pixel that gives the strongest indication of the presence of an edge at that point. This code was used effectively by Bruton (1999). The time complexity of this algorithm is $O(n)$. The “strongest” edge E_{max} in an array P of n pixels is found by:

$$E_{max} = \max_{i=0}^n (|P_i - P_{i+1}|) \quad (2.8)$$

⁶The pixel value is defined as the mean of the red, green and blue channel values for that pixel.

Area weighted detector

As an alternative to the proven differentiation technique (see previous section), it was proposed that perhaps a different criterion be selected for the purposes of detecting the track edges. Some prior knowledge, or at least expectation, of the track is held before any detection is performed. It is known that the track will be a certain width, most likely several orders of magnitude greater than any other objects on screen. The area weighted detector was developed to exploit this fact, it works on the principle that the track edge is most likely to be the edge bounding the largest single area of consistent color. The algorithm scans a single line of pixels horizontally, calculating the difference as it goes, much like the differentiator. The number of pixels since the last detected edge is then compared to the previous set of edge and distance. The point at which an edge occurs at the largest distance from any other edge is deemed to be the track edge.

2.5.8 CMVision

CMVision⁷ (Color Machine Vision) (Bruce, Balch and Veloso 2000) is a product of Carnegie Mellon's CORAL group and their research into fast image segmentation for mobile robots. Using a YUV color space image capture, it is possible to separate out chrominance (color) of a pixel from luminance (brightness). CMVision works by thresholding planes in the YUV color space to classify the pixels in an image as part of zero or more thresholded color bands. These pixels are then grouped by a fast segmentation algorithm and colored as a group.

The CMVision system is currently a freeware software library, used by the Carnegie Mellon Multi Robot lab with success on their Sony AIBO and soccer robot teams in the robot cup challenge.

It was thought that this technique would be useful in classifying pixels in real time from the video output off the CCC. Integrating this into the project required images to be captured in YUV format and use CMVision both to segment the image by color and translate the output to RGB color space, where the image analysis library could then be used to further detect objects in the image. Potential improvements from adding CMVision to the software would come from the improved speed of both YUV capture (as less data is copied through memory) and CMVision itself. The fact that the image would already be processed to some degree meant that there would be less need for processing by the image library routines, which were anticipated to be somewhat slower.

2.5.9 Minimum Message Length

Minimum Message Length (MML) (Wallace and Boulton 1968, Wallace and Freeman 1987) is an information-theoretic inference technique that can be applied to the problem of extracting the track from a noisy image. This application of MML works in theory by regarding a statistical model of the track as a series of regions along any particular scan line of an image of the track, each with a mean pixel value and standard deviation for the pixels from that mean, in the given region.

⁷<http://www.cs.cmu.edu/~jbruce/cmvision>

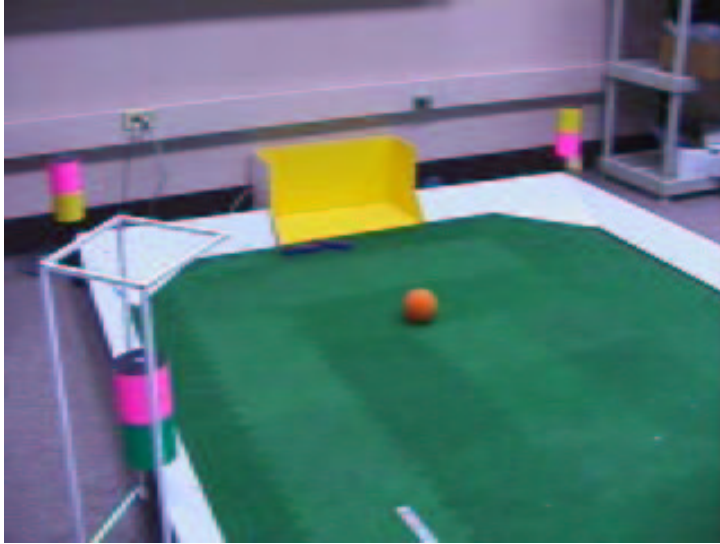


Figure 2.13: An image of a robot soccer arena captured in YUV format. (source: CMVision website)



Figure 2.14: RGB output of the image in Figure 2.13 after segmentation by CMVision. (source: CMVision website)

Scan lines are encoded and costed in natural bits (nits) as a two part message consisting of a given hypothesis on the distribution of regions along a scan line, and the encoding of those pixels given the hypothesis.

The MML cost for a given message⁸ is derived as per equations 2.9 and 2.10, where μ and σ are values calculated from the image. The message length is given in natural bits, or nits.

$$\sum_{i=1}^n -\log_e(\epsilon \times f(X_i)) \quad (2.9)$$

$$= \sum_{i=1}^n -\log_e\left(\frac{\epsilon}{(\sqrt{2\pi}\sigma)}\right) + \frac{1}{2}\left|\frac{X_i - \mu}{\sigma}\right|^2 \quad (2.10)$$

In order for the MML method to be fully inter-operable with techniques implemented in RGB, three separate values for μ and σ would be inferred, one for each color channel per segment.

The MML encoding of any given hypothesis on the track layout which produces the minimum cost, or message length, is treated as the most likely model given the data. This is strictly speaking not true MML, as there is no cost given to the hypothesis, but it can safely be ignored as it will usually be a constant – the cost of encoding the number segments, usually 3. Under this system it would be advisable to follow the most likely path. It was not known to what degree this system will work under high levels of noise, nor how it would interact with other filters in the image analysis library. These aspects were tested and results are presented in following chapters.

2.5.10 Motion Library

In order for the CCC to execute motion instructions once they are determined from image analysis, the control software must send binary information to the car via the serial port. To make this simpler, an abstraction layer was designed to handle the creation of correct binary data for any given navigational situation. To achieve this, the number of situations was first limited by exactly specifying the track, as described in section 2.5.5.

A library of commands was then constructed to produce a series of known moves where the starting and finishing configuration (heading and position) of the car was well defined. Following the paradigm used in section 2.5.5, where the track is thought of as the possibly infinite linear combination of a limited number of set sections, the motion commands to the car can also be thought of in this way. When one of the set of track pieces was detected, the software could respond by queuing one of a number of previously defined responses to navigate the path.

Motion commands were added to a queue and dispatched in a FIFO manner. The period of time between subsequent additions of new commands to the motion queue was known as the “event cycle.” Each move or “event” may be added to the

⁸The term “message” will be used from here in to mean an hypothesis of the distribution of color regions on the track and the encoding of that hypothesis. “Message length” will be used to mean the MML cost of a message

queue at any time, as the controller decides that the car needs to perform a certain move. Each event is made up of zero or more individual instructions to be sent at a fixed rate from the serial port. This rate is known as the “instruction cycle.” More high level moves can be queued more quickly by shortening the periodic event cycle, which leaves less time for path planning. This also creates a backlog of instructions which can only be dispatched by shortening the instruction cycle, which sends instructions to the serial port more rapidly.

The performance of the car was then tested against a dynamic implementation of the motion library, where the motion routines generate moves to keep the car inside the track where possible, by using lower level movements that can adapt to the errors that may occur between the expected configuration of the car at the end of a “move” and the actual configuration. This method discards the set move only paradigm, introducing complexity into the design with the potential advantage of better dynamic response. Previous projects have implemented the motion routine like this as a state machine (Bruton 1999), which responded to set situations as they occurred, allowing for errors of a range of magnitudes to occur.

The first proposed routine required that the pre-conditions and post-conditions, that is, the configurations, be well known for each “move.” A move is defined as a sequence of motion instructions forming a navigation around a known obstacle, for example, a corner. Using a queue to store command sequences, a set of moves was designed to implement a 45° turn left or right, and a single square of forward motion. Technically, these moves could be combined to provide adequate motion around any track formed from the tiles described in Section 2.5.5. High speed navigation however requires that these moves be combined in a manner such that there is no pause between moves. Therefore a variety of racing turns were added to the library. Appendix C.2 shows the commands which were queued to achieve the moves.

The length of the instruction cycle is also a factor in the shape of the move to be performed. Experimentation on different timings showed that the optimal timing was close to $250000\mu s$, otherwise individual instructions tended to begin running over previous instructions that had just left the serial port. The full effect of overrun instructions was manifested in the car’s movement, to a point where the car would not move at all if every instruction was overrun by the following instruction. If the timing was slower than $750000\mu s$, instructions were executed with pauses between them, slowing the overall driving speed of the car and modifying the shape of turns. At slower speeds, the car is less able to navigate tighter turns and is not as smooth. At higher speeds however, the car becomes unstable and uncontrollable, as latency introduced by a long event cycle, relative to the distance moved each frame, becomes significant.

Chapter 3

Testing Methodology

This chapter discusses the experiments that were performed on the CCC. It covers the main aims and motivations of each test and the conditions under which they were performed. Each piece of software and hardware was tested individually where possible in order to isolate the effects they had on the project as a whole, as well as the factors that determine their individual performance.

3.1 Software Architecture

The first tests were performed on the base software architecture itself. Most of the software modules for analysis and motion were later developed as add-on functions called from parts of the base software.

3.1.1 Threads

In order to handle input events in an efficient manner whilst also controlling the car, it was determined that the software should have multiple simultaneous execution threads, as shown in the results obtained by Tung (2000), so that the ongoing processes would be able to continue while the software waited for user input. Several candidate architectures for software were designed during the early implementation phase of the project. These were tested with the code that had been created at that time, which was expected to be representative of the tasks the CCC would perform when completed.

The test was to compare the number of times the main capture loop could be executed within each architecture by inserting performance profiling code into the main loop. Whichever architecture produced the best results would be considered most suited to this task. Performance profiling information was obtained using the GTop¹ program, and also by inserting a timer into the main loops of the control code. The timer class used was previously developed by Daniel Tung for the CCC.

The output speed of the CCC code is measured in frames per second (fps), as each loop represents the ability to complete the analysis and display of one full frame of video. The CCC code was also monitored for memory usage and CPU

¹©1998 the Free Software Foundation, www.gnu.org

utilisation, which are also factors relating to the performance of the code. This was done by running subsequent versions of the code and logging the profiling output in frames per second created by the timers. The GTop output was also observed and the memory sizes and CPU usage noted.

3.1.2 Classes and functions

The following is a summary, for reference, of main classes and functions used to implement the CCC components. All function calls and processing was done from within these three threads. See Appendices A.1 and A.2 for a full explanation of the use of the classes within the project.

Name	Use
control_loop	The main thread which listens for user input
capture_loop	The second thread which runs the capture and analysis
serial_out	A temporary thread created when data must be sent
video_card	A class encapsulating the video card functions
motionQ	A class that implements a linked list queue

Table 3.1: List of classes and threads

3.2 Video Capture Method

The overall performance of the car can be measured in terms of achieved lap times. The speed that is possible is mostly limited by the running time of the main control loop and the efficiency of what it produces. A large amount of time in this loop is taken up by the task of capturing a video frame to memory. This must therefore be as fast as possible to allow the resulting analysis time to be of a high standard.

In order to test which of the methods of reading data (read and mmap) from the video capture card allows for the fastest capture of data, both the read and memory mapping techniques were coded and tested using inline profiling with the timer class. The timer class was designed for this by Tung (2000). The video capture code runs in an infinite loop, using the timer to determine how long it has been running for and incrementing a counter each time a video frame is captured. At the end of execution the average number of frames per second is calculated. The method which captures the highest number of frames per second is the most suited to this task in terms of speed.

3.3 X Display Method

In order to achieve the fastest possible navigation speed, the time spent displaying results to the screen must also be kept to a minimum. The speed with which the various methods enable images to be drawn to the screen is measured by the number of frames per second seen on the PC. This is once again determined by

inline profiling information embedded into the software. Both methods, with and without the MIT-SHM X extension, were tested. It was anticipated that the shared memory method, using MIT-SHM would be faster, as it was designed to provide high bandwidth and efficient image display.

3.4 Hardware Speed Detector

The speed detector was designed to display the number of pulses generated by an optical encoder mounted on the car as an 8 bit number output to an array of 8 red LEDs. The test for this was done in two stages. Firstly, the circuit was connected to a function generator, as shown in Figure 3.1. The function generator was used to generate varying frequencies of square waves which were used as input to the translation circuit. The output was monitored on the LED display board created for the CCC, to ensure that the output was consistent with the input.

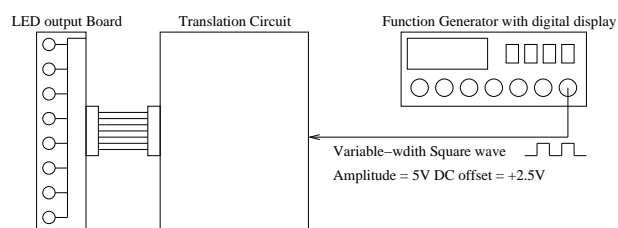


Figure 3.1: A Connection diagram showing the configuration used to test the speed detector translation circuit. The function generator is shown as used to simulate an optical encoder wheel output.

The second stage test was to replace the function generator with a real optical encoder wheel and feed the output directly into the translator to verify the results obtained using a function generator.

3.5 Track Design

Extensive testing was carried out to determine the optimal shape and materials to be used in the construction of the track. Most of these tests were somewhat ad-hoc, compared to the more methodical approaches taken to testing other components of the system. Due to this, the results of testing, as mentioned in Chapter 2, were simply that the track was constructed from black-backed carpet tiles cut into either 500mm squares or 500mm right angled triangles. Various observations on the performance of the filter algorithms suggested that the track edges should be outlined in white masking tape, to guarantee contrast even in low light conditions, although it was found that some of the latter algorithms do not make use of this feature. As no formal testing was performed, and these results are assumed in previous sections, there is no mention of track design in Chapter 4.

3.6 Image Analysis Filters

The CCC image analysis library is a simple library of image processing routines built especially to perform image analysis functions necessary to the operation of the CCC. The library functions are broken into two parts – image filtering routines and edge detection routines. The first collection of functions modify the image in memory in some way, returning no values, while the second set return positions of edges found in the image.

Most image filters have no directly quantifiable results. It is intuitive to see which filters work well, according to their specifications and video output, but a far more useful test in this application is to determine their effect on the detection of the track by the use of the edge detectors. The results shown in Section 4.5.2 compare the effects of various filters on the results obtained from the edge detectors to those resulting from the use of detectors on unfiltered images. This will give a metric for determining the “best” image filter or combination of filters under different conditions.

The filters are tested, for simplicity, with the simple differentiator edge detection scheme, without the use of the edge locker. Three lighting conditions are tested, direct bright light, very low light and soft ambient light with no direct reflections or shadows. They are tested facing parallel to the track and at an angle, such as in a corner.

3.7 Edge Detection

Track detection is the most vital step in making the CCC work. Edge detection is a useful means toward this end. Two forms of edge detection were tested for their reliability. Designing a perfect edge detector is not possible under all conditions, as the track can become totally lost in noise at times. Therefore a good edge detector should be able to detect the correct track edge most of the time under good conditions and decay gracefully as noise begins to obscure the track, with some robustness to noise. The design of the CCC software allows for images to be filtered before the edges are detected, so it is assumed that the detectors need not be “intelligent,” as the images will have already undergone filtering and analysis to remove some noise.

The testing procedure for both methods was the same. The CCC was held stationary, relative to the track, whose edge were at a known location. The lighting conditions and obstacles on and around the track were modified to test each method’s resistance to lighting changes, reflective highlights, peripheral obstacles and shadow. The detected edge location for each frame was logged to a file, which was compared to the known location of the edges. The result is expressed as a percentage of frames detected correctly over the total number:

$$DetectionAccuracy = \frac{F_{good}}{F_{total}} \times 100 \quad (3.1)$$

Where F_{good} is the number of frames where $E_{detected} = E_{true}$ for all edges in the frame. Frames where only one of the two edges are detected correctly are

counted as $\frac{1}{2}$ frame. This same test was recycled for the testing of image filters, as the results of the detectors, especially the differentiator, are assumed to be affected only by the quality of the image. The quality of the image is a result of the effectiveness of any filters applied to it and therefore the performance of an edge detector is a measure of the goodness of a filtered image.

3.8 CMVision

Colors found in images processed with the CMVision library will be thresholded in order to isolate segments of colors expected to appear in an image. In the case of the lab where most work was performed, the colors expected are white, black, pink and gray. Most areas of navigational significance can be expressed by these colors. The test of the effectiveness of the CMVision image filter was the number of pixels in an image that are correctly classified in one of the main color segments expected in the image from YUV color to RGB color. It was expected that a minimum number of pixels would be incorrectly classified due to excessive noise. All other pixels that are not classified are colored black and used as “background” pixels.

3.9 Minimum Message Length

It was expected that, due to the number of pixels influencing the fit of an MML segmentation, this method would be less prone to noise. The testing was therefore performed over what was considered a noisy environment, in which the edge detectors were known to have difficulty. Quantitative information of the performance was not highly required, as it would be obvious as to whether the segmentation was better than detection performed on the edges. Due to processing constraints on the CPU, the MML segmentation would be of no advantage to the real-time processing if improved results were not obvious to the human eye.

3.10 Motion Library Testing

It was necessary to test the proposed new motion library, which used the “set moves only” paradigm (see Section 2.5.10), against the traditional dynamic implementations (Bruton 1999, Tung 2000) to see if the paradigm gave any advantage in the motion library.

It was expected that the shortfall of the new model would be accumulation of small errors between the assumed position at the beginning and end of each move, and the actual positions on the track. A series of tracks were set up to determine the effect of those errors and whether they could be controlled or whether they were indeed irrecoverable over time.

A series of complete moves, including 45° turns and straight ahead segments, describing each track was sent to the car. The difference between the expected position and final position on the track was measured. It was not known whether this accumulated error would be critical or negligible. Critical was defined as a

condition that would cause the car to stray from the track in an irrecoverable manner at any point over the average length of a track, which was set at 50 carpet tiles in length.

This test was performed “blind,” that is with no assistance from the CCC visual routines. The reasoning for this was that if the outcome was at least satisfactory, then the visual routines will aid the driving speed and accuracy, to compensate for the fact that the shape of the track is not known in this fashion during a normal run.

Chapter 4

Discussion of Results

This chapter contains the results of tests, programs and experiments defined previously and discusses their significance to the project with regard to the goals and previous work in the field where applicable.

4.1 Architecture & execution speed

The following outlines the changes in execution speed noted between implementations of the program using differing architectures. These results may be also influenced by the state of the project software at the time of the measurement. That is, the amount of analysis and the nature of work performed inside the main loops. Every effort has been made to ensure consistency between results but there may be slight variations of the order of 5% of the loop speed due to the varying implantation's of different modules as time progressed. Major variations are noted and explained below.

4.1.1 Single Process

It was found with a traditional single process the program took a long time to execute. This result is included only for completeness, as the program was performing no analysis at the stage where it was implemented as a single process. The main result realised from this implementation was that the program was not at all adequate for the task of controlling the car, as it was also required to take user input and process that with low latency. A single process can only achieve this by allocating time every loop to deal with queued user requests and must have a very tight loop so that the user is not kept waiting. This is very wasteful as this time is allocated every loop in a fixed position so that it is wasting time when there are no requests and performing slowly when there are many requests.

4.1.2 Multiple processes

Using the fork system call to create a two process implementation of the program gave two significant advantages over a single process. Firstly, the code was able to respond instantly to user input, due to a parent process that continuously waited

for user input and processed it, and secondly, a child process was able to wait for captured images to come in from the video capture card without taking processing time away from the I/O tasks. This gave an overall speed increase. The speed increase noted was due to the fact that the original code waited for long periods of time until the video capture card returned a frame, time which in the two process version is used to process user I/O tasks.

4.1.3 Multiple threads

Although the results of using multiple processes were encouraging, threads have many distinct advantages of processes. Implementing the code using POSIX threads instead of processes gave a small speed up again, however this was less than one frame per second. A further advantage of this implementation (seen in the final code) was the ability of threads to share address space and variable data. It was found, as the project developed, that none of the tested implementations were entirely sufficient to process information in a timely manner and dispatch user requests correctly. In fact, a point was reached for each of the architectures in turn where they no longer responded to user events. This was not a function of the CPU speed, as utilisation was well below 100%. The bottleneck lay in the use of X Events for motion control.

User input triggers the control loop via a mechanism called an X Event. The control loop listens for X Events and processes them, based on the event type. The trigger for the control loop to switch from listening for events to processing output data to the serial port was also an X Event, of the type *X Client Message Event*, rather than the *X Keyboard Event* automatically sent when a keyboard button is pressed. These events queue up in the same internal X buffer at the rate of 25 per second (the rate of video frame processing) plus any extra keyboard events. However, serial data can only be sent to the car at the rate of 4 characters per second, otherwise the data is overrun before the car can respond fully. As each event had an average of 4 serial instructions per second, this created a backlog of 21 events per second to be processed. 21 events translates to $21 \times 4 = 84$ characters per second that could not be sent out of the serial port, with user input processing via the keyboard additional to this.

A refinement to the methods originally tested was to create a temporary thread to output characters to the serial port, while the next event was caught and processed (See Figure 4.1. If the next event was user input, it could be dealt with in a timely manner, otherwise the next client message would continue to queue up. The only complete solution was then to slow down the rate at which events were added to the queue. The final solution involved having a slow event cycle, which was the period between successive client messages being queued. Frames where no message was sent were still used to process path planning for the next event.

4.1.4 Pipelining

An arrangement of the threads was planned such that they formed a pipeline, where several threads worked in series on a single iteration of the program, but

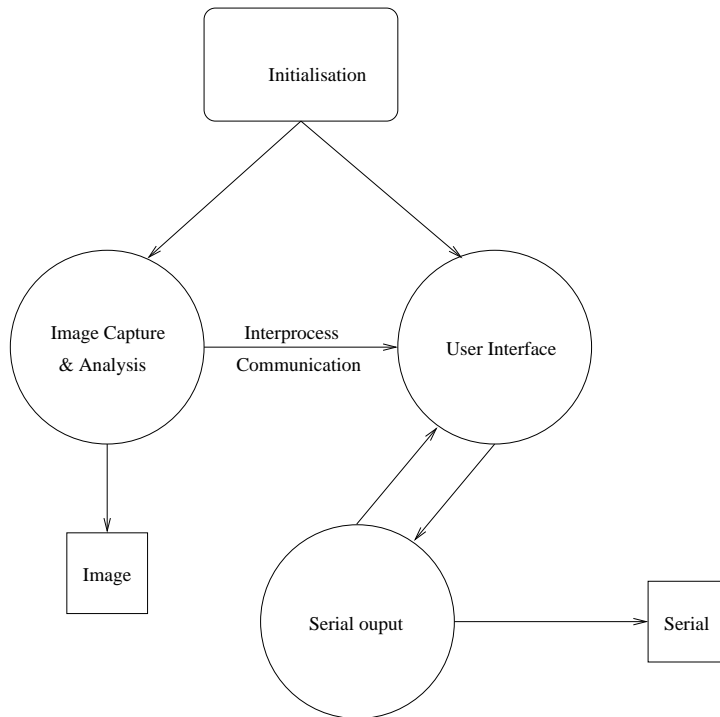


Figure 4.1: Final architecture for software threads. This is a refinement of Figures 2.5 and 2.4. Note that the serial output thread here does not always run and can be killed to make way for a new thread instead of waiting for termination.

each thread began the next frame while its successor was completing the current frame. This has the advantage of being as parallel as possible. The plan was to create four separate threads, each sharing the variables for up to four frames at a time, passing the results along the pipeline while starting the next frame. Unfortunately, further research showed that this technique is only applicable to tasks where each stage takes the same amount of time – which could not be guaranteed here. This method was abandoned because it would mean that the required synchronisation and buffering would degrade performance.

4.1.5 Summary

In summary, the following table shows the speeds achieved with the above architectures. It can clearly be seen that switching to a multi-threaded architecture gave the best performance. Heavyweight processes gave similar mean frame rates, but used more memory. It is logical to assume that the nature of the task is somewhat responsible for this behaviour, as the program can be most logically divided between user I/O tasks and internal processing, which is what the concurrent execution models take advantage of.

	Mean capture speed	CPU Usage	Memory Usage
Single process	6 fps	95%	N/A
Multiple process	14 fps	61%	9.4MB
Multiple threads	14 fps	53%	8.5MB

Table 4.1: Comparison of application performance. These figures were obtained at the stage where video capture was implemented using the read system call. All figures are approximations, as memory and CPU usage varies over time. Memory usage for single process architecture was not recorded.

4.2 Capture frame rate

4.2.1 Using the read system call

The results of capturing video frames using the read systems call to read the bytes for one frame from the device were poor. In the worst case the rate was constant at 6 fps, which meant that each frame took an average time of 0.16 seconds to capture. This is not enough for robust path planning to occur as, even if only one motion instruction were to be executed per frame of analysis, the car might move as far as 20 or 30 cm before the next frame is analysed. It is unlikely that this would produce fast lap times, as the car would need to be kept to slow speeds to guarantee that it would finish a move inside the track. Once the car loses the track it is very difficult to re-acquire it, as there is no information in the visual signal to indicate where it might be.

4.2.2 Using memory mapped buffers

Implementing memory mapped buffers produced a vast improvement in the rate of video capture. Using the same profiling routines, frame rates of over 26 fps were recorded when the software was performing no other analysis. Even with analysis turned on, frame rates were rarely observed below 24 – 25 fps, indicating that the video capture is potentially the most time consuming operation that the car performs. All other operations, such as analysis take up around 1 – 2 fps while image capture can provide bottlenecks of the order of 80 % of the processing speed, based on these figures. Tung (2000) reported that a similar application performed at 100 frames per second with no video capture enabled. This indicates that the actual figure might be closer to 25 % for some applications, which is still a significant part of the execution time.

4.2.3 Summary

The double buffered arrangement presented in Chapter 2 did not give any speed increase as the video capture, using memory mapped buffers, already uses parallel capture buffers. On the test PC, there were up to four buffers in use at any time. Evidently, memory mapping has a vast advantage in video capture applications.

	Mean capture speed
Read	14 fps
Memory Map	26 fps

Table 4.2: Comparison of application performance by changing the method used to capture video. All figures are approximations, as capture speed varies over time.

4.3 X Display Method

Using the XShmPutImage method provided by the MIT-SHM X extension to display images on the screen produced a mean frame rate increase of 2 fps, rounded down to the nearest whole frame, over 3 tests. It has been claimed (Tung 2000) that the shared memory extension provides possible frame rates of up to 100 fps on this application, where no significant other work is being performed, such as image capture. Compare this to a nominal frame rate of around 60 fps without the extension and a larger increase should have been expected. This indicates that the display to screen is not the limiting factor and that the frame rate is limited by the rate at which these frames can be created (i.e. captured), rather than displayed to the screen.

4.4 Hardware Speed Detector

4.4.1 Accuracy & Effectiveness

Initially the clock pulse was designed to latch the current count and reset the counter for a new count period once every second, to produce an output in pulses per second. This output maps directly into metres per second. This was found to be unresponsive to rapid changes in the speed of the car, as the display was only able to update as it was latched, once every second. In one second, the car can move large distances and change speeds by up to one meter per second. As the path planning routines worked optimally at around 20 times a second, this meant that most of the readings taken could be significantly in error.

The clock pulse was shortened 0.2 seconds and a look up table added to the software to reference the speed, using the 8 bit output of the speed sensor as an index to a 256 position velocity table, mapping the 8 bit number to a fixed speed. This means that 15 out of 20 frames could now be in error, but only by $\frac{1}{5}$ of the amount previously.

With the look up table it is possible to calibrate the software to read the detector circuit differently if, for example, the encoder disc is substituted for another model. The accuracy of the circuit is well within the 10% limit specified in the design, this is however only approximated in the look up table by the function $speed = 20S$, where S is the number read from the LEDs. This provides a close approximation of the output of the circuit and is faster to produce. Due to the smoothing effects of the function it is possible that it will help reduce any drift

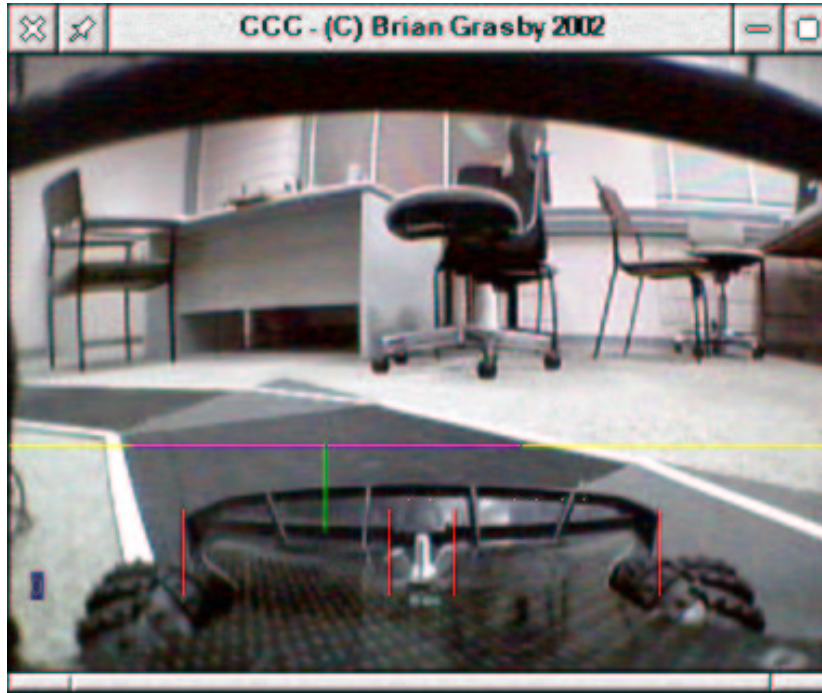


Figure 4.2: Telemetry information displayed to the screen. The “0” in the bottom left corner displays the currently detected speed. As the LED board is not mounted in this picture, the speed is detected against the black front lip as zero.

that may occur in the circuit.

4.4.2 LED output

A simple threshold was found to be sufficient to determine the difference between any LED being on or off. The difference was sufficient not to be affected by lighting conditions over the output board. The output was read into a memory variable and written to the screen. This is shown in Figure 4.2. The LED output board is shown in Figure 4.3

4.4.3 Software integration effectiveness

Due to time constraints, no integration was performed in software for the speed detector. The only software features that were added to support this device in the future were the reading of and display of telemetry data from the LED output board to the screen. See Section 6.4 for recommendations of future work that should be done in this area.

4.5 Image Analysis Software

A large array of image analysis software, described in Chapter 2, was designed and implemented for this project. This section presents the results of testing performed, as described in Chapter 3, on the image analysis library routines.

4.5.1 Edge Detectors

This section presents the results of testing the edge detectors developed for use with this project.

Simple Differentiator

The simple differentiator was able to robustly detect edges, given the appropriate lighting conditions. The CCC has previously worked under the assumption that lighting conditions were correct (Tung 2000). This is due to the automatic gain control iris featured on the video camera, which is not under the control of the software and can act to the detriment of the project in conditions where lots of light reflects at the car.

The main situation that caused problems was when multiple edges occurred in a scan region which were of greater magnitude than the track edge. Although this is a relatively rare occurrence, it is possible to create this situation when lighting conditions are critical. Other conditions that caused the edge detectors to stray were when obstacles such as table legs and cables, which create good contrast with the floor, were detected outside the track. Table 4.3 shows the result of testing the detection accuracy of this edge detector. Test four is an example of what can happen if the detector picks up a strong edge outside the track.

Overall, it was very difficult to create conditions under which the simple differentiator did not work reliably. Although it was reasonable to work under the assumption that lighting conditions will always be optimal, as this assumption simply carries over from previous years, a design goal of this project was to create

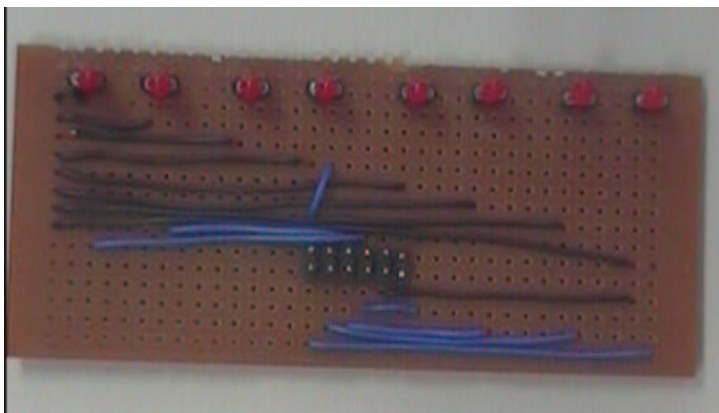


Figure 4.3: LED output Board

Test number	Frames captured	Frames detected correctly
1	158	81
2	158	147
3	158	31
4	158	0
5	158	140

Table 4.3: Results of testing the differentiator edge detector. Five tests were performed, each capturing 158 frames. Shown are the number of frames out of 158 for which the edges were correctly detected. $\mu = 79.8$, $\sigma = 58.12$

Test number	Frames captured	Frames detected correctly
1	158	140
2	158	111
3	158	28
4	158	142
5	158	11

Table 4.4: Results of testing the area weighted edge detector. Five tests were performed, each capturing 158 frames. Shown are the number of frames out of 158 for which the edges were correctly detected. $\mu = 86.4$, $\sigma = 102.94$

a system that is more robust.

Area Weighted Edge Detector

The “Area Weighted” edge detector was developed in an attempt to eliminate ambiguities between potential track edge candidates by taking a more human approach to the task of scanning for an edge.

It was found that this edge detector was more robust to noise caused by poor reception, as the pure difference between the weight given any two arbitrary edges was on average much greater, proportional to the noise, than when the simple differentiator was used. However it tended to perform extremely poorly on corners, where the width of the track relative to the rest of the floor changes. The greater variance is the result of the prior knowledge working both as an advantage and a disadvantage. Table 4.4 shows the results of testing this edge detector.

Tests three and five were performed in corners, showing the poor results of this detector compared to the previous detector in the same conditions.

The Edge Locker

The simple differentiator that was initially written to detect edges was somewhat good at detecting the maximum difference in a scan line, and therefore the strongest edge. In some conditions, especially low reception, the differentiator would alternately find several strong edges, from one frame to another. Due to

Test number	Frames captured	Frames detected correctly
1	158	158
2	158	157
3	158	109
4	158	149
5	158	72

Table 4.5: Results of testing the sub image pixel block filter. Five tests were performed, each capturing 158 frames. Shown are the number of frames out of 158 for which the edges were correctly detected. $\mu = 129$, $\sigma = 33.69$

random pixels occurring on the edges of objects in the video display, there was no way to guarantee that the strongest detected edge was not just random noise pixel creating an edge.

The adaptive edge locker was a method derived to potentially alleviate this condition. The edge locker works by allowing a detected edge to be “locked on” to. When the next frame is processed, the edge locker will not scan the entire scan line, but only a small area near the expected location of the locked edge. This small scan areas is known as the “search box.” This technique takes advantage of consistency between frames, as the track is only expected to move a small amount.

This solution was designed to research what would happen if consistency was exploited, due to the problems seen occasionally in testing the edge detector by itself. It was only applied to the differentiator. The results were very good, as the software would not detect an incorrect edge once the correct one was locked on. The problem created by this was that there was no way to determine if the locked edge was the correct edge, and no way to automatically activate and de-activate the edge locker once the car was moving.

4.5.2 Filtering

This section presents the results of testing the image filters which make up the bulk of the image analysis library, designed for this project.

Sub-image pixel block averaging

Averaging sub-image pixel blocks by pixel values worked to smooth out a lot of lower level noise, such as quantisation noise from the CCD camera.

In optimal conditions this results in excellent accuracy, however this tends to decline somewhat under glare conditions and is still subject to the same problem of obstacles outside the track, such as black cables or chair legs, causing distractions. The problem with glare on the track is that the magnitude of track edges is dulled somewhat due to the averaging process, making the edges of a glare highlight relatively stronger, thus more prone to being detected as an edge. Test 5 is an example of this.



Figure 4.4: Results using an averaging filter with the block size set to 6 x 2 pixels. This block size was found to be the most efficient by Bruton (1999)

3 bit Filter

The 3 bit filter gave impressive results under a wide range of conditions. It was found by analysing the histogram an average video frame that an optimal threshold value of around 200 was sufficient to classify as much of the frame as possible below the threshold while leaving the track intact as white. Figure 4.5 shows a histogram created with GIMP¹ which was used to determine what threshold values would minimise the number of pixels classified as “bright” without classifying any track pixels as “dark.” Figure 4.6 shows the results of using this filter.

Table 4.6 shows the results of testing the 3 bit filter as described in Section 3.6. As can be seen from these results, it was extremely difficult to create a situation under which the filter did not classify the track as black and the edges as white. This gives the greatest possible contrast, so no random noise will give a stronger edge, unless it appears on the track itself, before the track edge in the scanning sequence. To minimise the possibility of this, the detectors scan the track from inside to outside, the track edge being more likely to appear before any large noise this way.

Test five is a special case, where an unreasonable amount of light was shone directly into the middle of the track. This totally washed out the image, leaving the track almost indistinguishable in places, even to the human eye, after being received at the host PC. It is reasonable to assume, as in Tung (2000), that ambient lighting conditions are adequate for the car to operate correctly. Taking this into account, the aim of these filters is to increase that operating range. Test five

¹GNU Image Manipulation Program – <http://www.gimp.org/>

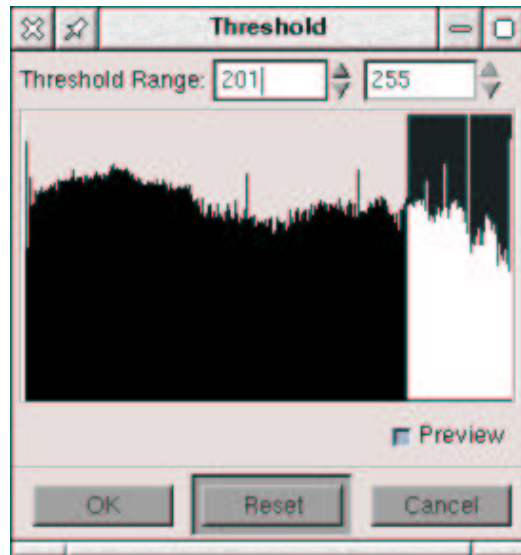


Figure 4.5: A histogram of an average image frame. Using the GIMP program it was possible to vary threshold ranges and analyse what effect (if any) threshold values had on the track pixels and other pixels in an image.



Figure 4.6: Results using a 3 bit filter. As can be seen, the track is largely untouched – being classified almost exclusively in white pixels, while the track is colored black. Pixels outside this vary, but have no effect if they do not occur before the track edge in the scanning pattern.

Test number	Frames captured	Frames detected correctly
1	158	158
2	158	158
3	158	155
4	158	157
5	158	8

Table 4.6: Results of testing the 3 bit filter. Five tests were performed, each capturing 158 frames. Shown are the number of frames out of 158 for which the edges were correctly detected. $\mu = 127.2$, $\sigma = 59.61$

however still proves the fragility of simple filters.

Monochrome Filter

The monochrome filter was found to be somewhat effective under optimal lighting conditions, for the same reasons as the 3 bit filter, but failed under difficult conditions. Figure 4.7 shows results in perfect lighting. As can be seen, the results are clear and the track is highly visible. However, the search algorithm failed to locate the track a large proportion of the time, as interfering obstacles were also very prominent and, by definition, were exactly as well defined as the track edges. This made it impossible to detect any difference between the track edge and irrelevant obstacles, using a naive single scan line algorithm, where reflections etc. appeared on the track. In contrast to this, the 3 bit filter was able to cope well with some degree of track highlighting, as most track reflections were colored either blue or cyan, rather than white.

Nonlinear Filter

It was found that the output of a nonlinear filter was excessively slow for the quality of results. A test of the output speed in fps showed that when running the filter on block sizes of 5 by 5 pixels, the speed of the main loop slowed from 24.6 fps to 11.75 fps. Speed increases were achieved by increasing the size of each block and thus minimising the amount of total averaging overhead per frame. At higher block sizes however, the filter was relatively insignificant, as shown in Table 4.7.

4.5.3 Combining Filters

The most interesting combination of filters proved to be the 3 bit filter, followed by the averaging filter, with a block size of 6×2 . This was a very accurate and stable combination, which was however prone to high CPU usage. This slowed down the available capture rate to approximately 21 fps. This level of performance is not unmanageable, but better results were achieved for similar frame rates using the MML detector.

The nonlinear filter also had the effect of slowing down frame rates in combination with other filters. It was found that the poorest combination in terms of

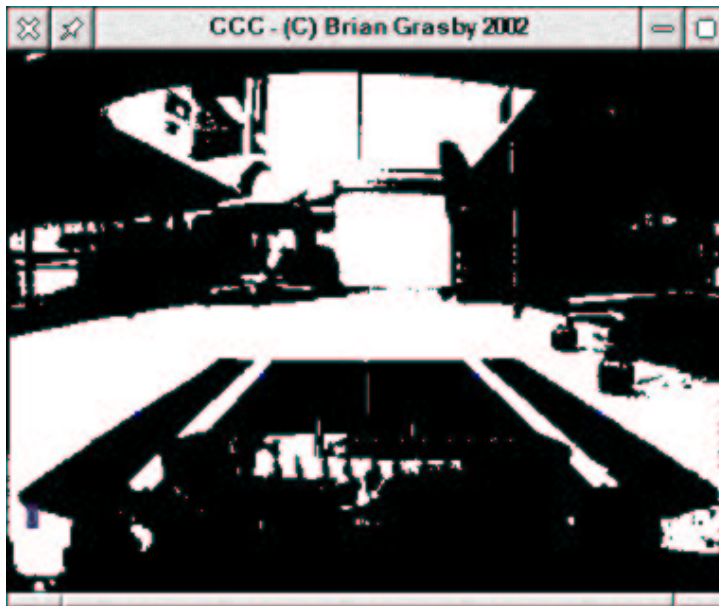


Figure 4.7: Optimal results obtained from using a monochrome filter, thresholding pixels greater than 50% intensity to white and all others to black.

Test number	Frames captured	Frames detected correctly
1	158	90
2	158	110
3	158	101
4	158	60
5	158	84

Table 4.7: Results of testing the nonlinear filter. Five tests were performed, each capturing 158 frames. Shown are the number of frames out of 158 for which the edges were correctly detected. $\mu = 89$, $\sigma = 17.04$

Test number	Frames captured	Frames detected correctly
1	158	6
2	158	4
3	158	8
4	158	9
5	158	0

Table 4.8: Results of testing the CMVision library. Five tests were performed, each capturing 158 frames. Shown are the number of frames out of 158 for which the edges were correctly detected. $\mu =$, $\sigma =$

detection accuracy was the nonlinear filter combined with a monochrome filter, which made the track largely undetectable from some angles.

4.6 CMVision

The CMVision library was tested in the same way as any other image filter, as the output was standard RGB, as opposed to the YUV color space input. This made CMVision fully compatible with the other filters and edge detectors, so long as the capture format was YUV and the CMVision library was the first filter in the chain. This having been mentioned, CMVision is nothing more than a larger scale image filter.

The results of testing CMVision are shown in table 4.8.

The results are quite poor due to the difficulties involved in tuning the YUV threshold values for color segmentation. Quantisation noise makes the library extremely volatile and segmentations change by very large amounts between subsequent frames, on very little noise. Figure 4.8 shows the noisiness of a particular segmentation. This would indicate that the thresholds are too tight, however increasing the ranges for each threshold makes the segmentation incorrect.

4.7 Minimum Message Length

The message length based segmentation function works in a similar manner to the edge detectors, rather than as an image filter. It can work in combination with filters and edge detectors, as it processes the unfiltered input from the camera without modifying the original image. Analysis functions are then able to process the image as normal. Testing problems were however uncovered combining MML with CMVision, as the MML function could not work on YUV format images. The solution found to this was to move the MML routine to work on the image after it had been transformed to RGB color space.

Results from testing the track detection using MML were also impressive. The software was able to identify the track under extremely heavy noise conditions, as illustrated in Figure 4.9. The detection was more stable than using edge detectors, which had a tendency to oscillate between edges rapidly at moderate noise. The

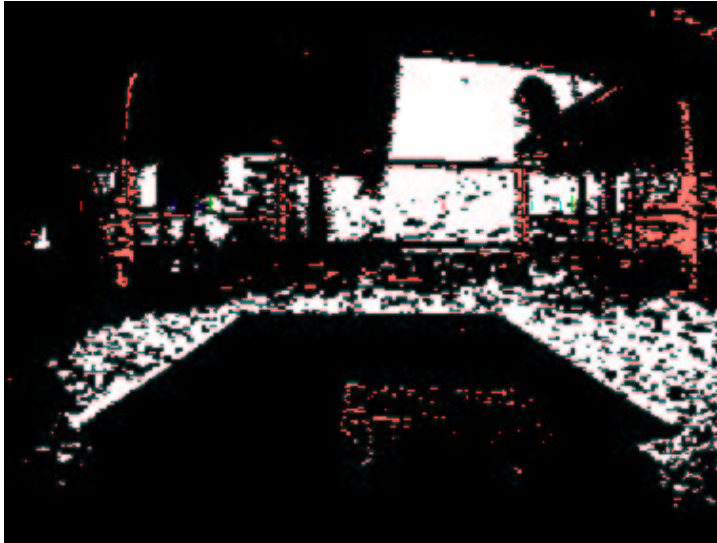


Figure 4.8: Results testing the CMVision library

Test number	Frames captured	Frames detected correctly
1	158	158
2	158	156
3	158	153
4	158	158
5	158	148

Table 4.9: Results of testing the MML detector. Five tests were performed, each capturing 158 frames. Shown are the number of frames out of 158 for which the edges were correctly detected. $\mu = 154.6$, $\sigma = 3.77$

MML detector also had this tendency, but it was not present until the track became highly obscured.

The results of testing a typical run using the MML detector under more normal noise conditions are shown in Table 4.9.

4.8 Motion Library Testing

The new motion library, generating only linear combinations of previously programmed moves, was tested against the more general previous approaches where a state machine was used to output dynamic motion commands depending on the position and heading of the car.

It was found that the previous approach was useful, as it was able to generate control commands that were customised to the situation, whereas the new library could not do this. Appendix C.2 shows the errors between expected position and actual position after the completion of a single move using the new library.

The results show that the error is significant, at an average distance of 15%

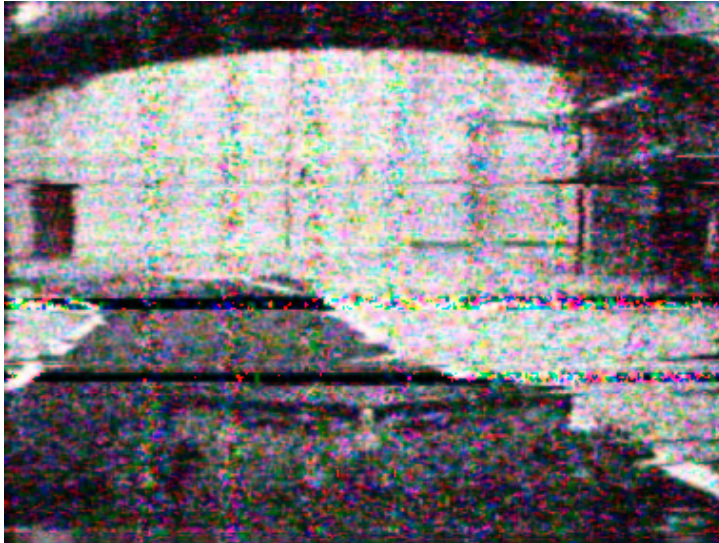


Figure 4.9: MML detector working on a noisy image. The vertical green marker indicates the MML detected centre of the track.

of the size of one tile for a single turn, which compounded to create irrecoverable errors over the length of only 6 moves. The track quickly became totally lost to the vehicle. The new library had no allowance for this circumstance, as it was designed so that this situation would never occur, to make the process simpler.

A second library, implementing a more traditional approach was also tested. This proved far more successful, and closely followed some of the ideas by Bruton (1999), namely to implement a state machine type of code, with a state variable that was assigned first, then executed at a later time. In this implementation of the library, a motion variable was added, along with a confidence variable, which was a new addition to the idea.

The motion variable was treated as a collection of individual flags, each of which were turned on by a bitwise or operation with the direction mask of the desired move. The variable was initialised to zero each event cycle. This implemented the path planning algorithm. The state of the motion variable was analysed against the state of the car on each frame and the confidence in the current next move updated to reflect the detected track. On the last frame of the event cycle, the move with the highest confidence was executed.

The car was able to navigate reliably at a rate of $0.5m/s$, but was capable of much higher rates of movement. Speeds in excess of $2m/s$ were recorded, at which the car was extremely unstable due to the lack of proper braking control.

Chapter 5

Conclusions

It has been shown through this research that it is possible to improve both the vision and motion of the Computer Controlled Car. This chapter presents the implications of this projects results and conclusions drawn thereof.

5.1 Image Capture

It has been shown that image capture is a major limiting factor in the processing speed of the current software, as it has previously been. This limit does not imply a lack of processing power to control the car, rather highlights the main bottleneck to increasing spare CPU time where required. The speed of capture using the video card class derived from this project is such that that bottleneck tends to become less significant compared to more processor intensive filtering algorithms. This shows that although image capture can take up a large amount of time, computer hardware has progressed to the point where this wait is within the bounds of normal operation and the time left is ample to perform all necessary operations on the car.

5.2 Motion Library

The track was much more clearly defined in this implementation which helped the new imaging software. Unfortunately, due to the nature of the car hardware, the a priori knowledge of the track metrics could not be taken advantage of in the motion software. The new motion library design did not work optimally because the previous “dynamic” approach was more suited to the racing paradigm. The design behind the new motion library was a good paradigm for the track design because it made all shapes of track possible at scales that are always navigable by the car. A motion library based only on set moves however, is not adaptive enough to prevailing conditions over which the software has no control. The increased resolution of re-evaluating the moves with each frame of input is vital to the success of the car.

The design of the serial output system is critically limited. A move which must be completed under a certain time limit must be limited to a fixed number

of instructions per time period, otherwise the car does not move at all. Contrast this with the instructions given by a human using the remote handset, that can issue an arbitrary number of commands per second, up to the physical limit of the human's movements, which can be re-evaluated and over ridden at any time by the next movement. This does not discount the use of the current system to accomplish human like driving, but reinforces the need for a path planning algorithm to take in a much longer range of vision and more detailed planning, up to the limit of the hardware's abilities.

5.3 MML

Results clearly show that track detection techniques requiring large numbers of pixels per frame are not required to obtain knowledge of where the track lies, in most conditions. Good performance is achievable with only one or two scan lines of the image. The best techniques tested are those that do not rely only on a relatively small number of pixels out of each scan line to detect the track. This is partly due to the fact that the image contains a lot of information that is often discarded by many techniques, the track for example is a very large segment of the image. Most scan line techniques treat all but 10% of the scan line as "background" pixels and pixels of interest may be so only due to the fact that they are noise pixels.

Techniques that use segmentation or manipulation of large regions in a scan line have more information per segment to work with, relative to the noise per segment found in the average image. For this reason, MML and the averaging filter returned good results. the nonlinear filter on the other hand only modifies one pixel per block, and so changes the image relatively little, which is detrimental in high noise, as the likelihood of more than one noise pixel per block is quite high.

5.4 CMVision

It is obvious from the results that CMVision's creators have achieved that robust segmentation by color is possible with the library. It is not known whether the failure of CMVision to perform was due to the high noise experienced, or the difficulty in finding optimum thresholds for color segments in the environment. It is believed that better results are achievable with CMVision, however tests may prove that these results are still not viable under the conditions that the CCC experiences.

One point stands out, and that is the fact that color thresholding alone is difficult in the CCC's environment, due to the fact that colors of interest have so much range. There is a lot of overlap between colors and it is difficult to find an optimum level, although above average results were shown with the 3 bit filter working at a threshold of 200.

Chapter 6

Future Research

6.1 Serial adapter

The current adapter cannot feed instructions to the car in a precise manner. This is seen from the testing of the motion library. It is very difficult to make the car perform a precise move, as the smallest possible move is quite long and therefore requires a large amount of time to complete, during which the environment changes rapidly. Higher resolution of motion commands and timing of output to the car is required to generate commands to the car that represent an achievable and fast motion stream. A solution to this may be found in a new interfacing device that is capable of processing commands to the car at the same rate that a human can send them using the remote control alone.

6.2 New Model Car

It is recommended that the CCC software be ported to a new hardware platform before any future research into developing new techniques is undertaken. The current model car hardware creates problems that are not trivial to overcome in software, especially in the area of accurate motion control. A fully proportional model is required to gain a finer level of control over CCC movements. This will not only mirror more accurately the real world with passenger sized cars, but should improve the efficiency of the software, as fewer iterations and serial output commands will be required to achieve the precise desired movements. This should be tested to determine the effectiveness of the current motion and image libraries on different kinds of hardware.

It is expected that the effectiveness of the image analysis library will not change excessively on a new car model. The motion library should be portable by rewriting only the *moveCar* function to suit a wider range of serial commands. New states may have to be added to this function to allow the new movements to be executed. This could be integrated with the current data structures passed from the analysis thread. The X Event queue will be able to be re-used as it is.

6.3 Adaptive edge locker

The edge locker presented in this project was lacking in several areas. It is proposed that the idea be developed to the point where the edge locking mechanism, by use of a search box, is adaptive to the speed of the car and can be automatically engaged and disengaged by the software. This will require a technique to be developed whereby the software can recognise a track edge with enough certainty to “lock on” to it.

6.4 Speed Detector Hardware

The hardware speed detector developed in this project is expected to be able to provide to CCC with improved performance, once it is mounted to the car and the telemetry information integrated with the motion control software.

6.5 Forward Motion Planning

A larger amount of forward motion planning is required to allow for greater speeds to be reached by the CCC. It has been shown in Section 4.8 that several frames per second must be analysed without generating any events, in order not to swamp the serial queue. This extra analysis can be used to build up track models in memory with complex path and motion planning. Finer control resolution and lower latency will allow this to be exploited, making the CCC faster to navigate and much more accurate around the track, taking a better racing line to combine a series of expected future moves.

6.6 On board filter hardware

An efficient way to implement fast filtering of images from the CCC’s camera is to implement the filtering routines in some sort of hardware, such as a CPLD or DSP chip. Such hardware ranges from cheap to moderate in price, proportional to the features provided. Most filtering routines can easily be implemented on the car using a chip with an on-board analog to digital converters and a digital to analog converters. Such chips would allow the image to be transmitted by the same means, with no other change to the hardware – the filter hardware would simply slot into place in the signal path. It is possible to implement this idea with switches, so that various filter techniques could be switched on or off, or even the order of the filters re-programmed.

6.7 Scaling to multiple processors

In order to increase the amount of processing done per event cycle, it is possible to run to software on multiple CPUs, either on the same machine, or over several machines. The current code is threaded to take advantage of this, but the current architecture should be tested for performance against different designs on a

multiple CPU machine. It is believed that the best performance increase may be available with the current architecture by aggressively multi-threading the code modules.

6.8 RALPH

Following the success of the MML routine, it is believed that similar success may be achieved by implementing the RALPH algorithm on the CCC. RALPH is a very simple algorithm in terms of CPU use and may introduce a speed increase. It is not known how this will perform against MML.

Appendix A

Software Notes

A.1 README

The following notes are taken from the README file supplied with the software developed for this project.

2.0 Compiling

```
make clean - removes all .o core and other not needed files
make          - dcar (debugging output, screen display, profiling info)
make car      - car (no debugging, screen display, no profiling info)
```

Compile Flags

```
-DDEBUG -DDISPLAY -DPROFILE -DRALPH -DDET_EDGES -DFILTER -DCMV
Turn on : Debugging output (console stdout)
          Main X display on or off
          Profiling Information (frames/second)
          RALPH prototype (not working in this version)
          Edge Detectors
          Image Filters (uncomment the ones you want in car.cpp)
Turn on CMVision Library
```

the default make builds the executable "dcar" (devel-car)
this is the software to control it, there is nothing else.
Not all of the code commented out of the default build or source
code is bad code, several functions were superceeded or could
not be tested together with other blocks. Some functions are buggy
and this is highlighted where possible.

2.1 Running

```
*Press E to engage edge locking
-Car will "lock" to current edge by attempting to only locate
edges in subsequent frames withn a small distance of the
```

current edge (assuming edges do not move much laterally from frame to frame)

- *Press Q to exit program
- *Press S to take a screenshot - saved as snap_x.ppm in the current directory
- *Press any other key to go into manual mode
 - Numpad drives the car (numlock must be on)
 - Q goes back to auto mode

A.2 Application Programming Interface

The following is a list of all the main functions in the program, ordered by the file and class in which they are found where applicable. The list includes parameters, return value and a description of each.

video_card::

```
-----
initialise(input, device, nwidth, nheight, nfmt, vFmt)
-----
```

```
int* input;
char* device
int nwidth, nheight;
__u32 nfmt;
struct v4l2_format vFmt;
```

PARAMETERS:

input is a pointer to an integer containing the number of the input jack to use (for flyvideo 2000 0=tuner 1=composite 2=svideo). device is the string path such as ‘‘/dev/video0’’. nwidth, nheight & nfmt are the width and height and video format of the capture. They are requests only - the driver will return the actual values. vFmt is a global v4l2 struct which holds this information in the video card. It initialises STREAMBUFFERS mmaped buffers for streaming video capture - for maximum speed.

RETURNS:

Boolean - True is successful, False on an error.

```
-----
captureFrame(void)
-----
```

DESCRIPTION:

captures an image from the video capture card's currently open input and returns a pointer to the buffer

where it is stored as a pointer to unsigned char (8 bit)
card must be opened with initialise() first

PARAMETERS:

None

RETURNS:

A pointer (char*) to the data is returned. This is in
the card driver's mapped memory buffers so it may be
best to move the data out quickly as streaming will be
turned on.

close(void)

DESCRIPTION:

closes the card's device and unmaps the capture buffers.
This should be called to clean up at the end of use.

PARAMETERS:

None

RETURNS:

None

motionQ::

append(m)

char m;

DESCRIPTION:

Adds a character to the queue

PARAMETERS:

m is the character to add to the queue

RETURNS:

Boolean - True if successful, False on memory error

serve(void)

DESCRIPTION:

Removes the next item from the queue

PARAMETERS:

None

RETURNS:

Character - a char value is returned which is the item that has just been removed from the queue

count(void)

DESCRIPTION:

Returns the current item count as an int

PARAMETERS:

None

xshm.h::

init_x(xdisplay, an_xwin, gc, w, h)

```
int w;  
int h;  
Window* an_xwin;  
Display** xdisplay;
```

DESCRIPTION:

Used to initialise all of the X Windows parts of the program. It opens a connection to the display xdisplay, creates a new window, an_xwin and graphics context gc. The window is w x h in size. Input masks and title bar are also set up. The window is then mapped to the display. (it should be visible upon return). The program will fail if it cannot connect to the display.

RETURNS:

None

newXShmImage(&shminfo, w, h)

```
int w;  
int h;  
XShmSegmentInfo shminfo;
```

DESCRIPTION:

This function creates a new XImage using MIT-SHM extensions. It also initialises a shared memory segment, which is arg 1. After calling this function, the new image is usable.

RETURNS:

The function returns XImage*, which is a pointer to the new image.

init_xshm(w, h)

```
int w;  
int h;
```

DESCRIPTION:

Sets up a standard XImage (contrast above) with size w * h

RETURNS:

None

init_Shm(w, h);

int w;

int h;

DESCRIPTION:

Initialises the MIT_SHM extension data - sets up a shared memory segment for fast XImage transfers into the display

RETURNS:

None

PARAMETERS:

w and h must be the intended width and height for any intended XShmImages otherwise xlib will crash.

image.h::

LedIsOn(LED)

int LED;

DESCRIPTION:

The locations on the screen are #defined in car_image.h The locations are also shown as white dots on the screen if the LED is found to be off. This allows for easy alignment of the LED array on the car.

RETURNS:

This function returns true if the numbered LED in the look up table is found to be on.

getSpeed(void)

DESCRIPTION:

Reads the LED's are as a binary number with MSB on left of screen

RETURNS:

This function returns an int value of the current speed of the car using LedIsOn() to read each LED from [0..7].

one_line_edge_x2x(row, startx, endx)

int row;
int startx;
int endx;

DESCRIPTION:

There are two variants of this function l2r and r2l which scan the lines left to right and right to left respectively. They implement a simple differentiator technique.

PARAMETERS:

row is the row number of the image to scan

NOTE: THE FUNCTIONS DO NOT CHECK THAT THEY ARE CALLED WITH SENSIBLE NUMBERS, PLEASE MAKE SURE THE START IS BEFORE THE END IN THE SCAN DIRECTION!

RETURNS:

Integer - returns the location on a row where the best edge was found

weighted_edge_x2x(row, startx, endx)

```
int row;
int startx;
int endx;
```

DESCRIPTION:

These functions are the same as the one_line_edge functions but they use a technique of giving a weight to each edge based on how large the region it bounded was. This would might better by implementing different filtering first.

PARAMETERS:

As above, with warnings

RETURNS:

As above

avg_sub_row(row, X, width, blk_width, blk_height)

```
int row;
int X;
int width;
int blk_width;
int blk_height;
```

DESCRIPTION:

This function averages all pixels in row from X for width units into arbitrary blocks of blk_width x blk_height size (*TAKE CAR WHEN USING! -- DO NOT RUN ON ANY ROW GREATER THAN IMG_HEIGHT - blk_height!*) It WILL seg fault.

PARAMETERS:

X is the starting location on row number row. It works on width pixels across, averaging them into blk_width * blk_height size blocks.

RETURNS:

None

```
-----  
drawSpeed(s, len);  
-----
```

```
char* s;  
int len;
```

DESCRIPTION:

Takes a string and the length of it then outputs it on the display after it goes into the X window so it wont appear on any screenshots taken with the snapshot function

RETURNS:

None

```
-----  
mml_by_line(scanline, road_left, road_right, mu);  
-----
```

```
int scanline;  
int road_left;  
int road_right;  
double* mu;
```

DESCRIPTION:

This function uses MML to detect regions of optimal continuity where the most pixels are closest to the mean and std. deviation for the whole of that region. This is quantised as the cost of encoding a "message" -- unlike pure MML, we have not included the cost of encoding an hypothesis in this message, as we always look for 3 regions: 1=side 2=track 3=other side.

RETURNS:

It will only return the cost of a single hypothesis on the left and right edges of the track specified in the parameters.

```
-----  
calculate_mean(mean, sDev, scanline, left, right)  
-----
```

```
double* mean;  
double* sDev;  
int scanline;  
int left;  
int right;
```

DESCRIPTION:

Only called by `mml_by_line` -- this is used to work out the mean and std. deviation in one pass.
(See LLOYD ALLISON)

RETURNS:

Boolean - True, sets `mean` to the mean and `sDev` to the standard deviation inferred for a scanline with road hypothesis between left and right.

motion.h::

manDrive(xdisplay, event, serial);

Display* xdisplay;
XEvent* event;
ofstream* serial;

DESCRIPTION:

If a key is pressed during normal operation that is not mapped to a special function, the car will enter a new event loop to process directional keypresses. Press Q to exit from this mode. It uses serial to output bits.

PARAMETERS:

xdisplay is a pointer to the X Display structure this should have been opened first (see init_X). event is a pointer to an event and serial is a pointer to the serial port's open file descriptor.

RETURNS:

None

moveCar(void);

DESCRIPTION:

called once per frame to control the movements of the car. It first takes the available data and calculates a new position, then tells the control loop that data is available to be sent to the car

RETURNS:

None

Appendix B

Screen Captures



Figure B.1: Further comparison of surface reflection



Figure B.2: Further comparison of surface reflection

Appendix C

Tables & Figures

C.1 Car Motion instructions

ASCII Command	Hex Value	Name	Output
0	0x30	NULL	Stop
1	0x31	Reverse	↓
2	0x32	Forward	↑
3	0x33	Forward (Fast)	↑↑
4	0x34	Right Turn	→
5	0x35	Backward & Right	→↓
6	0x36	Forward & Right	→↑
7	0x37	Forward & Right (Fast)	→↑↑
8	0x38	Left Turn	←
9	0x39	Backward & Left	←↓
:	0x3A	Forward & Left	←↑
;	0x3B	Forward & Left (Fast)	←↑↑

Table C.1: A List of the motion instructions that the serial adapter knows how to interpret and the corresponding moves generated by the RF remote. The Hex values shown are for the ASCII Characters used in this project for convenience – only the LSB is significant, the high byte does not matter

C.2 Motion Library Errors

Move Type	Command Stream	Position Error	Heading Error (°)
Left 45°	{2::0}	75 mm	20°
Left 45°	{2::1}	140 mm	-
Right 45°	{2660}		
Right 45°	{26661}	10 mm	5°
Left 90°	{2::0}{2::0}	200 mm	40°
Right 90°	{26661}{26661}	25 mm	20°
Left 90°*	{2::02::1}	250 mm	-
Right 90°*	{266026661}	15 mm	5°
Right 90°*	{2666126661}	20 mm	5°

Table C.2: Measured errors for the motion library

Some of these turns (designated by a *) are special “racing” turns that attempt to minimise the effects of position and heading errors accumulating over several turns by modifying the direct combinations of two or more turns.

C.3 Speed Sensor

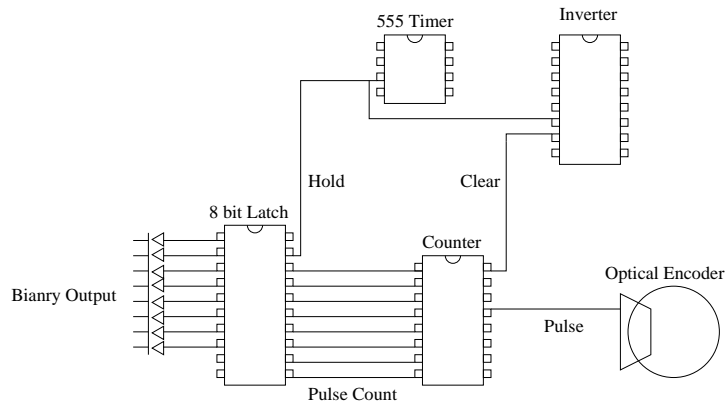


Figure C.1: A simplified concept diagram for the speed detector translation circuit.

Bibliography

- Balakrishna, S. (2000). Pthreads.
http://www.coe.neu.edu/signoril/op_sys/pthreads1.html, Last Accessed: (2/7/2002).
- Bertozzi, M. and Broggi, A. (1998). GOLD: A parallel real-time stereo vision system for generic obstacle and lane detection, *Image processing*.
- Broggi, A. (1995a). A massively parallel approach to real-time vision-based road markings detection, *Proceedings ICIP – Second IEEE International Conference on Image Processing*, pp. 532–535. Washington DC.
- Broggi, A. (1995b). Vision-based road detection in automotive systems: A real-time expectation-driven approach, *Journal of Artificial Intelligence Research* 3 pp. 325–348.
- Broggi, A. (1996). The evolution of a massively parallel vision system for real-time automotive image processing, *Proceedings IEEE IPSP'96 - International Parallel Processing Symposium*, pp. 724–728. Honolulu, HI.
- Broggi, A., Bertozzi, M. and Fascioli, A. (1999a). The 2000km test of the ARGO vision-based autonomous vehicle, *IEEE Intelligent Systems*.
- Broggi, A., Bertozzi, M., Fascioli, A. and Conti, G. (1999b). *Automatic vehicle guidance: Experience of the ARGO autonomous vehicle*, World Scientific.
- Bruce, J., Balch, T. and Veloso, M. (2000). Fast and inexpensive color image segmentation for interactive robots, *Proceedings of IROS-2000*.
- Bruton, T. (1999). Computer controlled model car. Monash University, CSSE. Honours Thesis
<http://www.csse.monash.edu.au/hons/projects/1999/Tim.Bruton> Last Accessed: (30/4/02).
- Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. and Stal, M. (1996). *Pattern-Oriented Software Architecture: A System of Patterns*, Wiley, chapter Pipes and Filters.
- Corbet, J. and Packard, K. (1991). MIT-SHM – The MIT shared memory extension. <http://pantransit.reptiles.org/prog/mit-shm.html> Last Accessed: (1/11/2002).

- Dickmanns, E., Behringer, R., Hildebrandt, D., Maurer, M., Thomanek, F. and Schiehlen, J. (1994). The seeing passenger car 'VaMoRs-P', *Proceedings of the IEEE Symposium on intelligent vehicles*, pp. 68–73. Paris.
- Duggins, D., McNeil, S., Mertz, C., Thorpe, C. and Yata, T. (2001). Side collision warning systems for transit buses: Functional goals, *Technical Report CMU-RI-TR-01-11*, Carnegie Mellon University.
- Franke, U. and Gavrilla, D. (1999). Autonomous driving goes downtown, *IEEE Intelligent Systems*.
- Gilbert, D. (2001). Magneto-resistive wheel speed sensors, *AutoInc Magazine*.
- Graefe, V. and Kuhnert, K. (1991). Vision based autonomous road vehicles, *Vision based vehicle guidance* pp. 1–29.
- Horn, B. K. P. and Schunck, B. G. (1981). Determining optical flow, *Artificial Intelligence* **17**: 185–203.
- Jochem, T., Pomerleau, D. and Thorpe, C. (1993). MANIAC: A next generation neurally based autonomous road follower, *Proceedings of the International Conference on Intelligent Autonomous Systems*.
- Kluge, K. and Thorpe, C. E. (1990). *Vision and navigation: The Carnegie Mellon NAVLAB*, Kluwer Academic Publishers, chapter Explicit models for robot road following, pp. 25–38.
- Law, L. (1998). Magnetic sensors and timing applications, *Sensors Magazine*.
- Martin, M. (1998). Breaking out of the black box: A new approach to robot perception. Doctoral dissertation, Carnegie Mellon University
http://www.ri.cmu.edu/pub_files/pub2/martin_martin_c.1998_1/martin_martin_c.1998_1.pdf Last Accessed: (31/7/2002).
- Matthies, L. (2000). A portable, autonomous, urban reconnaissance robot, *6th international conference on intelligent autonomous systems*.
- Muys, A. (2000). Pthread tutorial.
<http://www.cs.nmsu.edu/~jcook/Tools/threads/threads.html> , Last Accessed: (2/7/2002).
- Navarro-Serment, L., Grabowski, R., Paredis, C. and Kholsa, P. (1999). Modularity in small distributed robots, *Proceedings of the SPIE conference on Sensor Fusion and Decentralized Control in Robotic Systems II*.
- Pomerleau, D. (1992). Progress in neural network-based vision for autonomous robot driving, *Proceedings of the 1992 Intelligent Vehicles Conference*, pp. 391–396.
- Pomerleau, D. (1995). RALPH: Rapidly adapting lateral position handler, *IEEE symposium on intelligent vehicles*, pp. 506–511.

- Quenot, G., Pakleza, J. and Kowalewski, T. (1997). *Experiments in fluids*, Springer-Verlag, chapter Particle image velocimetry with optical flow, pp. 177–189.
- Richardson, J., Ward, N., Fairclough, S. and Graham, R. (1996). Prometheus/drive aicc safety assessment: Basic simulator. DoT Vehicle Standards and Engineering Division Contract DPU 9/81/1 , HUSAT Research Institute, UK.
- Robbins, D. (2000). Posix threads explained.
<http://www-106.ibm.com/developerworks/library/posix1.html>, Last Accessed: (18/9/2002).
- Thorpe, C. and Herbert, M. (1997). *Intelligent unmanned ground vehicles*, Kluwer Academic Publishers.
- Thorpe, C., Duggins, D., Gowdy, J., MacLachlan, R., Mertz, C., Siegel, M., Suppe, A., Wang, C., and Yata, T. (2002). Driving in traffic: Short-range sensing for urban collision avoidance, *Proceedings of SPIE: Unmanned Ground Vehicle Technology IV*, Vol. 4715.
- Tribe, R. (1996). *Advanced robotics and intelligent machines*, The Institution of Electrical Engineers, chapter Intelligent autonomous systems for cars, pp. 165–176.
- Tung, D. (2000). Computer controlled model car. Monash University, CSSE. Honours Thesis
<http://www.csse.monash.edu.au/hons/projects/2000/Daniel.Tung> Last Accessed: (30/4/02).
- Wallace, C. and Boulton, D. (1968). An information measure for classification, *Computer Journal* **11**(2): 185–194.
- Wallace, C. and Freeman, P. (1987). Estimation and inference by compact coding, *J. R. Statist. Soc. B* **49**(3): 240–265.
- Wang, C. and Thorpe, C. (2002). Simultaneous localization and mapping with detection and tracking of moving objects, *IEEE International Conference on Robotics and Automation*.