THE UNIVERSITY OF QUEENSLAND

Vision for a Humanoid Robot

By

D. J. Stonier

The School of Information Technology and Electrical Engineering
The University of Queensland

Submitted for the degree of Bachelor of Engineering in the division of Electrical Engineering

October 27, 2004

27th October,
Professor Paul Bailes,
Head of School,
School of Information Technology and
Electrical Engineering,
University of Queensland,
St Lucia, Qld, 4072.

Dear Professor Bailes,

In accordance with the requirements of the degree of Bachelor of Engineering in the division of Electrical Engineering, I present the following thesis entitled

"Vision for a Humanoid Robot."

This thesis project was conducted under the supervision of Dr Gordon Wyeth.

I declare that the work submitted in this thesis is my own, except as acknowledged in the text, and has not been previously submitted for a degree at the University of Queensland or any other institution.

Yours Sincerely,

(Daniel Stonier).

Acknowledgements

Dr. Gordon Wyeth for introducing me to the field of robotics and patiently tolerating this pagan mathematician when bumbling around with hardware he knew very little about.

David Prasser for his advice on the image processing code.

The rest of the GuRoo team for their unstinting support.

Abstract

This thesis describes the continual development of the vision system for the University of Queensland's humanoid soccer playing robot. The vision system consist of a camera, vision board with SH4 processor and peripherally, an inertial measurement unit.

The work on the vision system represents the integration of the previous two years of work on both hardware and image processing routines to successfully enable the vision system to detect and track objects in its field of view whilst online (on the vision board).

It also completes the migration of the cross-platform environment for programming the embedded system to windows and begins an exploration into the concepts required for fusing inertial and visual data to better determine world-coordinate positions of both objects and robot within its local environment.

Contents

List of Figures

Chapter 1

Introduction

As an introduction, a brief discussion concerning humanoid robots and vision systems will be discussed. This is followed by an outline of the remainder of the thesis.

1.1 Humanoid Robots

Many conventional robots exist in industry performing automated tasks in an isolated environment away from the challenges posed by the usual interferences and challenges we as humans subconsciously navigate every day. However, the push for development of robots to operate within society to aid humans in tasks trivial in nature has become apparent in recent years.

Despite this increased interest, the challenge of creating a robot that can perform even the simplest of tasks in a human environment remains incredibly complex. This is significantly due to the overwhelming integration of subsystems that are needed to co-operate to achieve a single task. In this respect, robotics is still far behind the natural ability of a human.

This is where the humanoid robot emerges. It not only emulates an image that we are comfortable to interact with, but also emulates the advanced nature of a human's total integration of sensors and controls.

1.2 RoboCup Competition

The RoboCup is an international competition in which various types of robots compete against each other in games of soccer. The goal of RoboCup is to promote research in robotics whilst at the same time providing a series of increasingly complex problems for robotics and artificial intelligence development. The University of Queensland is working on their own humanoid robot, the GuRoo ("Grossly underfunded Roo") to compete in future humanoid soccer competitions.

1.3 GuRoo

The GuRoo is approximately 1.2m high and carries onboard power and computer systems (refer to Figure 1.3.

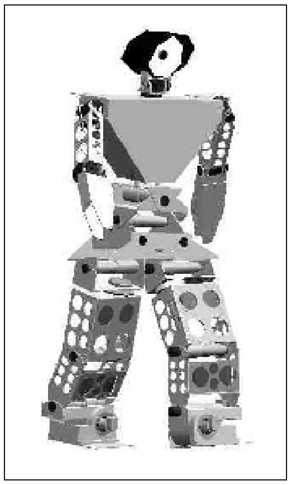


Figure 1.1: The GuRoo

GuRoo is intended to be fully autonomous. It is controlled from a central computer (iPaq) that is used to drive the many control processors as well as handle the generation of walking patterns/behaviours and gameplay intelligence.

Also on board the GuRoo will be a SH4 processor that analyses output from a CMOS camera as well as an IMU (inertial measurement unit). SH4, camera and IMU will all be mounted in the GuRoo's head.

1.4 Robotic Environmental Sensors

One of the most critical problems in robotics involves extracting information from its surroundings to develop an awareness of both its local environment and its location within that environment. A variety of sensors can be used to achieve this task, however the approach to the design of a humanoid robot will require that this must be attempted in a manner similar to that evolved by humans.

Fundamentally this entails using the sense of vision coupled with a unit resembling a human's vestibular (inner ear) system. The vision system is our primary sensor and is consciously used to identy objects within the external environment. The vestibular system plays a role that is largely subsconscious, though no less important. It is critical for balance and in aiding us to predict responses that cannot be met by our vision system alone.

On a humanoid robot these systems must be online and operating as close to real time as possible to be useful. This can be a challenging task both in hardware and software to meet this requirement.

It is planned to further develop and integrate camera, vision board (SH4) and inertial measurement unit (IMU) to meet this task in relation to the specific goals required whilst competing in a RoboCup competition (ball detection, self-localisation, pose determination).

1.5 Thesis Outline

- **Background:** Provides an introduction to the Guroo vision system and the status of both hardware and software at commencement of the project.

- **Project Definition:** This chapter outlines the project goals and the tasks that had be met to further development towards these goals throughout the course of the year.

- **Cross-Platform Programming Environment:** Documents the processes involved in setting up the cross-platform environment for embedded programming of the SH4 on a Windows host.

- **VDebug:** This chapter describes the various problems in the Vision Debugging software as well as the motivation and processes involved in implementing various extensions to the software.

- **VBSH4:** The problems in cross-compiling for the SH4 in windows are discussed as well as the issues that arose in adding the image processing routines to the program. It also describes the more important elements of the VBSH4 source code so that future students do not have to spend unnecessary time in learning how the software operates.

- **Future Work:** Outlines future work that can be immediately addressed by students carrying on with the project.

- **Visual and Inertial Fusion:** An exploratory outline of concepts related to fusing visual and intertial data that was gained from readings earlier in the semester.

Chapter 2

Background

This chapter provides a brief introduction to the various hardware and software applications that constituted the Guroo's vision system at the commencement of the project.

2.1 Hardware

Previous work on the Guroo's vision system had already included the selection of camera as well as the choice of processor and design for the vision board to interface with the camera. Together with a PC that can be connected to the vision board via a serial connection, these currently comprise the subsystems responsible for image acquisition and processing. Refer to Figure 2.1.

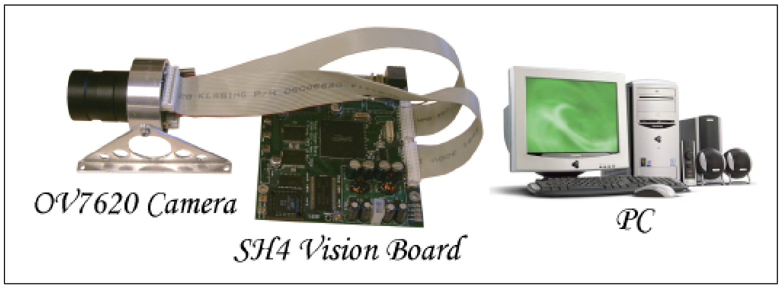


Figure 2.1: Vision System Hardware

In future, once the system is established and mounted on the guroo it is expected the role of the PC (currently used for debugging the vision routines) will be replaced by a direct connection to the Guroo's *brain* - an iPaq.

2.1.1 Camera

Head Design

Although the Guroo currently has two camera's mounted into the headpiece, the system currently only connects one camera to the vision board. At this early stage of development with the current equipment, the goals are simply to enable monocular vision. Moving to fulfill design requirements in hardware and software for stereo vision is projected for future developments.

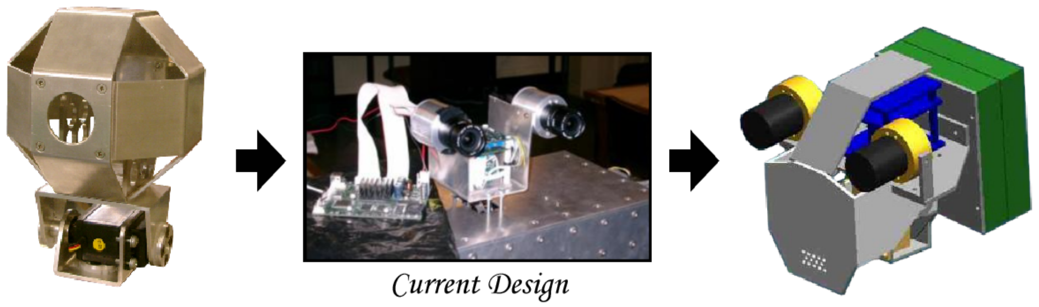


Figure 2.2: Head Design

The headpiece illustrated in the centre is a makeshift arrangement whilst the Guroo's head remains in the process of being redesigned. The redesign is intended to incorporate a stereo camera arrangement, housing of the vision board and the addition of an IMU (Inertial Measurement Unit). It also aims to provide a more aesthetically pleasing look for a humanoid robot.

Camera

The camera is a CMOS camera controlled by an OmniVision OV7620 onboard chip. The onboard chip incorporates a wide array of image processing and acquisition features that enable variable bit data output in RGB or YUV format, subsampling, interlacing, filters and customisable routines that automatically adjust for image exposure lengths, brightness, gain, and white-black pixel ratios. The default resolution is 640x480 with a frame capture rate of up to 60Hz, however its most important function is its ability to capture a smaller viewport of the actual image.

The primary advantages of such a system is the removal of processing routines from the board to the camera chip itself - previously RGB$\rightarrow$YUV conversions and subsampling had to be performed on the vision board itself, necessitating an expensive time cost to preprocess the images before object detection analysis could be performed.

The OV7620 manual can be found on the accompanying CD in

"GurooCD://vision-material/documentation/OmniVision OV7620.pdf"

This is an important manual to become familiar with if trying to improve image quality by configuring various aspects of the image acquisition process.

Lens

The lens chosen for the camera is an AVENIR SSV0358. Both lens and housing were chosen and remodelled last year by Anthony Peters. More information on this process can be found in [13].

2.1.2 Vision Board

The vision board was designed by Mark Change for use in the University of Queensland's ViperRoos robotic soccer team. The design requirements are similar in most aspects to that of those required by the GuRoo.

The core components on the board include an Hitachi SH4 microprocessor capable of running at 360 million instructions per second (MIPS) for the primary image processing functions, a Xilinx Spartan II FPGA that resides between the SH4 and the camera for secondary processing, 512KB static ram (SRAM) and 16MB static-dynamic ram (SDRAM).

The FPGA is not currently utilised, future development may include it to buffer the data before it reaches the SH4 as well as carrying out some pre-processing on the data before it reaches the SH4.

The board also includes usb and serial output connections. The debugging utility on the PC currently makes use of the serial connection and is adequate for its purpose. The usb interface would improve the ease of debugging, however the interface needs the development of an accompanying usb driver set for the board. Consequently it is not a priority at this time.

2.1.3 PC

The PC provides a suitable platform for execution of the vision debugging software Vdebug. The Vdebug software may optionally run on either Linux or Windows platforms. The software sends requests for images or data (or both) to the vision board via the serial communication link between PC and vision board.

2.1.4 Inertial Measurement Unit (IMU)

Although not strictly part of the vision system, its inclusion here is a direct consequence of its importance in conjunction with the vision system to enable location and pose estimation for the GuRoo. An exploratory discussion on these concepts will be investigated in Chapter 8.

The IMU is a product of the CSIRO labs and consists of three primary sensor units:

- One (1) 3-Axis Gyroscope.

- Three (3) Magnetometers.
- Four (4) Accelerometers.

Unlike most inertial measurement units, it also incorporates complementary filtering and correctional analysis on the measured data to provide estimates of velocities and positions. More information can be found in [2].

2.2 Software

2.2.1 VBSH4

VBSH4 is the application name for the software compiled to be run on the SH4 processor located on the vision board. It is responsible for configuring and acquiring images from the camera. Upon commencement of the project, VBSH4 was functional in terms of its capability to acquire images, although image quality and rate of acquisition had room for improvement. Analysis and the continued development of VBSH4 is detailed in Chapter 6.

2.2.2 VDebug

Vdebug is a debugging utility that allows transfer and display of images from the vision board to the PC. It was originally written on Linux in ansi C using gcc and the openGL and glut libraries, but was ported to MSVC++ and Windows last year by Anthony Peters [13]. Upon commencement of the project, VDebug displayed the images retrieved from the vision board and performed a cursory analysis that produced a YUV map of the image as well as a histogrammatical representation of the RGB/YUV componets.

2.3 Image Processing Analysis

In 2001, David Prasser investigated techniques for object detection and tested various algorithms on static images in his undergraduate thesis [14]. His procedures can be found on the accompanying CD in

"GurooCD://vision-material/software/guroo_stonier/prasser/"

In order to integrate David Prasser's image work into the vision software a thorough understanding of the techniques involved is required. This is essential in ironing out the inevitable bugs as well in acquiring enough knowledge to intelligently optimise the routines - a concern of high priority given the limited processing power and memory available on the vision board. A description of the more important image processing concepts relevant to robotic vision and object detection is outlined below - for more detailed information, refer to [14].

2.3.1 Overview

Real time object detection in robotic vision is notoriously difficult and is presented with many of the difficulties encountered in human vision. Rapid changes in the sampled images can provide little in the way of useful information and differing levels of ambient illumination can entirely change the context of our search for objects within the images. Typically in the RoboCup competitions however, the colours of all significant objects is standardised and landmark features are often used. Processes used for extraction of objects within the image are usually simple and optimised for speed. The first step in developing the GuRoo's vision sytem is to extract real-time information concerning the location of an orange soccer ball.

2.3.2 YUV Colour Space

One of the primary difficulties in extracting useful colour information from an image is in resolving the problems that varying levels of illumination can cause. These effects can be minimised by working in YUV space (as opposed to the more familiar RGB space). In YUV space, Y represents the intensity of the light, while U and V represent red and blue chrominance respectively.

Note that in many situations, the role of U and V is swapped, that is, U and V represent blue and red chrominance respectively. The representation defined above however is the representation the camera uses within its specifications.

YUV can be a little difficult to visualise. Often it is easiest to begin with the RGB colour cube - refer to the first diagram in Figure 2.3.2.

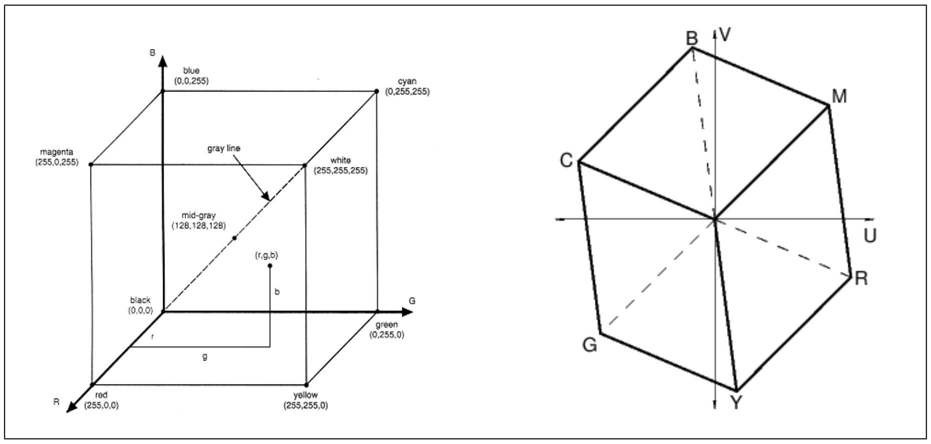


Figure 2.3: RGB Cube and YUV Space

The gray line represents the Y axis, while the UV plane lies transverse to this axis (though not quite perpendicularly). Some UV maps for varying levels of luminance (Y) are illustrated in Figure 2.3.2. The advantage in using YUV space is that changes in luminance largely affect only the Y value. This can be seen in the UV maps in Figure 2.3.2. The actual transformations relating RGB to YUV values is linear but varies slightly from camera to camera. As the camera may be programmed to calculate these internally, there is no need to implement the transformations on the vision board, but the reverse transformations are required for displaying the image in Vdebug. These will be covered in detail in Chapter 5.

2.3.3 Luminance Thresholding

Thresholding is a simple technique traditionally used for distinguishing between foreground and background objects, although its usefulness does extend in

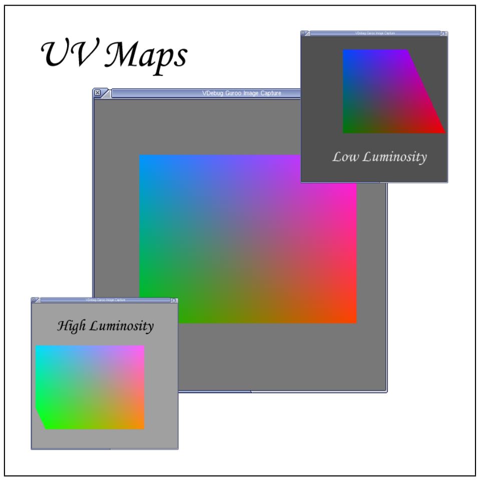


Figure 2.4: UV Maps

principle to other concepts.

Determing the average luminance of the image can be useful in **normalising** the image around a central value. This value can then be used to select a UV lookup map (see below) appropriate to the local conditions for colour segmentation of the image. The camera however, can run (on by default) automatic routines that adjust for the local brightness - this can offset the need for doing expensive average luminance calculations. A qualitative analysis on the need for such calculations would be useful.

Thresholding can also be used to discard colours in regions of the image that are excessively dark or bright. Ordinarily these regions are typically of little significance for object detection, but they also become increasingly difficult to segment properly. Recalling the RGB cube in 2.3.2, it should be easy to observe that as one approaches the corners at either end of the gray line (the luminance axis), the corresponding UV space (slice) must become vanishingly small. Consequently the process of colour classification in these regions becomes sensitive to measurement errors.

When thresholding dark and bright regions, a segmentation process will typically classify these regions as black or white. It is worth keeping in mind that white regions can often be a result of specular reflection on coloured surfaces such as an orange golf ball.

Colour thresholding can also be utilised, however this greatly increases the computational time (by a factor of three) and classification by UV maps provides an alternative and much faster segmentation method for colour.

2.3.4 Colour Segmentation

Colour segmentation simply classifies each pixel in the image as belonging to a subset of colours directly related to the objects we are required to detect. The colours of importance for the Robocup competition are

White	-	Field Lines
Orange	-	Ball
Green	-	Field
Yellow	-	Goal
Unclassified	-	Other

Classification in robot vision is usually done through three dimensional YUV lookup tables. Each YUV value in the lookup table is associated with a specified colour - providing a hand customisable mapping which is fast and efficient.

Unfortunately, a three dimensional table can be cumbersome to load into the limited memory on the vision board. If an 8 bit value (0-255) is associated with each of the Y, U and V intensities, the resulting three dimensional lookup table is 256^3 bytes (16M) in size.

Reducing the scale or dimensionality of the table are alternative options that need to be assessed for performance. A vector-based approach used at Carnegie Mellon University (details can be found in David Prasser's thesis [14]) is another alternative, but was found to have limitations primarily due to its restrictive rectangular classification regions as well as its inability to handle adaptive Y thresholding.

Another alternative for systems with little memory is to pie slice the regions in a UV map associated with a particular Y value [15]. This is alot more robust to variation in light conditions than rectangular classification, and does not need the memory for a lookup table. It will be mentioned in more detail in

Chapter 5.

2.3.5 Morphology

The interference caused by noise often associated with segmentation of low resolution images can often be reduced through the morphological processes of *erosion* and *dilation*. Each operation acts on each colour separately. Erosion cleans the fringes of coloured clusters whilst dilation will tend to fill in holes caused by speckled noise or misclassified pixels that lie on the boundary of a segmentation region. The processes are illustrated in Figure 2.3.5.

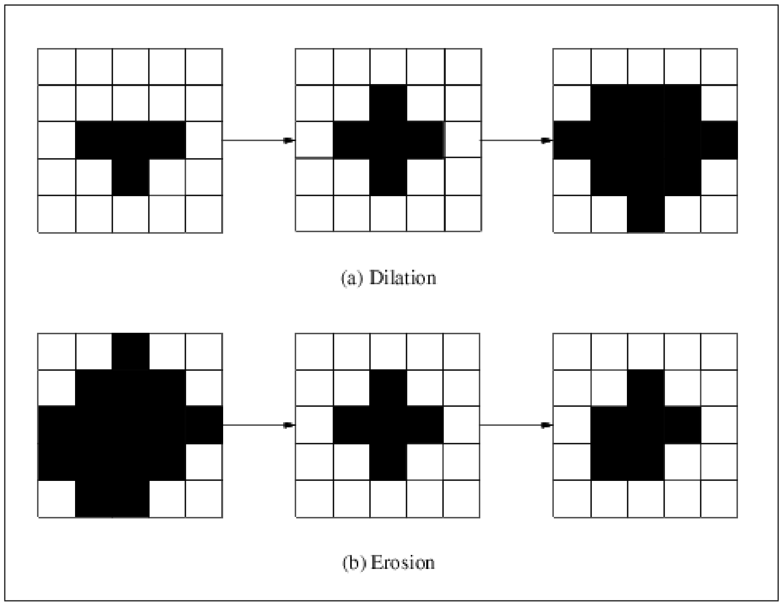


Figure 2.5: Morphological Operations of Dilation and Erosion

A combination of these two processes are often used in tandem. An *opening* is the process of an erosion followed by a dilation, whilst a *closing* is the opposite. David Prasser implemented erosion on 1x3 structuring elements (compare with the 3x3 elements pictured in Figure 2.3.5) as a quick means of reducing noise. Relevant demonstrations of how this compared to other combinations (openings, 3x3 structural elements) in terms of effectiveness and speed were not available.

2.3.6 Run Length Encoding

Run Length Encoding is the first stage of the grouping process. An often used representation for bitmaps, a run length encoded image encapsulates each horizontal block of pixels of identical colouring. This allows the image to be stored in a compressed format (convenient for transmitting to the iPaq/PC if necessary where it can be reconstructed).

Each run length encoded structure is identified by the following parameters:

- **begin:** The x co-ordinate corresponding to the beginning of a run.
- **end:** The x co-ordinate corresponding to the end of a run.
- **y:** The y co-ordinate of the run.
- **colour:** The colour of the pixels in the run.
- **tag:** The number of the run counting from the bottom left corner.
- **blobpointer:** A pointer to the RLE's blob structure.

The last two values are necessary for further grouping analysis methods (blobbing). The tag value of zero is reserved for terminating a list, whilst blobpointers are initially set to null.

2.3.7 Blobbing

After run length encoding, neighbouring runs (in the vertical sense) are grouped together into blobs if they are connected by at least four pixels. If two runs are connected and no blob has yet been set for them, a new blob object is manufactured, otherwise both blob pointers are set to point at the older blob (or at the defined blob if only one of the runs currently points at a blob). The process is illustrated in Figure 2.3.7.

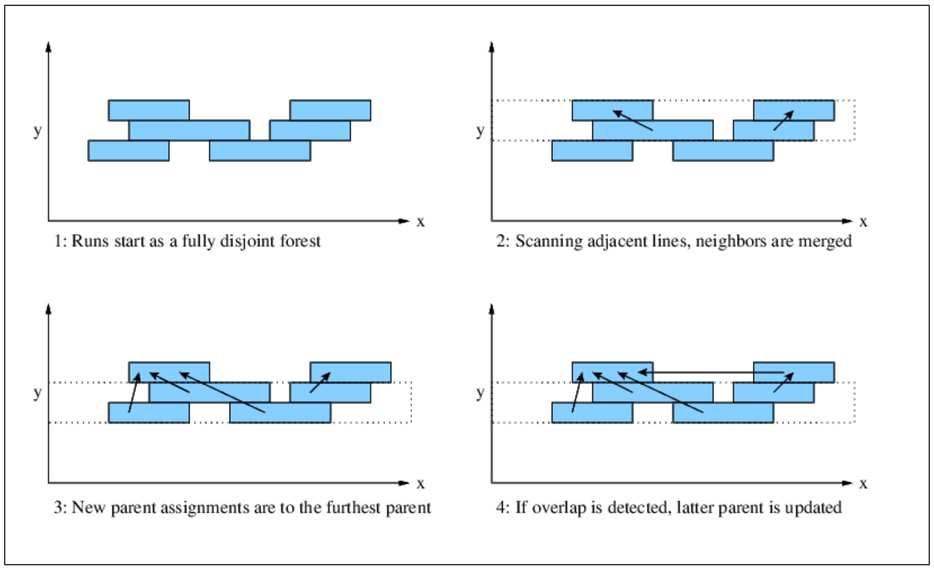

y
x
1: Runs start as a fully disjoint forest
2: Scanning adjacent lines, neighbors are merged
3: New parent assignments are to the furthest parent
4: If overlap is detected, latter parent is updated

Figure 2.6: Grouping Runs into Blobs

The process occasionally involves reiterating over the entire list of runs up to the current point to properly sort the blob list. This is an essential step although not quite obvious at first glance in the code.

Each blob is associated the following parameters:

- **colour:** The blob's colour.
- **xmin,xmax,ymin,ymax:** Bounding corners of the blob.
- **area:** Blob area. Updated as each new run (or blob) is added to the blob.
- **valid:** Identifies if it is currently an active part of the blob list or has been discarded.
- **cx,cy:** Blob centroid - centre of the rectangle defined by xmin,xmax,ymin,ymax.

Determination of the blob's centroid can be more effectively determined using more complicated algorithms at the cost of processing speed.

2.3.8 Blob Analysis

The final stage of object detection is to correctly distinguish objects from possible incorrect matches. This process is dependant on the type of objects being detected and the algorithms will ultimately depend on the final implementation.

The first step is to correctly detect and classify a soccer ball within the image which is relatively easy to classify given its rotational invariance. To properly distinguish them from other blobs of identical colouring may require additional parameters to be assigned to the blob that evaluate how well an ellipse may be fitted to the structure.

Chapter 3

Project Definition

The previous chapter provides an evaluation of the status of the Guroo's vision system at the commencement of the project. Although David Prasser established the basic image processing routines (tested on static images only) to be used for the vision system in 2001 and Anthony Peter solved some of the hardware related issues in 2003, the system had yet to be integrated as a fully functional real-time system.

3.1 Vision System Goals

The primary target requirements of a real-time vision system for the Guroo involve

- **Object Localisation** - Identify and track target objects in its field of view.
- **Self Localisation** - Identify landmarks in the surrounding environment to determine the robot's location.
- **Pose Information** - Obtain a good estimate for the robot's pose in three dimensional space.

3.2 Project Tasks

In meeting these goals, the following issues were identified as necessary steps.

3.2.1 Improving Image Quality

The quality of image being received from the camera/vision board remained below par despite the hardware improvements made in the previous year. Consequently improving the image quality was to be a priority for the duration of the project.

3.2.2 Repairing Vdebug

Vdebug in its current state at the commencement of the project was quickly found to have some major problems in its code. Callbacks, zooming, scaling, region of interest specifications, colour transformations (related to improving image quality) and yuv/histogram mapping were all either broken (after the addition of various features in the previous year) or incorrectly implemented mathematically. These had to be systematically re-written.

3.2.3 Creating a Cross-Platform Environment

Previous programming for the SH4 had been done using cross-compiling tools on a Linux or Solaris host platform. Moving the programming environment to Windows was a priority so that undergraduate students unfamiliar in a *nix environment would not lose time in familiarising themselves before beginning further development on the vision system. It was originally understood that this had already been done in the previous year, however apart from porting Vdebug's code for the PC into Visual C++, any details regarding setting up the environment for SH4 programming in windows turned out to be invalid and untested.

3.2.4 Integration of David Prasser's Routines

This involved taking David's routines and implementing them in a functional manner on the vision board to enable object detection in real time. This was the primary task in realising a fully functional real time tracking system.

3.2.5 Extensions to Vdebug

Testing and debugging of image routines on the SH4 is largely a 'blind' process that is inherently difficult. Very early on it was realised that efficient progress in this direction would require extensions built in to Vdebug to enable processing and testing of routines offline.

3.2.6 Exploration of Visual and Inertial Fusion

The problem of determining the robot's pose had become an issue in the previous year when investigating the problem of balance for the humanoid robot. Currently GuRoo estimates a position of the head through a cumulative calculation of joint angles (ankle, knee, waist). This often transpired to a large error in the determination of GuRoo's head which had an adverse affect on the control loops.

Using the newly installed IMU in conjunction with the vision system has recently become a fairly important issue in robotic vision and the process of fusing the data from both sources in an intelligent way can lead to a far better estimation of robot location and pose than by using either source alone.

This task had originally been the primary focus of the thesis on the assumption that integrating the object detection code with the current system was to be a trivial operation (which unfortunately did not turn out to be the case).

3.2.7 Documentation

In previous years Mark Chang (designer of the vision board and creator for most of the software) had been at hand to provide advice with regards to both software and hardware involved in the GuRoo vision system.

He has since moved on and the lack of documentation has often slowed progress in certain areas. With the continual nature of the project, documentation of methods and code has become a high priority. This thesis will hopefully help fill that void.

3.3 Progression

The tasks listed above were not tackled in a linear fashion. Some were not realised until problems had been uncovered and quickly became a priority. The GuRoo was also not available for the first two months (during this time the GuRoo was in residence at CSIRO) and so the first few weeks were used in exploring different approaches used for visual and data fusion.

Nevertheless, I've addressed the issues in this thesis in this particular order so that it will make logical sense to a reader continuing development on the GuRoo's vision system in future years. Visual and Inertial fusion can only logically be addressed once we have a functional vision system that can perform object detection, which, in turn can only be addressed once software and hardware problems are solved.

To this end, I have decided to leave the results of a literature review and investigation into visual and inertial data fusion until Chapter 8 so that ideas and concepts do not get intermingled between the comparatively separate tasks involved in getting the vision system to detect objects within its field of view in real time.

Chapter 4

Cross-Platform Programming Environment

The aims of this chapter (and chapters 5 and 6) are twofold. Most importantly they document the progress made in the continuing development of the vision system. They also aim to provide suitable documentation for students continuing the project in future years (something that was greatly missed this year). Without such documentation, alot of time was spent familiarising myself with the system and configuring a suitable environment.

4.1 Introduction

The GuRoo vision programs (VBSH4 and VDebug) were originally written and compiled on the Linux platform. With the decision to move the programming environment to Windows, several changes were needed.

It is important to realise immediately that it is not as straight forward as compiling the programs in Microsoft Visual C++, the environment most students would be familiar with. This is due to the lack of a cross-compiler and the inability to manage the project with makefiles - both of which will be explained in the following sections.

4.2 Managing the Project

Most projects created in IDE's (such as MSVC++) automatically manage the code for you. They typically use a system that manages your project in a way keeps the process transparent from the programmer. This has the advantage of reducing the level of complexity in a large project at the cost of portability (to different IDE's) and the ability to customise the way in which the program is managed.

Makefiles are an alternative way of managing programming projects (they can also be used to manage any other style of computing project you have also - it doesn't have to be a programming project). They simply supply a list of instructions and inference rules that may be collected together and executed by the current shell. Typically they provide rules for compiling, linking instructions, pointers to libraries and commands for general file management. They may be auto-generated to manage your project by your IDE or programming environment or they may be hand-written to manage a project that can span several applications at once where all that is needed to manage the project is access to a shell.

The original code written by Mark Chang utilises Makefiles to manage the GuRoo vision system code (VBSH4 and VDebug) and although VDebug was created as a separate project within MSVC++ last year, the same is not possible of VBSH4. Firstly a cross-compiler is needed (as mentioned above) and secondly the Makefiles for VBSH4 contain a very customised set of instructions that would prove difficult if not impossible to translate into an auto-managed project. In addition, the using makefiles provides us with the advantages of being able to transport the code between different IDE's with ease and the flexibility in being able to customise the makefiles as we wish. Learning the fundamental principles of using makefiles is not overly complex and should not take more than a couple of hours. I used a particularly good introduction in Chapter 5 of [5]. Alternatively there are many tutorials on the net that should prove instructive when looked at in conjunction with the makefiles in the VBSH4 and vdebug code directories on the accompanying CD.

On a linux platform, an environment suitable for executing makefiles is automatically provided by default, the only difficulty lies with windows where such an environment must be installed and configured.

4.3 MinGW/MSYS

MinGW provides a minimalistic runtime environment for the GNU gcc copmilers. It compiles and links code to be run on win32 platforms, provides several fundamental libraries and utilises the Microsoft runtime libraries.

MSYS is a minimal POSIX and shell environment for use with MinGW. It provides several shell utilities as well as the standard packages that allow use of Makefiles to manage your project. It is a recent fork of the more traditional Cygwin environment which provides a full POSIX layer and many more utilities for the win32 environment.

The MinGW/MSYS was chosen in preference to setting up the full Cygwin environment simply for its ease in installation and configuration. The home page for MinGW and MinSYS can be found at [10].

4.3.1 Installation

Installers for MinGW and MSYS can be found on the accompanying CD:

"GurooCD://vision-material/programs/MinGW-3.1.0-1.exe"
"GurooCD://vision-material/programs/MSYS-1.0.10.exe"

Installation instructions:

- Install MinGW-3.1.0-1.exe to C:\mingw.
- Install to MSYS-1.0.10.exe to C:\msys.
- Whilst installing MSYS allow it to recognize your MinGW directory.
- Go to Control Panel→System→Advanced→Environment Variables, edit the user's path variable and append to it "c:\mingw\bin;c:\msys\bin"

To utilise the shell environment simply open the shortcut provided with MSYS or open a command prompt within windows.

4.4 Compilers

Compiling programs and creating binaries (executables) is dependant on both the **host** platform the program is compiled on and the **target** processor the program is expected to run on. In most cases both host and target are equivalent.

For example, a MSVC++ compiled program is designed with host and target being the win32 platform/processor. No other option is possible, hence alternative solutions and compilers must be found.

When host and target are different, the process is referred to as **cross-compiling**. For each combination of host/target pair, a different compiler is required. For our purposes, we wish to be able to compile VBSH4 and VDebug under the following situations:

	Host	**Target**
VBSH4	Windows	SH4
	Linux	SH4
VDebug	Windows	Windows
	Linux	Linux

The GNU Toolchain provides a standard set of compilers and cross-compilers that can be utilised across several platforms. Generating the binaries for these compilers can be a difficult process with the plethora of options. There are however several pre-made binaries available in a form ready for us to use with the vision project.

gcc	: Linux compiler. Installed by default on linux platforms.
gcc	: Windows compiler. Installed with the MinGW environment.
sh4-linux-gcc	: Linux-SH4 cross-compiler. Installed separately.
sh4-elf-gcc	: Windows-SH4 cross-compiler. Installed separately [8].

The linux and windows cross-compiler binaries have been placed within the GuRoo project directories (*"GuRooCD://vision-material/software/guroo_stonier/cross-sh4"*) and are already referenced correctly by the project Makefiles.

4.4.1 Installation

If it is necessary to re-install the windows cross-compiler, simply run the installer in

"GurooCD://vision-material/programs/cross-sh4-win/gnushv0402.exe"

pointing the installer to

"Guroo://vision-material/software/guroo_stonier/cross-sh4/sh4-win/".

4.5 OpenGL and GLUT

VDebug makes use of the OpenGL libraries for rendering images. The OpenGL libraries should already be installed on whatever platform is being used if current drivers for the video card have been installed (if this isn't the case the MESA libraries are an option, but it hasn't been necessary to explore this option yet).

By default however, glut is not installed. Glut is the GL Utility Toolkit. It is a cross-platform window management library that provides an extra layer to make the process of handling window operations identical on any platform the glut libraries are available for.

4.5.1 Installation

RPM's for linux and Windows installers (zipped) can be found in

"GurooCD://vision-material/programs/glut/"

Simply unpack the RPM's on linux. On windows unzip the relevant zip file depending on whether you are using MinGW (gcc) or MSVC++ to compile VDebug.

- Copy the dll into c:\windows\system32.
- Copy the lib into c:\(path to compiler)\libs
- Copy the headers into c:\(path to compiler)\include

4.6 Integrated Development Environment

While it is possible to execute the Makefiles to compile VBSH4 and VDebug from the command line, the project is much more easily managed from an IDE. I found IBM's Eclipse with the CDT (C Development Tools) provided a useful development environment for both VBSH4 and VDebug on both Linux and Windows. Useful information for both Eclipse and the CDT can be found at [3] and [4] respectively.

4.6.1 Installation

Java

Eclipse requires the Java Runtime Environment. On the university's computer's this is presently installed with their images. If it isn't, an rpm (linux) and installer (windows) can be obtained from Sun's websites or alternatively use those found in

"GurooCD://vision-material/programs/java/"

Eclipse

Zipped versions of eclipse for linux and windows can be found in

"GurooCD://vision-material/programs/eclipse/zips/"

Simply unzip the file to the directory of your choice. The executable is located in the first directory down.

CDT

Similarly, zipped versions of eclipse for linux and windows can be found in

"GurooCD://vision-material/programs/eclipse/cdt/"

Unzip in the same place as eclipse was placed in, overwriting files if necessary.

4.6.2 Using Eclipse

In windows, eclipse must be started from the command prompt to obtain the full features made available by the MinGW/MSYS environment.

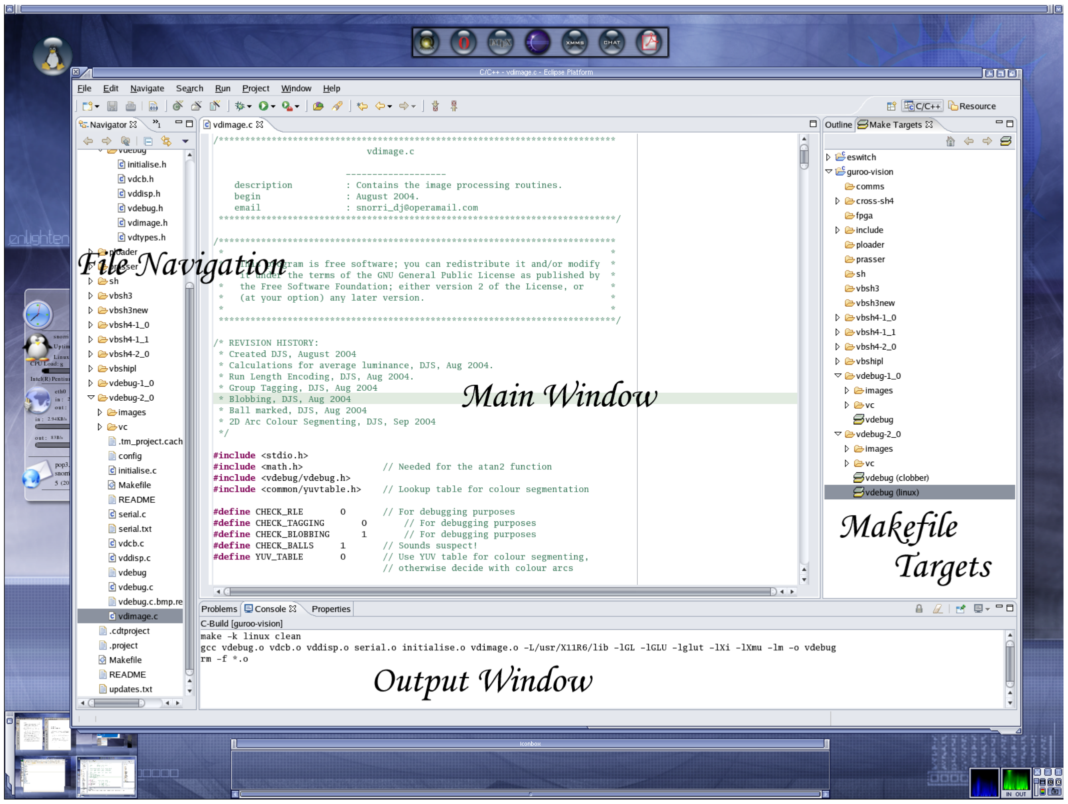
Eclipse uses *perspectives* to construct the appropriate window environment for the language you are programming in. Subsequently the first thing to do after installing is to switch to the C/C++ perspective (Window→Open Perspective).

The GuRoo project file (containing both VBSH4 and VDebug) is stored in

"GurooCD://vision-material/software/guroo_stonier/.cdtproject"

Simply select File→Import and browse to the directory containing the .cdt-project file.

Since we are not working with an automatically managed project, you will also need to turn off the option for automatic builds (Project→Build Automatically). In windows you may also need to turn off Project Indexing (Project→Properties) to prevent crashing. I haven't used it in Windows extensively and am not sure what causes this yet. Compiling simply involves



```
/***************************************************************************
                              vdimage.c

                          -------------------
    description          : Contains the image processing routines.
    begin                : August 2004.
    email                : snorri_dj@operamail.com
 ***************************************************************************/

/***************************************************************************
 *                                                                         *
 *   This program is free software; you can redistribute it and/or modify  *
 *   it under the terms of the GNU General Public License as published by  *
 *   the Free Software Foundation; either version 2 of the License, or     *
 *   (at your option) any later version.                                   *
 *                                                                         *
 ***************************************************************************/

/* REVISION HISTORY:
 * Created DJS, August 2004
 * Calculations for average luminance, DJS, Aug 2004.
 * Run Length Encoding, DJS, Aug 2004.
 * Group Tagging, DJS, Aug 2004
 * Blobbing, DJS, Aug 2004
 * Ball marked, DJS, Aug 2004
 * 2D Arc Colour Segmenting, DJS, Sep 2004
 */

#include <stdio.h>
#include <math.h>               // Needed for the atan2 function
#include <vdebug/vdebug.h>
#include <common/yuvtable.h>    // Lookup table for colour segmentation

#define CHECK_RLE       0       // For debugging purposes
#define CHECK_TAGGING       0       // For debugging purposes
#define CHECK_BLOBBING      1       // For debugging purposes
#define CHECK_BALLS     1       // Sounds suspect!
#define YUV_TABLE       0       // Use YUV table for colour segmenting,
                                // otherwise decide with colour arcs
```

C-Build [guroo-vision]

```
make -k linux clean
gcc vdebug.o vdcb.o vddisp.o serial.o initialise.o vdimage.o -L/usr/X11R6/lib -lGL -lGLU -lglut -lXi -lXmu -lm -o vdebug
rm -f *.o
```

Figure 4.1: Eclipse

right clicking on the appropriate makefile target in the right hand window (note 'make clean' removes object files and 'make clobber' removes objects and executables).

Once compiled VBSH4 is ready to be loaded onto the vision board while VDebug can run from either the command line or by setting up a 'run' in the drop down menu within Eclipse. Different runs can be made by passing different command-line options in its setup.

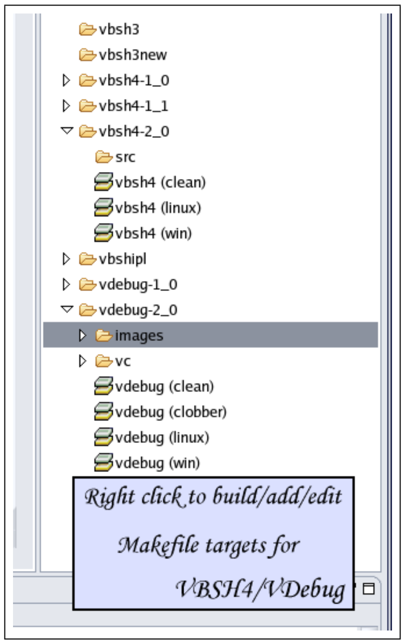


Figure 4.2: Compiling VDebug/VBSH4

Chapter 5

Vision Debugging Software

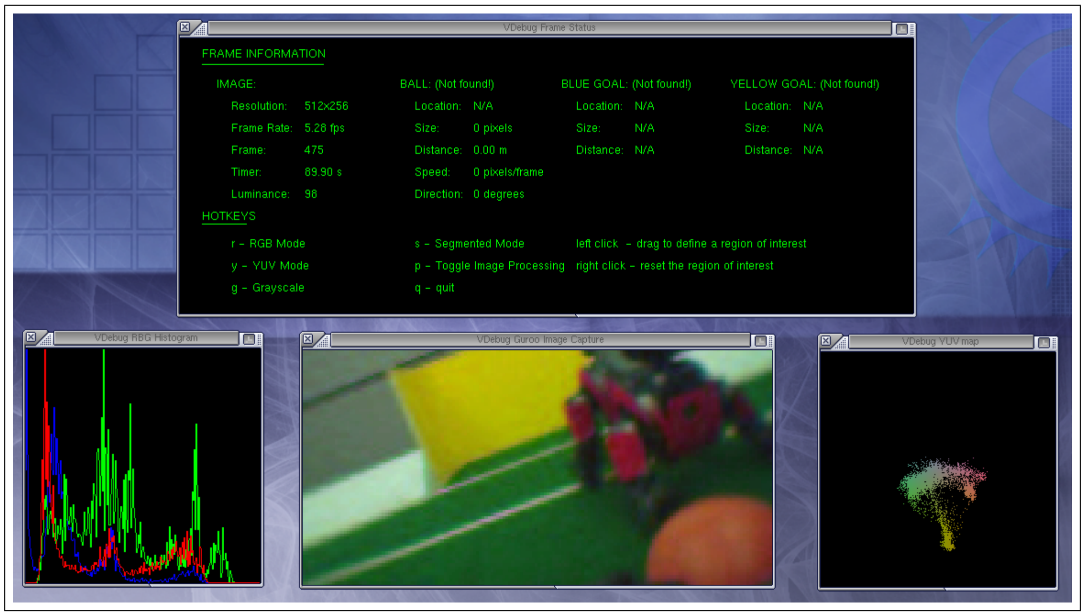
5.1 VDebug - Overview

VDebug's purpose is to provide a platform upon which images retrieved from GuRoo's camera can be retrieved and analysed, as well as providing an application upon which image processing routines can be tested offline.

Initially VDebug's sole mode of operation was to retrieve images from the vision board via a serial connection (very slow transfer ≈ 0.2Hz) and as noted earlier, many of its features were buggy. Throughout the year these features were recoded and various extensions added to make further testing as convenient as possible. The remainder of this chapter will proceed to document the various changes and additions added to the VDebug application.

5.2 Callbacks

With the late addition of the status window (top window - Figure 5.1) in 2003, keyboard and mouse callbacks had been broken. This was a minor fix, but essential to enable switching between modes of operation.



```
FRAME INFORMATION

  IMAGE:                        BALL: (Not found!)            BLUE GOAL: (Not found!)     YELLOW GOAL: (Not found!)
    Resolution:   512x256         Location:   N/A               Location:   N/A             Location:   N/A
    Frame Rate:   5.28 fps        Size:       0 pixels          Size:       N/A             Size:       N/A
    Frame:        475             Distance:   0.00 m            Distance:   N/A             Distance:   N/A
    Timer:        89.90 s         Speed:      0 pixels/frame
    Luminance:    98              Direction:  0 degrees
HOTKEYS
    r - RGB Mode                  s - Segmented Mode           left click  - drag to define a region of interest
    y - YUV Mode                  p - Toggle Image Processing  right click - reset the region of interest
    g - Grayscale                 q - quit
```

Figure 5.1: VDebug

5.3 Zooming

An attempt to define a zoom factor for the rendered image last year failed to work properly. It was also defined in the header files so zoom changes could not be affected without a recompile. Even recompilation would not properly allow a change in the zoom level without needing subsequent changes in the window size definitions. I decided to implement extra routines to allow command line parsing and added an option for passing the executable a zoom factor. The zooming function itself was recoded to allow OpenGL and GLUT to do the appropriate scaling/smoothing of the texture (the rendered image) and resizing of the rendering window.

5.4 Region of Interest Definitions

A region of interest can be specified in VDebug by using the mouse to draw a rectangle on the rendered image. This then causes the histogram and UVmap routines to focus solely on the defined region. The addition of the zoom factor in the previous year broke the relative co-ordinates the callbacks were using to specify the region - these also had to be recoded.

5.5 UVMaps

U and V definitions for red and blue chrominance (as defined in the camera manual [11] are opposite to the usual definitions (ordinarily U is blue and V is red). This meant the UVMap window was actually displaying a VU map. This was corrected and now displays an accurate reconstruction of a UV map for the camera.

5.6 Improving the Image Quality

Since we did not have a means of verifying that the image seen rendered by VDebug was actually what the camera was viewing, there was no means of determining whether the initial lack of image quality was being caused by camera configuration or by rendering problems within VDebug. Checking the problem at the camera level whilst unsure of the rendered map by VDebug was not a logical option, so VDebug was investigated first. Two primary issues concerning VDebug's coding arose.

5.6.1 Pixellation

Whilst correcting the zoom levels within the code it was found that each pixel in the image was being copied and pasted four times within the rendered image (effectively creating a crude 2x stretching of the actual image). As a result, the image size of 256x64 that previous theses had claimed to be receiving was in actual fact only 128x32. The intentions behind the process were not obvious and appeared to serve no purpose. Consequently the process was removed and VDebug was reconfigured to render the actual 128x32 image and the zooming factor allowed resizing of the image using the smoothing applied by the OpenGL routines. As expected, this removed the heavy pixellation originally seen within the image.

5.6.2 Static Image Rendering Modules

Pixel data from images received from the vision board comes in YUV format. VDebug has to then apply YUV→RGB transformations so that the OpenGL libraries can render it as a texture. There were several commented transformations in the code, so the first check was to ensure the correct transformations were being used. To do this however required having VDebug render a known image so a definitive comparison could be mad.

To implement this, several single colour pictures were scripted in plain ascii in a format closely resembling the information received by the application. A module was added that would use these static files as input rather than the serial connection and the resulting image displayed. The end result produced images that were considerably duller (by approximately 20%) than the original YUV colours specified in the ascii files. This implied the transformations were not extending across the full spectrum of RGB values.

YUV↔RGB Transformations

A review of the formulas involved in YUV↔RGB transformations revealed a lack of any standard definition. Most cameras supply pixel data in *RGB* format, but for those that supply data in YUV format (as is the case for this project) the transformations are dependant to a small degree on the algorithms used in their respective cameras. The Omni Vision manual [11], supplies the following transformation for its camera:

$$\begin{aligned} Y &= 0.59G + 0.31R + 0.11B, \\ U &= R - Y, \\ V &= B - Y. \end{aligned}$$

This yields the relative proportions required in the transformation however the data supplied by the camera to the vision board provides YUV information calibrated within a 16-240 range (in fact it does not explicitly state this in the manual, though it does note that RGB and YCrCb data is calibrated to this range and testing has so far not proven inaccurate to any great degree with regards to this). Consequently translation and scaling operations on the above transformations given RGB and YUV values in the 16-240 range yield

$$\begin{aligned}
Y &= 0.59G + 0.31R + 0.11B,\\
U &= (R - Y)/(1.38) + 128,\\
V &= (B - Y)/(1.88) + 128,
\end{aligned}$$

The corresponding inverse transformations must accept the incoming YUV data in the 0-240 range and provide RGB data to the OpenGL texturing algorithm in the 0-255 range. These are given by

$$\begin{aligned}
R &= (255/224) * [(Y - 16) + (U - 128) * 1.38],\\
B &= (255/224) * [(Y - 16) + (V - 128) * 1.38],\\
G &= (255/224) * [(Y - 16) - 0.31 * R - 0.11 * B].
\end{aligned}$$

Small errors were involved in the process of rounding integers that occasionally output values marginally beyond the 0-255 range so a check was added to limit lower and upper values to 0 and 255 respectively.

With these transformations, the test image displayed correctly and the images received from the camera showed a significant improvement in colour intensity.

Ascii and Bitmap File Readers

Whilst developing the module to read in ascii files the, usefulness of having an offline image reader became apparent, particularly in the light of employing, testing and debugging the image routines in the near future. Subsequently the ascii image reader module was formally integrated into the program, a bitmap reader was also added and command line arguments to enable both modes of operation also included.

5.7 Image Processing Routines

The following routines were added to Vdebug's functionality:

- Average Luminance Calculation
- Colour Segmentation
- Run Length Encoding
- Blobbing
- Cursory Ball Detection

5.7.1 Average Luminance

These calculations were included as a permanent feature. The current calculation is displayed in the Status window.

5.7.2 Colour Segmentation

The colour segmentation routines were initially trialled using a simple UV lookup table that was utilised for any Y value. After running into memory problems on the SH4 at a later stage, a pie slicing method was implemented and retained to compare processing results on the SH4 with VDebug's processing routines. An illustration of the segmentation of a standard UV gradient can be seen in Figure 5.7.2.

The gray region in the middle was classified using simple if then calls with minimum and maximum U, V bounds. The slices in the four quadrants were initially classified by angle, but later classified by linear equations to remove the cost in computing trigonometric angles (in addition, the arithmetic on the SH4 cannot use floating point values, so VDebug's algorithms were switched to integer calculations to match). For example, classifying the yellow region pictured uses the following rule:

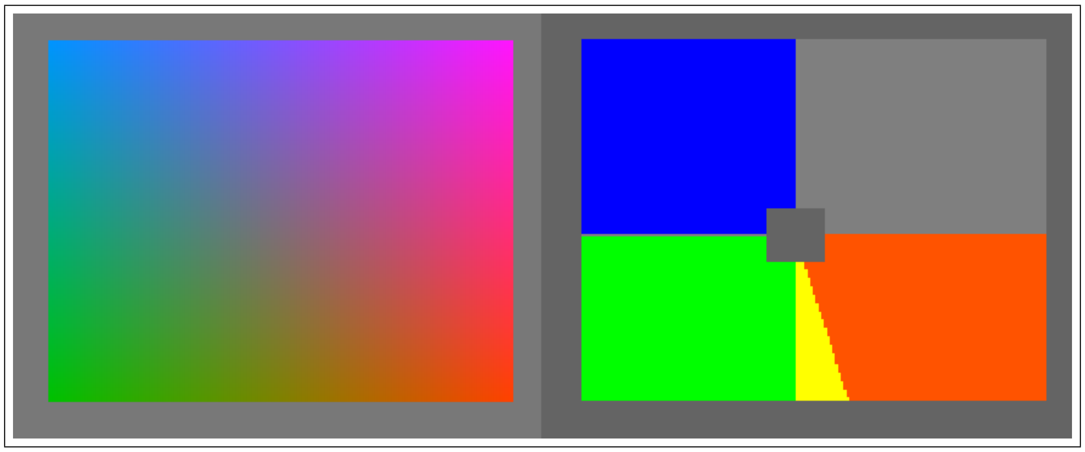


Figure 5.2: Pie Slicing the Standard (Y = 128) UV Gradient

```
if( (V + 3*U < 0) && ( U > 0 ) ) then
  colour = COLOUR_YELLOW
elseif ...
```

Colour segmentation was implemented in VDebug as a separate mode of viewing (along with RGB/YUV modes). An rendering of a segmented image using the slicing algorithm can be seen in Figure 5.7.2.

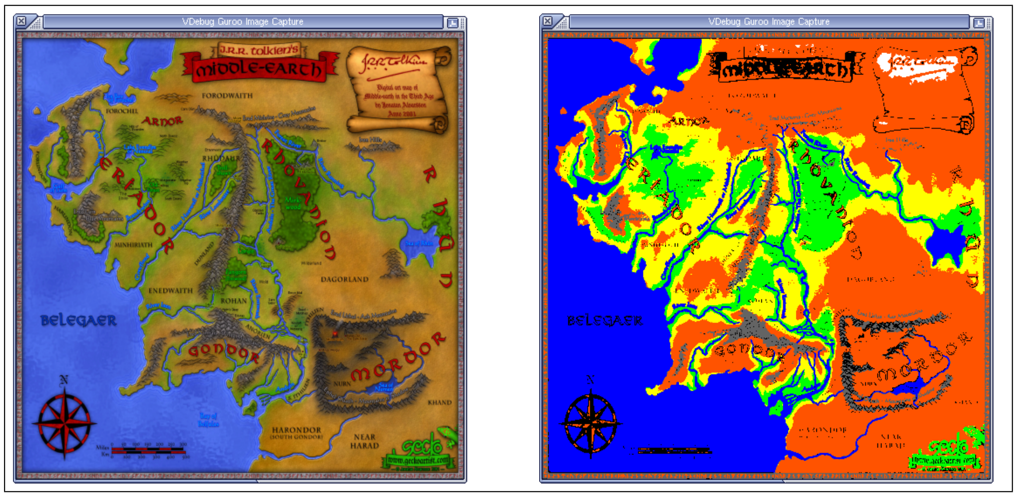


Figure 5.3: Colour Segmentation Results

5.7.3 Run Length Encoding/Blobbing

Some bugs were found in the run length encoding algorithm (problems with ordering of tags and discarding of run lengths encompassing entire rows) and were resolved. Otherwise the run length encoding and blobbing worked with few difficulties.

5.7.4 Ball Detection

A very simple (and temporary) algorithm for finding the ball was implemented at this early stage. It simply searches the blob list for the first available blob of a previously specified minimum size with the correct colour attribute. Object co-ordinates are saved and continually updated in the status window and a cross-hair is placed over the rendered image at the location of the specified object. Image processing (RLEncoding, blobbing and ball detection) can be toggled on and off by pressing 'p' within VDebug.

5.8 Noimage, Nodata and Default Modes

Once ball detection procedures had been implemented, three new modes were added to the execution of VDebug. These ran VDebug in serial mode (in connection with the vision board) requesting periodically both image and ball data (default mode), ball data only (noimage mode) or image data only (nodata mode).

- **Default mode** is useful for general debugging.
- **Noimage mode** allows for demonstration of the real-time tracking of objects in the robot's field of view and illustrates the information the iPaq will theoretically receive.
- **Nodata mode** is expected to be useful for testing image processing routines newly implemented on Vdebug that have not yet made it to the vision board. This allows VDebug to perform all processing operations whilst only requiring image transfer from the vision board.

All three modes may be specified on the command line.

5.9 Command Line Arguments for VDebug

The following is simply a reference for the command line options now passable to VDebug.

–help	Help menu (this list).
–text <filename>	Retrieve data from a text file (.yuv format)
–bmp <filename>	Retrieve data from a bitmap (.bmp format)
–zoom <1-9>	Zoom level. Take care this is not too large.
–process	Process image for blob analysis (default).
–noimage	Retrieve ball data only from the serial connection.
–nodata	Retrieve image only from the serial connection.
–both	Retrieve image and ball data from the serial connection (default).

5.10 VDebug Code Road Map

All code relevant to building VDebug may be found on the accompanying CD in

"GurooCD://vision-material/software/guroo_stonier/".

The relevant directories and files pertaining to the VDebug application are as follows:

- **vdebug-2_0**
  - **vc**: MSVC++ project files for VDebug,
  - **images**: Test images.
  - **vdebug.c**: Contains main().
  - **initialise.c**: Contains argument processing, variable and window initialisations.
  - **vddisp.c**: Window display routines.
  - **vdcb.c**: Callbacks and glut idle loop.
  - **vdimage.c**: Image Processing Routines.
  - **serial.c**: Serial cable communications routines.
- **include**
  - **common**: Header files common to vbsh4 and vdebug (note img-types.h).
  - **vdebug**: VDebug header files.

Chapter 6

The Vision Software

6.1 Introduction

VBSH4 is the software that runs on the SH4 processor and controls the variety of components on the vision board. Its projected tasks include:

- Camera initialisation and configuration.
- Control the image acquisition process.
- Perform image processing on the images as they are received.
- Extract the relevant data from image processing (ball location, size, robot location).
- Transfer images/data to VDebug for processing.
- Transfer data to the iPaq for integration with information from GuRoo's other subsystems.

As noted in Section 2.2.1, VBSH4 initially performed camera initialisation and handled the image acquisition process. It could also pass along images when requests were made from VDebug via a serial connection. The rest of this chapter aims to document the improvements made to VBSH4 as well as providing a guide to the software mechanisms used in VBSH4.

6.2 Loading VBSH4 onto the Vision Board

Unfortunately at this stage, VBSH4 must be loaded onto the vision board manually each time it is powered up. Details illustrating this process and can be found in [12].

6.3 Cross-Compilers

Since VBSH4 is targeted for the SH4 processor, it must be compiled using either the windows or linux cross-compilers. Compiling involves pre-processing of the .S instruction files, compilation and linking. Instructions for these are provided in the makefiles, so all that is needed is to make the appropriate target (either from command line or by specifying one of the targets in Eclipse).

The linux cross-compiler has previously been the compiler of use, however some issues arose with the new windows cross-compiler.

6.3.1 SH3 and SH4 Target Flags

The cross-compiler can be targeted at any one of the SH processor family. Initially GuRoo's code had been written for use with an SH3 processor, then later recoded for use with the SH4. SH3 code was left intact, and compilation for routines designed for one or the other was dependant on whether the __SH3__ or __SH4__ macro had been passed to the compiler.

Unfortunately a known bug with the windows cross-compiler causes the __SH3__ macro to remain permanently on for all SH targets. This caused compilation of the SH3 rather than the SH4 code. The #if directives were recoded to avoid using the __SH3__ macro.

The SH3 code could be removed - at this stage it is currently redundant as we are no longer working with the SH3.

6.3.2 Linking and Symbol Recognition

The windows cross-compiler uses a different output format and also differs when recognising symbols associated with the programs functions. This required some editing of the pre-processing files leading to a split in the make process. Separate makefiles and pre-processing files were written for both linux and windows compilations, whilst the source code remained identical for both.

Compilation can be initiated on the command line with either "make win" or "make linux", or alternatively by selecting the appropriate target within Eclipse.

6.4 Memory Usage

Once the image processing routines were included in VBSH4, free memory in the 512K SRAM became scarce. The following list represent the difficulties that arose as well as providing some direction for future students working with the vision board.

6.4.1 Memory Allocation

Memory on the SRAM is partitioned between memory reserved for the program, variables and data. Allocations are defined in **vbsh4-<host>-lds.S** and variables in **sram.h**.

There was initially approximately 256K allocated for the program itself and another 256K for the program variables. Since the actual program is relatively small ($\approx 20K$) 128K was borrowed from program space and allocated to variable space. This was be done by redefining the address that the RAM_PTR points to in sram.h.

Once the extra memory was reserved, the SH4 was able to process images with a resolution of 128x128.

6.4.2 Lookup Tables

With little room on the SRAM, lookup tables were skipped for the present and the simpler pie slicing algorithm used. There is the option of placing the lookup tables on SDRAM (although slower). This will be discussed further in Chapter 7.

6.5 Camera Configuration

There are a host of configuration options that may be set and tuned for the camera in **ov7620.c**. Care needs to be taken when adjusting these parameters, particularly with anything that redefines the image size as handling the images in **main.c** requires explicit information from the programmer as to how large the incoming images are.

If changing viewport settings, image resolution, interlacing or subsampling - changes to IMG_XSIZE and IMG_YSIZE (defined in ***imgtypes.h****) must also be made!*

6.5.1 Automatic Routines

The camera has many adjustable routines (automatic exposure, gain and white/black pixel ratios) that can be reconfigured to provide varied image quality in return. These were experimented with in a cursory fashion.

Adjusting the white/black pixel ratios were found to dramatically affect the brightness of the image once the automatic routines had settled. A filter for indoor settings was also enabled providing a slight improvement to image quality.

6.5.2 Known Issues

The other aspect of configuration applies to image dimensions and setting the viewport. Currently there is an ongoing problem with the dimensions being received from the camera. The horizontal dimension is being subsampled, although retaining its full field of view. In progressive scan mode (retrieval of every vertical field) this causes the image to appear stretched. Current settings use interlaced mode (alternately odd and even fields are scanned) and process each alternate image as though it was a separate image. This is only a temporary solution and ideally needs to be solved.

6.6 Speed

Real-time tracking of a ball was achieved very late in the project. Several tests were made with an image at a resolution of 128x128. The first test used vbsh4 compiled solely for image acquisition, the latter for vbsh4 compiled for image acquisition and image processing (segmenting, RLE, blobbing and data extraction).

	Hz	**Processing Time/Frame**
Image Acquisition Mode	2.6	N/A
Image Processing Mode	2.5	$\approx$ 25ms

This indicates a problem with the image acquisition mode. The camera is rated at approximately 60Hz. Download to the vision board will consume some time, however the frequency at which it is running is far under what we had expected. Unfortunately there was no time to investigate this issue.

6.7 VBSH4 Code Road Map

All code relevant to building VBSH4 may be found on the accompanying CD in

"GurooCD://vision-material/software/guroo_stonier/".

The relevant directories and files pertaining to the VBSH4 are as follows:

- **cross-sh4**: Location of cross-compiler binaries.
- **include**
  - **common**: Header files common to vbsh4 and vdebug (note img-types.h).
  - **sh**: SH4 header files.
  - **vbsh4**: VBSH4 header files.
- **sh**: Contains routines specific to the SH4 processor.
  - **gio.c**: Vision board LED routines (used for output signals).
  - **intc.c**: SH4 interrupt handler.
  - **tmu.c**: SH4 Timer Unit.
  - **dmac.c**: SH4 Dmac.
  - **sci.c**: SH4 Serial Communication Interface routines.
  - **rtc.c**: SH4 Timer Clock.
- **ploader**: Source code for the program that loads vbsh4.bin onto the SH4.
- **vbsh4-2_0**
  - **src**
    - **main.c**: Primary File.
    - **isr.c**: Interrupt Service Routines.

    * **ov7620.c**: Camera configuration and camera interrupt routines.
    * **vbimage.c**: Image processing routines.
  - **vbsh4.bin**: VBSH4 binary (executable).
  - **ploader.exe**: Loads vbsh4.bin onto the SH4 via serial connection.
  - **entry.S**: Interrupt definition file (pre-processed).
  - **vbsh4-lds.S**: Defines memory layout on the vision board (pre-processed).

Chapter 7

Future Work

The vision system is now in a functional state for real time tracking. Unfortunately this was only reached late in the semester and optimisations could not be implemented to improve its performance. This is an area which is critical since both memory and speed of performance are a high priority. The following issues are immediate areas of interest with regards to performance.

7.1 Camera Configuration

A systematic analysis of the possible configuration modes for the camera could do alot to improve image quality. I do not think this has been done in any depth yet - initial configuration appears simply to have been directed towards obtaining a functional system.

7.2 Image Acquisition

Transfer rate from the camera to the control board is currently slow. An investigation into the cause of this is necessary to enable a worthwhile processing rate (>5Hz). There is the possibility that this is being caused by the way VBSH4 handles incoming images from the camera.

7.3 Image Analysis

7.3.1 Lookup Tables

Lookup tables are generally the preferred method for colour segmentation. They are customisable and fast. With a lack of memory for a suitably sized table on SRAM, it may be worth considering placing the lookup table on the 16MB of SDRAM. The SDRAM however is significantly slower - tests should be made to determine if using a lookup table here is more beneficial than using alternative methods akin as the pie slicing currently implemented.

If the pie slicing method is retained (or something similar) then it should be extended to cater for varying ranges in luminance. That is, it needs a tree-based branching of decisions to cater for segmentation on more than a single UV map. This is particularly important in the classification between yellow and orange values where yellow only clearly exists on a UV map at high luminance (Y).

7.3.2 Morphology

To date there has not been a systematic analysis of the effects of various morphological routines. This would be useful to implement and optimise for the most useful approach as the camera's image is sometimes speckled in nature and has difficulties in highly specular regions.

7.3.3 Blob Analysis

Currently the blob analysis is a very cursory inspection of the blob lists for a minimally sized blob list. Ultimately this will be better replaced by routines which fit the shape of blobs to ellipses (to detect spherical balls for instance) as well as various rules for discarding erroneous images.

7.3.4 Edge Line Detection

Routines for detecting edge lines will eventually have to be implemented to recognize a playing field. They can also be used as self localisation aids when a lack of landmark features are experienced. David Prasser has more details on this in his thesis [14] and a few introductory routines in his code.

Chapter 8

Visual and Inertial Fusion

8.1 Why Visual and Inertial Fusion?

The need for sensory information is critical to a human's ability to perform any task, trivial or complex. Our primary channel for gathering this information is through our sense of vision. However, as with many human subsystems, the data obtained from our sense of vision is supported by information from a secondary source - our vestibular (inner ear) system.

Both systems have means of obtaining similar data. They are each able to determine a vertical cue (aiding in balance) and to a lesser extent calculate self-motion within the local environment. This appears at first be redundant but by having both systems acting simultaneously it actually provides a very robust solution. Where one sense may at times provide insensible information the other sense will take priority.

For example, our vision system becomes relatively inaccurate at high linear or angular speeds - the images our brain receives become very blurred, yet our vestibular system is able to provide us with a reasonable estimation of our self-motion.

In addition, it is also suspected that the brain fuses the data from both sources in some way - though medical research cannot yet confirm or disprove this yet.

When applying these concepts to robots, the need for fusing information from camera (vision) and inertial measurement unit (vestibular) becomes even greater. A robot's senses, particular the vision sense, is still nowhere near as advanced as a human's. Sample rates are much slower than the human eye, and sensory measurement is in general severely affected by noise.

The other appealing attribute of combining information from vision and inertial sources is due to the fact that their sources of error are complimentary.

- *Visual Data* can only be processed at low speeds, has low uncertainty at low speeds and suffers from low frequency noise.

- *Inertial Data* can be delivered at high speeds, has high uncertainty at low speeds, but low uncertainty at high speeds and suffers primarily from high frequency noise.

8.2 The Inertial Measurement Unit - Review

A typical inertial measurement unit consists of 3 gyrometers and 3 accelerometers accounting for the three axes of rotation and translation.

8.2.1 Vertical Cue

While the linear accelerations experienced by the robot remain relatively small in comparison to that of gravity, the resultant linear acceleration vector can be used as a cue for the vertical. An accurate measurement of the vertical can be useful in determing the robot's pose and to aid in extraction of data from the camera.

8.2.2 Drift

Given these accelerations and an initial position, it is possible to integrate and gain an estimate velocities and positions in time. These estimations however suffer from one important problem - **drift**.

This can be observed by considering the simple problem of accelerating in a linear direction. Sampling results in discrete dynamics for the system given by

$$\mathbf{x}_{k+1} = A\mathbf{x}_k + Bu_k,$$
$$\begin{bmatrix} x_{k+1} \\ \dot{x}_{k+1} \end{bmatrix} = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_k \\ \dot{x}_k \end{bmatrix} + \begin{bmatrix} \Delta t^2/2 \\ \Delta t \end{bmatrix} u_k.$$

where u_k is the sampled acceleration. Simulating with an acceleration of $u(t) = \cos(t)$, yields position and velocity estimates illustrated in Figure 8.2.2.

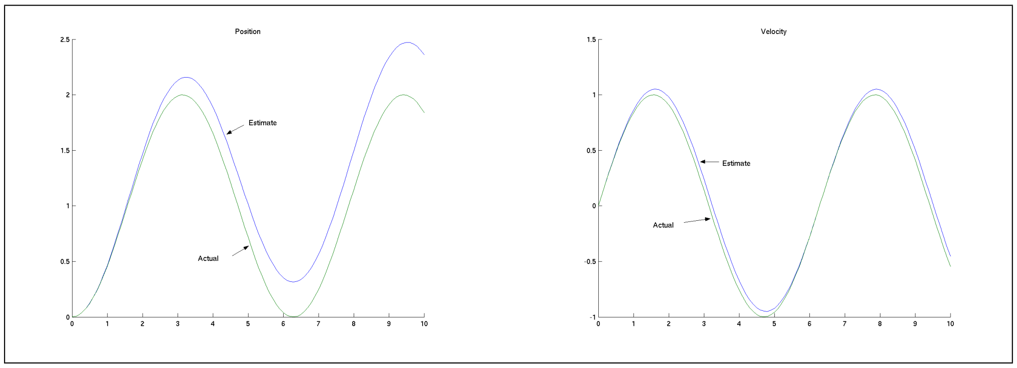


Figure 8.1: Drift in Position and Velocity

In this example the drift is caused by the sampling, however the addition of noise will also create unnecessary drift.

One way to correct the inevitable drift is to realign the position measurement occasionally from an alternate source. This method was predominately used in flight navigation before the advent of GPS.

Rotational accelerations can make use of this process by using the vertical cue obtained from the linear accelerations to realign the rotational co-ordinates. This however only allows correction against drift for two of the angular rotations - the third can be corrected by obtaining measurements from a magnetometer. This process is automatically implemented within CSIRO's inertial measurement unit. Using the same process for the linear accelerations requires obtaining from sources external to the unit itself.

8.3 Vision System - Review

Information extracted from the vision system is given in terms of image co-ordinates. These only become useful in determing object and robot localisation if they can be accurately translated into world co-ordinates.

8.3.1 Perspective Geometry

Perspective transformations provide a convenient approximation for moving between two dimensional image co-ordinates and their corresponding real-world point in three dimensional space. The geometry of the transformation is outlined in Figure 8.2.

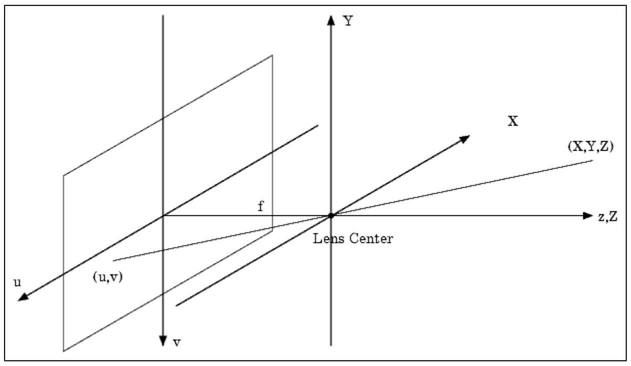


Figure 8.2: Perspective Transformation

Relationships between co-ordinate systems is easily found using similar triangles (f is the focal length),

$$uf\frac{X}{Z} \qquad vf\frac{Y}{Z}. \tag{8.1}$$

where f is the focal length. A more detailed construction can be found in [6]. These are in general the usual approximations used transforming world to image co-ordinates. They remain accurate whilst the object of interest remains close to the perpendicular line (i.e. X, Y remain small compared to Z).

The obvious problem with the above transformation is that image and world co-ordinates do not have a 1-1 relationship. This is exacerbated by the fact that the GuRoo is relying on monocular vision and consequently cannot obtain extra visual informations from another source.

8.3.2 Approximating World Co-Ordinates

Soccer fields provide certain conditions that enable quick approximations to the transforms from image to world co-ordinates.

- *Ground Plane:* Objects of interest such as the soccer ball can be assumed to lie on the ground plane at a specific distance below the level of the camera. This fixes one of the dimensions from which the other two can be computed.
- *Landmarks:* If multiple (known) landmarks can regularly be found within the robot's field of view, triangulation can be used to complete the transformations required to determine the robot's exact position in world co-ordinates.

These approximations are often used to find rough position estimates on a soccer playing robot.

8.4 Fusion Techniques

Recent papers (refer to [7, 9, 1, 16, 17, 18, 19, 20]) provide a wealth of solutions to fusing data in a variety of environments. There seem to be three general themes which may be of significant interest to the humanoid project.

- *3-Dimensional Feature Recovery* using cues from the inertial sensors.
- *Improved Estimation of Spatial Awareness* by fusing initial estimates obtained from both sensor systems through a series of filters or custom made algorithm.

- *Gaze Tracking* using inertial cues to predict movement of objects in the image.

8.4.1 3-Dimensional Feature Recovery

Extracting three-dimensional information from the images will ultimately be essential in assisting the Guroo locate a soccer ball on the playing field as well as identifying its boundaries and any obstacles in its field of vision. If the approximations outlined in Section 8.3.2 prove to be inaccurate (which may be the case if the vertical distance between robot and ground plane does not remain a constant), then alternative methods need to be found.

The problem then is one of observer design. Given image co-ordinates, can the inertial measurements be utilised to provide the extra information needed to estimate the state (object or robot).

This topic is covered in detail in [21]. The system in this state is observable whilst motion occurs transverse to the radial line between camera and object. Consequently a period of scanning in a transverse direction is needed before a control algorithm to approach the target can be made. Refer to Figure 8.4.1 where a) illustrates the transverse movements required to solve the observer problem and b) illustrates movement in a direction lacking the information to solve for depth.

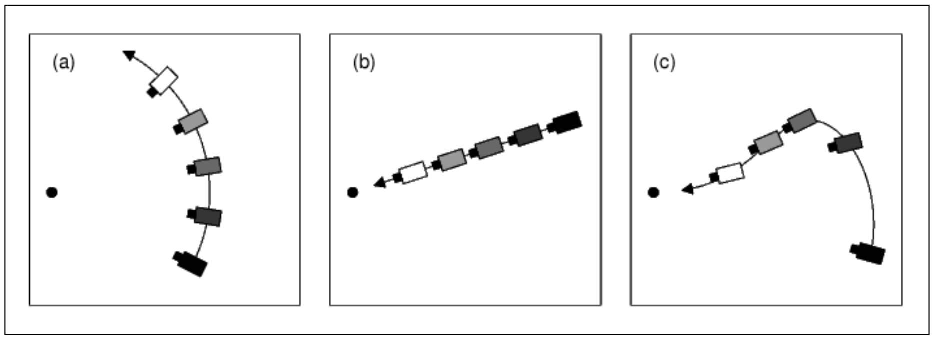


Figure 8.3: Observability Problem for a Monocular Camera

A simple test example was designed in the thesis to perform a systematic analysis of Kalman Filters and Monte Carlo methods used in attempting to solve the problem. The Kalman filter proved inaccurate in dealing with the nonlinear relationship (8.1). Even when the Extended Kalman filter was applied, it required initialisations very close to the actual values otherwise divergence would rapidly take the estimate away from the true state. The rest of the thesis formulates another approach which has been tested on robots at Stanford University and has proven to be quite successful.

Other feature recovery techniques typically involve geometry techniques to achieve various tasks. Lobo and Dias [9, 19, 20] provide some useful examples of ground plane recognition using inertial cues and depth maps recreated from horizontal line detection. These appear to be better suited to offline methods.

8.4.2 Improved Spatial Awareness Estimation

If good approximations to world co-ordinates can be retrieved from the images using the methods in 8.3.2, then the system can be linearised in a manner similar to the simple drift example used earlier without the need for the complex nonlinear observation (8.1). In this situation, it is possible to generate predictive estimates of the new state from both vision and inertial systems and then fuse these estimates to provide an improved estimate.

This can be seen in [17]. Corke generates estimates for velocity by tracking features in the image sequence and prioritising them with a statistical significance. Extraction is achieved by tracking features with obvious corners (typical for the terrain it is needed to work in) through the image sequences to derive estimates for velocities. The end result is fused through a complementary filter with the inertial accelerations to provide an enhanced estimate of velocity.

In [7] inertial and visual navigation procedures are run simultaneously and autonomously. Key landmark features with known absolute co-ordinates are extracted from the images to obtain estimates for position and velocity from the vision system. Rather than using a filter, autonomous estimations are passed from one process to the other to evaluate weighted, *assisted* estimations. The principle relies on the fact that the error sources associated with each process (vision and inertial) are different and independent.

Other techniques adopt a common sense approach to utilising the information from its most reliable source where appropriate - this is illustrated in [1] where multiple contours are tracked in the image sequences. From this, a number of pose estimates is obtained which are more or less effective depending on their alignment with the projection axis. A Kalman filter is used to provide the most optimal estimate from these. The process however is uncertain when there are only small changes in the camera's orientation from one time step to the next. When this is the case, the inertial sensors (which do not have problems at low velocities) are used to provide the missing information.

8.4.3 Gaze Tracking

Not necessarily a fusion technique for better estimation on its own accord, but the technique does fuse the input to provide improvements in the methods used by other utilities. For example, accelerational readings can be used to provide an a reasonably accurate prediction of where objects in the robot's field of view may be at the next sampling point (remembering accelerational information can be delivered at a much faster rate than vision information). This can aid in

- Tracking of objects so they do not leave the field of view from one instant to the next.
- Identifying a reduced search region to speed image processing times.

Conclusions

The vision system for the GuRoo has been developed this year to the point where it can process images and retrieve object data, enabling it to track a simple object such as a ball. It is at the stage where it can now begin to usefully pass information to the other subsystems on the Guroo, however the image processing routines need some refinement to make the system more reliable in an operating environment and to optimise the speed of the routines.

Other developments include an incremental improvement the image quality, at the debugging end as well as at the camera and successful migration to a windows programming environment for the whole vision project - previously a task set for 2003.

Due to the unforeseen obstacles, the initial goal of moving towards fusing inertial and visual data for a better spatial awareness could not begin on a practical side, however the collection of papers and summary provided here should assist in directing future work in the right direction as quickly as possible.

Bibliography

[1] G.Alenya, E.Martinez, C.Torras, *"Fusing Visual and Inertial Sensing to Recover Robot Ego-Motion"*, (printed copy only), Journal of Robotic Systems, Vol. 21, 2004.

[2] J.Roberts, P.Corke, L.Overs *"EiMU - User Manual Version 0.47"*, (printed copy only), CSIRO Manufacturing and Infrastructure Technology, Queensland Centre for Advanced Technologies.

[3] Eclipse, *"Eclipse Home Page"*, (http://www.eclipse.org).

[4] Eclipse, *"C Development Toolkit"*, (http://www.eclipse.org/cdt).

[5] W. Gay, *"Linux Programming"*, (printed copy of chapter 5), Sams Publishing, 1999.

[6] R. Gonzalez, R. Woods, *"Digital Image Processing"*, Addison-Wesley, 1993.

[7] S.Graovac, *"Principles of Fusion of Inertial Navigation and Dynamic Vision"*, (printed copy only), Journal of Robotic Systems, Vol. 21, 2004.

[8] KPIT, *"KPIT Gnu Tools and Support"*, (http://www.kpit.gnutools.com/).

[9] J.Lobo, J.Dias, *"Inertial Sensed Ego-Motion for 3D Vision"*, (printed copy only), Journal of Robotic Systems, Vol. 21, 2004.

[10] MinGW, *"MinGW Home Page"*, (http://www.mingw.org).

[11] OMNI Vision *"OV7620 Manual"*, (GurooCD://vision-materials/documentation/OmniVision OV7620.pdf).

[12] A. Peters, *"VBSH4 - User Guide"*, (GurooCD://vision-materials/guides/VBSH4 - User Guide.pdf), Guroo Manuals, University of Queensland, 2003.

[13] A. Peters, *"Vision Software for Humanoid Robot Soccer"*, (GurooCD://vision-materials/theses/peters.pdf), Undergraduate Thesis, University of Queensland, 2003.

[14] D. Prasser, *"Vision Software for a Humanoid Robot"*, (GurooCD://vision-materials/theses/prasser.pdf), Undergraduate Thesis, University of Queensland, 2001.

[15] P. Thomas, P. Wolfs, R. Stonier, *"Robustness of Colour Detection for Robot Soccer"*, (GurooCD://vision-materials/papers/colour-spaces/slices.pdf), FIRA, 2002.

[16] I.Stratmann, E.Solda, *"Omnidirectional Vision and Inertial Clues for Robot Navigation"*, (printed copy only), Journal of Robotic Systems, Vol. 21, 2004.

[17] P.Corke, *"An Inertial and Visual Sensing System for a Small Autonomous Helicopter"*, (printed copy only), Journal of Robotic Systems, Vol. 21, 2004.

[18] M.Ribo, M.Brandner, A.Pinz, *"A Flexible Software Architecture for Hybrid Tracking"*, (printed copy only), Journal of Robotic Systems, Vol. 21, 2004.

[19] J.Lobo, J.Dias, *"Recovering 3D Structure from Images and Inertial Sensors"*, (printed copy only), Journal of Robotic Systems, Vol. 21, 2004.

[20] J.Lobo, J.Dias, *"Integration of Inertial Information with Vision Towards Robot Autonomy*, (printed copy only), ISIE 97 Guimaraes, Portugal.

[21] A. Huster, *"Relative Position Sensing by Fusing Monocular Vision and Inertial Rate Sensors"*, (GurooCD://vision-material/papers/kalman/estimator-thesis.pdf), PhD Thesis Dissertation, Stanford University, 2004.