Machine Vision as the Primary Sensory Input for Mobile, Autonomous Robots

by

Nathan Lovell
B.E. (Software) (Hons), University of Newcastle, 2001

A thesis submitted in fulfillment of the requirements
for the degree of Doctor of Philosophy

School of Computers and Information Technology
Griffith University
Nathan QLD 4111
Australia

January 2006

Abstract

Image analysis, and its application to sensory input (computer vision) is a fairly mature field, so it is surprising that its techniques are not extensively used in robotic applications. The reason for this is that, traditionally, robots have been used in controlled environments where sophisticated computer vision was not necessary, for example in car manufacturing. As the field of robotics has moved toward providing general purpose robots that must function in the real world, it has become necessary that the robots be provided with robust sensors capable of understanding the complex world around them. However, when researchers apply techniques previously studied in image analysis literature to the field of robotics, several difficult problems emerge.

In this thesis we examine four reasons why it is difficult to apply work in image analysis directly to real-time, general purpose computer vision applications. These are: improvement in the computational complexity of image analysis algorithms, robustness to dynamic and unpredictable visual conditions, independence from domain specific knowledge in object recognition and the development of debugging facilities.

This thesis examines each of these areas making several innovative contributions in each area. We argue that, although each area is distinct, improvement must be made in all four areas before vision will be utilised as the primary sensory input for mobile, autonomous robotic applications.

In the first area, the computational complexity of image analysis algorithms, we note the dependence of a large number of high-level processing routines on a small number of low-level algorithms. Therefore, improvement to a small set of highly utilised algorithms will yield benefits in a large number of applications. In this thesis we examine the common tasks of image segmentation, edge and straight line detection and vectorisation.

In the second area, robustness to dynamic and unpredictable conditions, we examine how vision systems can be made more tolerant to changes of illumination in the visual scene. We examine the classical image segmentation task and present a method for illumination independence that builds on our work from the first area.

The third area is the reliance on domain-specific knowledge in object recognition. Many current systems depend on a large amount of hard-coded domain-specific knowledge to understand the world around them. This makes the system hard to modify, even for slight changes in the environment, and very difficult to apply in a different context entirely. We present an XML-based language, the XML Object Definition (XOD) language, as a solution to this problem. The language is largely descriptive instead of imperative so, instead of describing *how* to locate objects within each image, the developer simply describes the properties of the objects.

The final area is the development of support tools. Vision system programming is extremely difficult because large amounts of data are handled at a very fast rate. If the system is running on an embedded device (such as a robot) then locating defects in the code is a time consuming and frustrating task. Many development-support applications are available for specific applications. We present a general purpose development-support tool for embedded, real-time vision systems.

The primary case study for this research is that of Robotic soccer, in the international RoboCup Four-Legged league. We utilise all of the research of this thesis to provide the first illumination-independent object recognition system for RoboCup. Furthermore we illustrate the flexibility of our system by applying it to several other tasks and to marked changes in the visual environment for RoboCup itself.

Certificate of Originality

I hereby certify that the work embodied in this thesis is the result of original research and has not been submitted for a higher degree at any other University or Institution.

(Signed) 

Nathan Lovell

Approval

Name:	Nathan Lovell
Degree:	Doctor of Philosophy
Thesis Title:	Machine Vision as the Primary Sensory Input for Mobile, Autonomous Robots
Submission Date:	19 January, 2006
Supervisor:	Prof. Vladimir Estivill-Castro Griffith University Australia
Co-supervisor:	Prof. Manuella Veloso Carnegie Mellon University U.S.A.
External examiners:	Prof. Robyn Owens University of Western Australia Australia
	Dr. Thomas Röfer Univerity of Bremen Germany

Acknowledgements

I would like to sincerely thank my supervisor, Prof. Vladimir Estivill-Castro. His assistance, encouragement and support has been exemplary and tireless for many years now.

Thanks are also due to my colleagues in the Machine Intelligence and Pattern Analysis lab at Griffith University for their helpful suggestions on many occasions. Particular thanks to Joel Fenwick for his regular assistance with mathematics and for his valuable contributions to my work on linear-time line analysis.

Finally, I wish to thank those who have supported my wife and me throughout this thesis: our families and our second family at St. John's Anglican Church, Wishart.

To the God who invented vision, whose algorithms we try to imitate.

And to my wife who loves and encourages me.

List of Outcomes Arising from this Thesis

Papers in International Conferences

- V. Estivill-Castro and N. Lovell. Improved object recognition — the RoboCup 4-legged league. In *Proceedings of the 4th International Conference on Intelligent Data Engineering and Automated Learning*, pages 1123-1130. Springer-Verlag, 2003, ISBN: 3-5404-0550-X.

- N. Lovell and V. Estivill-Castro. A descriptive language for flexible and robust object recognition. In *Proceedings of RoboCup 2004 — Robot Soccer World Cup VIII, Lisbon, Portugal*, pages 540-547. Springer-Verlag, 2004, ISBN: 3-5402-5046-8.

- N. Lovell. Real-time embedded vision system development using AIBO Vision Workshop 2. In *Proceedings of the 5th Mexican International Conference on Computer Science*, pages 268-274. IEEE Computer Society, 2004, ISBN: 0-7695-2160-6.

- N. Lovell and J. Fenwick. Linear time construction of vectorial object boundaries. In *Proceedings of the 6th IASTED International Conference on Signals and Image Processing*, pages 914-919. ACTA Press, 2004, ISBN: 0-8898-6434-9/0-8898-6442-X.

- N. Lovell. Illumination independent object recognition. In *Proceedings of RoboCup 2005 — Robot Soccer World Cup IX, Osaka, Japan.* Springer-Verlag, 2006, *To appear.*

- N. Lovell. Fast posture and object recognition using symmetries. In *Proceedings of the 2005 Australasian Conference on Robotics and Automation*, CD-Rom Proceedings, 2005, ISBN: 0-9587-5837-9.

Results from the International RoboCup Competition

- Second place in the open challenge, 2005, (as assessed by peers) for work on a real-time posture recognition system for the AIBO.

- Team places 5^{th} overall in the technical challenges, 2005, and prequalifies for 2006. Our solutions to two of the three challenges were based on my vision system: the open challenge (posture recognition) and the illumination challenge (illumination independent object recognition).

- Participation in competition in RoboCup 2005, Osaka, Japan.

- Participation in competition in RoboCup 2004, Lisbon, Portugal.

- Participation in competition in RoboCup 2003, Padua, Italy.
- Participation in competition in RoboCup 2002, Fukuoka, Japan. The team places 3^{rd} overall in the soccer competition[1].

[1]At this time I was a member of the team from Newcastle University, Australia.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Computer Vision and Mobile, Autonomous Robotics

Image analysis, and its application to sensory input (computer vision) is a fairly mature field, so it is surprising that its techniques are not extensively used in robotic applications. On the surface it seems that mobile, autonomous robotics is an ideal platform for the application of computer vision techniques. Mobile robots are required to operate in the same dynamic real-world environment where vision is the primary sensory input for humans. In fact, so much do humans rely on vision that we often recruit seeing-eye-dogs or the like to do it for people who are visually impaired. Nevertheless, when we apply the algorithms developed for computer vision to robotic applications, several very difficult problems emerge.

Traditionally robots have been used for manufacturing or other repetitive tasks. In this environment, machine vision has rarely been used as a primary sensory input because, where applicable, other sensors are far easier to use. For example, the combination of pressure sensors and joint position sensors is enough to manufacture a car. The robot does not need to use vision to execute a script of motions if it has knowledge of its joint positions with sufficient accuracy and its environment is carefully controlled. Where image analysis has been used, such as verification in circuit design, the camera is generally stationary, in a known lighting condition and with no temporal performance restrictions.

In contrast to this, an autonomous, mobile robot is expected to operate in unknown conditions. These robots must carry everything that they need to

perform their task with them — including the camera. This means that the camera will not be fixed; the image stream will suffer from jolts of motion as the robot moves around. Furthermore, not everything that the robot needs to see will be in the viewing range of the camera at all times. The robot will need to decide what to look at. If it has the equivalent of a head, then the decision of where to point the camera is a strategic decision that is independent of the direction of motion of the robot. Furthermore, mobile autonomous robots need to interact with the real world in a way similar to humans. Not everything in the environment will be controlled. This imposes a temporal restriction on image processing. Whatever algorithms are used, they must keep up with the pace of change in the environment[1] or the robots' actions will lag significantly behind the external stimuli that caused them. These conditions make the application of image processing algorithms a challenging task.

Many researchers expected that solutions to these problems would become evident as the hardware used in the robots became faster and, to a certain extent, this may be true. However, experience has shown that even though hardware is now relatively inexpensive, applications of image analysis for machine vision in mobile robotics are still very limited [72]. It is increasingly apparent that no matter how much faster computers become, robots will never be able to use vision exclusively as a primary sensory input (that is, see like a human) until more groundwork is done in the fundamentals of computer vision to overcome the unique problems associated with mobile, autonomous robotics. There are four distinct problem areas that must be addressed by the computer vision community if visual sensory input for robotics is realised. We detail these areas now.

1.1.1 Computational Complexity of Image Analysis Algorithms

The first problem area is that of algorithmic complexity. The emphasis in image analysis literature has been primarily on code robustness and secondly on efficiency. The reason for this is apparent: an image processing algorithm needs to be extremely robust to noisy conditions. However, this means that there have been many fine algorithms developed for sub-problems within the field of image analysis that are almost useless when applied to the field of mobile robotics

[1] This will usually mean keeping up with the frame rate of the camera.

due to their run-time cost. For example, basic analysis techniques such as edge detection, shape and texture analysis and other feature detection routines are computationally expensive.

The requirement of real-time vision is that the image analysis system be able to process (usually) at least 25 full images per second. Add to this the requirement that there be enough left-over processing cycles to operate the other elements of the robot, and the real-time restriction is quite rigid. It is infeasible to run an edge-detection or texture analysis routine that takes over a second if you are required to completely finish image analysis with time to spare for other tasks in less than $\frac{1}{30}$ of a second. Neither is it sufficient to argue that Moore's law will solve the problem eventually. Digital cameras have also increased in resolution in proportion to Moore's law because the same technology that facilitates the extra transistors on computer hardware also facilitates the extra resolution of digital cameras. At the time of writing mid-range digital cameras permit resolution of around 2 Mega-Pixels (that is, 2 million pixels). Even a modern processor that is capable of 2 G Hz (2 billion cycles per second) will fail to process 30 such images in a second. As the processing speed of computers has increased, so has the image resolution. For each linear increase in processing power we would expect at least an equivalent increase in memory capacity, thus there will be a quadratic increase in the number of pixels. Therefore the CPU power will never catch up with the number of pixels. The hope is that there will one day be a resolution sufficiently large for the environment.

A good illustration of this phenomenon is in the Sony AIBO platform. The early model, the ERS-210A, contained a camera that was capable of a resolution of 176 x 144 (25344) pixels (3 bytes per pixel) with a frame rate of 25 fps. This meant that the vision system was required to handle $3 * 176 * 144 * 25 = 1900800$ bytes, or 1.81 Mb per second. The ERS210-A was equipped with a 600 MHz processor for this task. By contrast, the new model (the ERS-7) has a camera capable of 208 x 160 pixels (again, 3 bytes per pixel) with a frame rate of 30 fps. This requires the processor, which is 1 G Hz in this model, to handle 2.83 Mb per second. This is a 64% increase in data with only a 60% increase in processor speed.

If vision is to become a useful sensory input for mobile, autonomous robotics, beyond applications in simple environments, then algorithms must be developed that run with realistic efficiency — say linear order on the number of pixels that

are examined. Algorithms for most vision-processing tasks are currently non-linear. For example, the Hough transform is commonly used to detect straight lines and other shapes within images. The Hough transform is $O(n^2)$ on the number of pixels[2] to detect straight edges, and increases as the detection task becomes more complex. It is $O(n^3)$ to detect circles or rectilinear parabolas and $O(n^5)$ to detect an arbitrary conic-section at an arbitrary orientation. Another example is the common task of edge and corner detection. Edge detection algorithms are linear on the number of pixels but with an extremely high constant factor that depends on the size of the window used in the algorithm (usually around 9). Such a constant, when operating on all of the pixels in an image, makes an edge detection algorithm extremely slow. For example, to run an $O(n)$ algorithm with a constant of 9 on the ERS-7 means that the processor must handle 25.47 Mb per second as opposed to 2.83 Mb for an $O(n)$ algorithm with a constant of 1.

1.1.2 Vision for Dynamic and Unpredictable Conditions

The second problem area is that of robust adaption to varying conditions. Image analysis is a difficult and processor intensive task when the conditions are known *a priori* but the tasks are more difficult when faced with unknown conditions — especially unknown lighting conditions. Most of the image analysis literature overcomes this problem by calibration, which is a method poorly suited to mobile robotics. For that reason, one of the most discussed topics in the literature for robotic vision at the moment is that of robustness to varying lighting conditions.

Take, for example, the common image processing task of colour segmentation which is used as the basis of many object recognition systems. This task seeks to assign each pixel in the image to one of a discrete set of *a priori* colour classes — to label all the pixels in the image that look (for example) blue as belonging to the colour class BLUE. Colour classifiers are usually learned by supervised training from a set of sample images. This works well when the lighting conditions are relatively static. A problem arises when the robot must operate in a new lighting condition that is significantly different to the one in which the classifier was trained. The appearance of colour changes dramatically and in a way that has been shown to be very difficult to model, depending on the reflective properties

[2]Refer to the Glossary for a definition of Big-O notation on Page 181.

of the surface and the temperature, colour and intensity of the light that it is exposed to [83]. Thus, if a classifier is suited to one particular condition, and extra lights are turned on or off or a shadow passes by, then the classifier will cease to function well. If the lighting condition changes dramatically, then it will cease to function at all and consequently the object-recognition system will yield incorrect results.

Again, if vision is to become a useful sensory input for mobile robotics then techniques must be developed for use in real-world lighting environments. In fact both outdoor and indoor environments have dynamic lighting conditions in the real world. It is only dedicated venues that have controlled illumination. Even in simple applications we expect to be able to operate our robots at different venues. This is infeasible if the calibration process for each venue is a time consuming task. Many current calibration systems require time in the order of hours where a human operator supervises the learning process. This kind of calibration is impossible when the robot must cope with sudden and unexpected variations in lighting conditions while it is in operation.

1.1.3 Reliance on Domain-Specific Knowledge in Object Recognition

The third problem area is that there is a high reliance on domain-specific knowledge in most image-processing applications. Many software systems work well for their particular domain but adapt poorly when a new environment is encountered. This is undesirable for a mobile robot that must operate in a variety of conditions — some of which will surely be outside the parameters imagined by the original software designer.

In simple applications there will be a discrete and fixed set of objects in the environment of the robot. The task of the object recognition system is to locate and identify these objects within each incoming image. Feature extraction for this task is well studied and there are almost as many different techniques as there are objects. This is part of the problem. In a real-world environment, the robot must have some way of learning the properties of unexpected and previously un-encountered objects. This is not an easy task and there are many unsolved problems. For example, what internal representation will the robot store for encountered objects so that it will recognise them if it ever sees them again?

How will the robot deduce the properties of the object that it is observing? How will robots classify objects into semantically-related groups to determine their function?

In the field of autonomous robotics it is not adequate to rely on a vision routine that contains large amounts of domain knowledge about all the objects we expect our robot to encounter. Many currently existing applications utilise this domain-dependent style of object recognition, and therefore become unsuitable for use in a general purpose robot. Even if the task of autonomously learning object properties of new objects seems distant, steps toward this goal are very feasible. For example, object recognition systems can be developed using a generic description language that is capable of describing objects in the environment in such a way as to enable the vision system to identify them if it ever comes across them. Information about the utility and functionality of the object may also be written into the description. This task requires a descriptive language and not an imperative one such as most programming languages. The descriptions should be based on some form of logic language. Even though unsupervised learning of such descriptions remains a difficult task, at least robots may be easily configured for use in different environments while such problems are solved. Moreover, a descriptive language will be preferable for humans as they code and verify their programs, because they will be relieved of the orthogonal task of describing object properties in the code.

1.1.4 Development and Debugging Facilities

There are yet other reasons why the body of knowledge in computer vision is often not directly applicable to the task of autonomous robotics. Some of these reasons are practical rather than theoretical, because autonomous robots differ from computers in several important aspects.

Firstly, development for most mobile robots is a very difficult task. Compiled code is often not native to the development machine, that is, code is often cross-compiled[3] on an external computer such as a PC and uploaded to the robot. This means that debugging facilities that are normally available to developers such as Just In Time (JIT) debugging[4], code stepping and variable inspection, may not be available for code developed for the robot.

[3]Refer to the Glossary on cross-compiled code on Page 182.
[4]Refer to the Glossary on JIT debugging on Page 183.

In fact, mobile autonomous robots remain generally one of the hardest systems to develop for. They operate in dynamic and unknown environments, which minimises the utility of domain specific knowledge and makes the task of predicting what sensory input will be encountered very difficult. They are non-accessible when they are operating, and this, combined with the non-deterministic nature of the environment, makes understanding the runtime execution path of the code extremely difficult. Often they must operate in a team situation, adding to the complexity of a real-time environment with the difficulties of multi-agent systems. Sometimes this is even an adversarial environment where the robot must compete or fight against other agents while cooperating with its own team. For these reasons there has been an enormous investment in software simulators.

Another problem that is detrimental to the development of vision systems is that there are very few robots equipped with a facility such as a computer monitor that is suitable for viewing the incoming frames from the camera. It is extremely difficult to develop an algorithm to process a stream of images if it is difficult to visualise the entire stream. Even when robots are equipped with a network device as a means to send images to a PC, the limitations of bandwidth make streaming every image impossible. Interactive text based I/O to and from the robot is desperately inadequate for this kind of problem. Even wireless links, which are the usual communication channel of choice between a robot and PC, currently possess inadequate bandwidth for visual data.

Finally, given that real-time processing is so important for this task, code profilers and code analysis tools become all the more critical. These kinds of facilities are commonplace for general-purpose development environments but are usually lacking in the environment that exists for a custom-built robot. This makes the task of developing useful debugging tools essential. Again, these tools should not need to be built on a per-application basis (as is often the case). A little forethought and planning will produce a vision development system that is useful across a wide variety of domains and applications.

1.2 Vision Pipelines for Robotic Vision

A vision pipeline is a sequence of techniques and algorithms applied in a pipeline architecture [115], where each step in the pipeline manipulates or analyses image data. There are certainly many varied methodologies for image-processing solu-

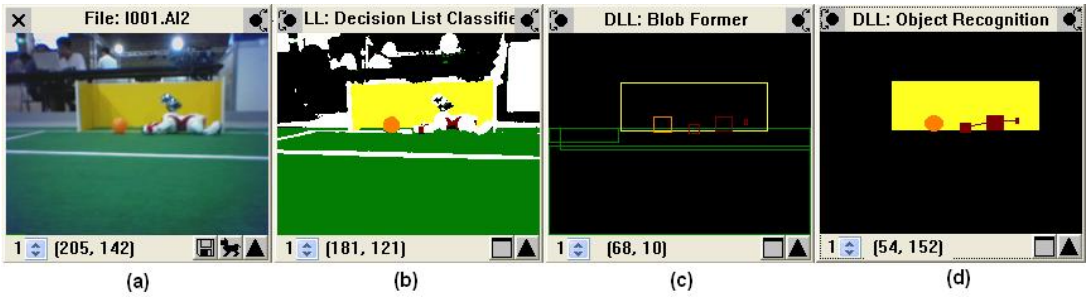


Figure 1.1: The Bruce *et al.* image processing pipeline first segments the image (a) by colour (b), then forms connected regions (blobs) (c) and uses this information to do object recognition (d).

tions and it is also true that robotic vision systems reflect this variety depending on the specific application context of the system. However, we have noticed across a variety of problem domains within mobile autonomous robotics, that a similar pipeline is emerging as a standard. This pipeline was first described in 2000 by Bruce, Balch and Veloso [16] in the context of the RoboCup competition (see Section 2.4) and is shown in Figure 1.1.

The first stage in the pipeline is image segmentation (b). In this stage each pixel in the image is labelled as one of a set of colour classes. Pixels that look, for example, blue are labelled as belonging to the class BLUE. After the image is segmented, it is passed to a blob-former (c). Blobs are groupings of connected pixels that all belong to the same colour class. Each blob can then be analysed to determine its properties or relation to other blobs. This is the object-recognition stage (d).

The Bruce *et al.* pipeline is certainly not the only pipeline used by mobile, autonomous robots. Indeed, even within the RoboCup competition we have seen many interesting and significant variations in recent years, such as the one in the German team code [107]. However, the Bruce *et al.* pipeline does possess certain advantages that make it an extremely popular choice, especially for teams that do not emphasise vision research. Firstly, it is very efficient — it requires only one pass over the raw data and one pass over colour segmented data in order to complete the object recognition task. It is relatively easy to implement and there are third-party libraries available that implement some of its functionality[5]. Another advantage is that it is also relatively easy to calibrate

[5]See CMVision - http://www.cs.cmu.edu/ jbruce/cmvision/.

and to use in various different conditions[6]. So significant are these advantages that even some competitive teams do not deviate far from the pipeline, despite its age [25, 26].

Because of these advantages we have seen this pipeline emerge not only as a standard in RoboCup [96, 130, 25, 26], but also in other robotics applications [54, 72]. It is generally useful when the objects that we expect our robot to encounter are distinguishable by colour and when we know the lighting conditions in which our robot operates. We believe that this accounts for its widespread use. Operation in dynamic lighting conditions or with objects that contain more than one simple colour are regarded as difficult problems. So, as we have seen in RoboCup, researchers have tended to simplify the environment in order to maintain progress in other areas of robotics that are dependent on a working vision system. It has been expected that after initial progress has been made, the research will address the more difficult and general problem of complex objects in dynamic and unpredictable lighting conditions. We believe that the Bruce *et al.* pipeline has therefore found popularity in many different, simplified application domains.

However, for this reason we also believe it is becoming increasingly less useful. A large proportion of the machine vision literature of recent times, both in RoboCup and elsewhere [51, 117, 72][7], has attempted to address the problem of dynamic illumination conditions — a situation in which the Bruce *et al.* pipeline will not work. Indeed, we present a viable solution in this thesis, provided that the conditions are not *too* widely variable (see Chapter 8). One thing is clear: the old way of doing machine vision for mobile autonomous robotics will not work in more realistic environments. Something new is required.

Of course, the Bruce *et al.* pipeline is not a general solution to machine vision for mobile, autonomous robotics. In some applications the robot could never use colour as a means of object recognition. For example, robots that must perform SLAM[8] cannot use colour to map their environment unless the environment is very artificial. We make two observations about these types of robots. Firstly, they are not often bound by a real-time restriction. It is perfectly acceptable for a SLAM robot to stay in one place long enough to run whatever image

[6]Note: it is not useful in *dynamic* conditions, but it is easy to use in many different conditions if calibration is permitted prior to use.

[7]See Chapter 8 for a more detailed discussion on this literature.

[8]Refer to the Glossary on SLAM on Page 183.

analysis algorithms one desires because the environment will not change around it. Secondly, vision is not usually employed as the primary sensor on these kinds of robots. For example, SLAM robots usually employ laser range finders or the like.

In the context of real-time autonomous robotic vision, colour is one of the only object features that is sufficiently easy to distinguish in order to make the image processing fast enough to keep up with the frame rate of the camera. A pipeline similar to Bruce *et al.* can even be used in military hardware in order for missiles to detect targets via the infra-red spectrum [113]. In general, where colour segmentation and blob forming has not been useful for the task of object recognition, either different sensing hardware has been employed or the real-time response requirement has been relaxed.

1.3 Aims and Contributions

The contribution of this thesis is to advance the state of the art in vision systems for mobile, autonomous robotics. We have argued that there are four distinct, but overlapping problem areas that must be addressed in order to do so. It is extremely difficult to advance in the field without contribution in all four areas. For example, there is no way that any high-level processing algorithm will run efficiently enough to be useful if there are no suitable low-level algorithms on which to build. Yet there is no point in improving the low-level algorithms if the high-level algorithms still will not be useful in the dynamic visual conditions present in robotic domains. Furthermore, if development of these systems remains a difficult (sometimes impossible), time-consuming and arduous task, who will implement the solutions?

We therefore divide this thesis into four sections. Each section addresses the state of the art and our research and achievements in one of the problem areas identified in Section 1.1. We have done this for organisational purposes, however, we continue to recognise the inter-relationship that exists among all problem areas.

1. It is necessary to reduce the computational complexity of many common low-level image-processing algorithms. As mentioned previously, most algorithms are far too expensive to use in a real-time environment. Although the number of image processing algorithms is enormous, the number of

available low-level techniques is actually fairly small. Thus most high-level algorithms are built on a small set of low-level techniques. We therefore argue that improvements in the low-level algorithms will yield a great improvement in the runtime of many different image processing algorithms and therefore some algorithms that have previously been too expensive to use may now become feasible. Our contributions in this area are detailed in Part I of the thesis.

One of the most studied techniques in computer vision is that of image segmentation and there are several very efficient and fast techniques for segmentation in the literature. Nevertheless, we have made several important contributions to this algorithm, both in classification speed and calibration efficiency. The memory required to store our calibrations is also an order of magnitude smaller than that of comparable systems, making it extremely useful in systems where memory capacity is limited. We discuss our technique in Chapter 4.

Edge detection has seen only limited use in machine vision literature. The reason for this is that it is generally regarded as an extremely expensive process. Although it runs in linear time on the number of pixels in the image, each pixel must be examined more than once. This is undesirable given the large number of pixels in most images. Another important reason why edge detection is sometimes disregarded in machine vision pipelines is because it does not cope well with blur in the image. In the context of machine vision the camera is mounted on a non-stationary platform (the robot) and therefore blur will probably be encountered. We have performed several optimisations on standard edge detection techniques in order to overcome these two problems. Not only is our edge detection fast, but we also have the ability to compensate for blur without a complex and expensive pre-processing phase. Our improvements are detailed in Chapter 5.

Another commonly used and studied low-level algorithm is that of straight-edge detection. Possibly the best known algorithm in this category is the Hough transform which runs in quadratic time on the number of pixels in the image. Its memory requirements are also $O(n^2)$ on the number of pixels. There are some slight improvements on this algorithm in the literature

as well as some faster heuristics. One such faster algorithm is commonly found in the computational-geometry literature: the Douglass-Perker algorithm [140]. This algorithm runs in $O(n \log n)$ complexity on the number of pixels in the boundary of the region to be vectorised. We have improved this algorithm to improve the complexity of the overall vectorisation to $O(n)$ on the number of pixels. Experimental evidence suggests that this is a substantial improvement in the context of image processing because n is generally large. We detail our method in Chapter 6.

2. The task of adaptation to dynamic conditions remains a challenging problem for mobile autonomous robotics. As we have argued, most robotic vision systems are built on the basis of colour segmentation which works well only in known lighting for which the robot has been calibrated. Our contributions in this area are detailed in Part II of the thesis.

   We present a technique that significantly overcomes many of the problems associated with vision in dynamic and variable lighting environments. We show experimentally that our technique allows for robust object recognition even when lighting, temperature and intensity vary dramatically, as well as in the presence of dynamic shadows in natural lighting conditions. We detail our method in Chapter 8.

   We also present a technique for posture and gesture recognition that overcomes some of the problems associated with unknown viewing distances and angles. Our technique is based on the property of symmetry so it is useful in a wide variety of situations. It runs in linear time based on the number of pixels in the boundary of the object so it is extremely fast. Furthermore it is useful in a wide variety of problem domains. We present our technique in Chapter 9.

3. There is a large reliance on domain-specific knowledge in image processing systems. For every object we wish our robot to identify, we must tell it what features to look for in each image. Our contributions in this area are detailed in Part III of the thesis.

   Before it will be possible for a robot to autonomously learn to recognise and respond appropriately to previously un-encountered objects, it will first be necessary to develop some generic internal representation of objects.

We present a declarative XML-based language (XML Object Description: XOD) capable of robustly describing not only the properties of objects, but the techniques required to detect such an object within an image. In this way it is possible to serialise object definitions to files and therefore the object recognition system will be able to recognise any object for which there exists a valid file. Therefore no code will need to be changed should the object recognition system be required to operate the robot in a different context. We detail our language and an implementation for it in Chapter 11.

4. The final problem area is that of development and debugging facilities for machine vision on mobile autonomous robotics. We have argued that even with modern compilers and debugging utilities, development for mobile robots, especially in the area of vision, remains an extremely difficult task. Our contributions to this area are outlined in Part IV of the thesis.

   We present a tool to support the entire software development life-cycle for generic embedded vision systems. Our tool provides a flexible, modifiable, off-line immersive environment where the code that is running on the robot can be tested natively on the PC. It does this both through support for C/C++[9] add-in modules (DLLs[10]) and C/C++ scripted files. The code that runs on the robot may be directly tested, debugged and profiled on a PC. The results of running each segment of the code on each image can be presented visually to the developer in a pipeline architecture. Our tool supports direct streaming of images from a robot via any network connection as well as saving, loading and modifying image files and streams. The format of images is flexible so that our tool can be used for many different image formats from many different cameras. We present this work in Chapter 12.

[9] Refer to the Glossary on C/C++ on Page 181.
[10] Refer to the Glossary on DLLs on Page 182.

Chapter 2

Our Development Platform

Our primary development test platform is the Sony AIBO in the context of the RoboCup Four-Legged League. Where it is relevant to our work we have used other platforms and environments besides those described in this chapter, but we leave descriptions of those environments to the corresponding chapter. There are several models of the AIBO, of which we use two, the ERS-210/A and the ERS-7.

2.1 The ERS-210/A

The ERS-210/A is the earlier of the two models. The only difference between the 210 and the 210A is the processor speed. The ERS-210 has 20 degrees of freedom:

- Three per leg: rotators, abductors, and knees.
- Two in the tail: tilt and pan.
- Two in the head: pan and roll.
- One at the base of the neck: tilt.
- One in the mouth: open/shut.
- One per ear: up/down.

As well as these, there are several other actuators:

Figure 2.1: The Sony AIBO ERS-7.

- Seven lights in the display panel of the head.
- One light on the chest (on/off).
- One speaker in the mouth capable of playing 8-bit wave files.

The sensors on the model are:

- A colour camera (176 x 144 pixels, 3 bytes per pixel (YUV colour format), operating at 25 frames per second, in the nose.
- Push button (on/off) on the chest.
- One touch sensor on the pad of each foot.
- One touch sensor underneath the chin.
- One touch sensor on the head.
- One touch sensor on the back.
- Three internal gyro-meters (tilt, pitch and roll).
- One infra-red distance sensor in the nose.
- One microphone in each ear.
- One internal temperature sensor.
- All joints report their load and position.

The model comes equipped with 802.11B wireless Ethernet capabilities. Its processor is either a 384MHz (ERS-210) or 576MHz (ERS-210A) MIPS processor and it is equipped with 32Mb of RAM.

2.2 ERS-7

The ERS-7 has some minor variations from the ERS-210 design beyond the obvious external differences:

- There is no roll of the head. Instead there is an extra tilt at the top of the neck as well as the one that was already at the bottom.

- There is a much more sophisticated facial display containing 83 lights of variable colour.
- There are mode-lights behind the ears capable of three colours each.
- There are three touch sensors on the back, each with a corresponding light.
- There are three (instead of one) infra-red distance sensors: one long range and one short range in the chest, and one long range in the nose.
- The camera captures at a resolution of 208 x 160 pixels, 3 bytes per pixel (YUV colour format), 30 frames per second.

This model still comes equipped with 802.11B wireless Ethernet support. Its processor is a 64-bit RISC processor and it is equipped with 64Mb of RAM.

2.3 Development Environment

The operating system for the AIBOs is called Aperios[1] and is distributed free of charge by Sony along with a development platform which is called Open-R[1]. Code for the AIBO is developed in the C++ programming language on a PC[2] and cross-compiled for Aperios.

On boot, the AIBO reads the operating system as well as any programs and configuration files from a Memory Stick[2] that is inserted into its undercarriage. Memory sticks come in varying sizes (8, 16, 32, 64Mb) and so they currently impose only a theoretical restriction to the size of the program — not many executables reach 8Mb in size. Nevertheless, the lack of a hard-drive in the AIBO means that the memory stick size limitation can be exceeded if the AIBO is writing data (such as images or debugging information) to the stick at runtime.

Each program that runs on the AIBO is considered an object by the operating system. In traditional object-oriented terminology[3], an object has state and a set of messages that can be called to communicate with it and this is true of Open-R objects as well. The only difference is that an Open-R object represents an entire processing thread. Complete documentation on Open-R may be obtained

[1] "Aperios", "Open-R" and "Memory Stick" are trademarks of Sony Corporation.
[2] See the Glossary on PC's on Page 183.
[3] Refer to the Glossary on Object-Oriented Programming Paradigms on Page 183.

at Sony's website[4]. The code is event driven and messages are sent to objects whenever one of the following events occurs:

- A new image is obtained by the camera.
- New sensor information (for all sensors other than the camera) has been collected.
- The motors/actuators are ready to receive a new instruction.

If the code that was called on one event has not completely finished by the time the same event is triggered again, then the call will be ignored until the code is ready.

This interrupt-driven code model presents a problem to the vision system developer of the AIBO platform. The image data interrupt is triggered at 30 Hz and there is one new image per interrupt. The sensor information is triggered at 125 Hz and there are 4 frames of data for each sensor per interrupt. This means that even though there is sensor data for each of the 30 images per second, there is no way to precisely match an image with the sensor data that was recorded at the same time, though a rough correlation is possible. Experimental evidence suggests that the head may move up to 8^o even within the time it takes to process a single image.

2.4 RoboCup Four-legged League

While there are several applications for the AIBO, our main test-bed has been in the domain of the RoboCup Four-legged League. RoboCup is a collaborative effort by a large international group of universities and other research institutions to produce useful research in the field of robotics. There are several endeavours within RoboCup including competitive robotic soccer, robotic rescue in disaster areas and RoboCup Junior which encourages participation in technological fields from an early age. The Four-Legged league is one of the divisions of the soccer competition where the hardware is restricted to Sony AIBOs. Therefore the difference in quality of play between teams is completely determined by the quality of the software that is written for the task.

[4]http://openr.aibo.com/

Although the specific rules vary somewhat from year to year, in the 2005 competition each soccer team is composed of four AIBOs that play on an approximately 6m x 4m field. There is one player that is designated goal keeper who can go anywhere on the field. The other three players are free to go anywhere except in the defending goalie box. The robots on each team are allowed to communicate with each other via wireless Ethernet, but no external communication (either to a PC or a human controller) is allowed. The robots play autonomously. The winner is the side with the most goals after two 10 minute halves.

To assist in play the field is largely colour coded. Refer to Figure 2.2. Teams wear red and blue uniforms, the ball is orange and the field is green with white lines. One end is designated the blue end and has a blue goal and blue and pink field beacons. The other end is designated the yellow end and has a yellow goal and yellow and pink field beacons. The lighting, for the moment, is fixed at a uniform 1000 Lux at 3000^o Kelvin over the field. There has been a push in recent years toward using natural illumination conditions, but sufficiently good object recognition has eluded the league. This illustrates the difficulty of the problem.

There are several other important aspects of this league in relation to computer vision. Firstly, the camera is located in the head of the AIBO which may be pointing in a direction independent of the rest of the body. Although vague readings can be obtained from sensors and accelerometers, it is very difficult to accurately determine the view plane of the camera relative to the ground. This makes image processing a very difficult task. For example, we must usually locate the horizon within each image from features of the image itself if we require accuracy. It is possible to estimate the horizon from the known positions of the joints and accelerometer readings, but the results will be quite inaccurate if the AIBO is walking. Therefore image processing techniques must be used to refine it. The horizon is such an important feature to detect because, although colours and lights are controlled below the horizon, above the horizon there may be any combination of lighting effects and colour. Someone in the crowd wearing a yellow T-shirt can easily be mistaken for the yellow goal. The capacity of the head of the AIBO to pan and tilt through different axes makes it possible (and common) that the "top" of the image will not be the pixels with the lowest y-value in the image. This precludes any standard top-to-bottom processing technique unless the horizon can be accurately determined.

Figure 2.2: The 2005 RoboCup Four-Legged League playing field, showing the typical "kick-off" configuration of the robots.

Another challenge that is unique to this league is that the camera has no capacity for omni-view, making the choice of where to orient the head a strategic decision. If the head is raised the ball may be missed but beacons are more likely to be seen. In a game of soccer, is is better to know where you are or where the ball is? Even if you know where the ball is, you still need to know where to kick it.

The robots of this league are also legged which means that as they move around, kick and walk they change shape. This presents two problems to the vision system. Firstly, as the robot moves around, the camera moves with it, making the image shake in unpredictable ways (as opposed to robots that are on wheels where the camera is relatively stable). The other problem is that there is no easy way to identify other agents with any degree of accuracy. The colour of the uniforms makes them relatively easy to detect within an image, but the many possible morphologies of an opponent means that details such as orientation, posture and even proximity are difficult to determine. The league is yet to develop a usable recognition system for opponent AIBOs.

More information about the Four-Legged League can be found at the league web site: http://www.tzi.de/4legged/bin/view/Website/WebHome, or at the official RoboCup web site: http://www.robocup.org.

2.5 Rationale

We have selected the Sony AIBO and, in particular, the RoboCup Four-Legged League as our primary development and testing application for several reasons. Firstly, the AIBO is a fairly typical example of an autonomous, mobile robot. While it is certainly true that there are many different varieties of robots — perhaps as many as there are applications — there are some features that are common.

Mobile autonomous robots are expensive items. Therefore, to minimise the cost where appropriate, they are usually fitted with the minimal hardware to perform the task required. This means that the processor on a robot is likely to be several generations behind the current hardware technology in computing. This is certainly true of the AIBO which has neither a fast (by modern standards) processor, nor a high quality camera.

Secondly, development for autonomous robots fits into the category of em-

bedded systems. Robots usually do not come with all the facilities of a PC — a screen, keyboard and other HCI[5] devices. Code must be cross-compiled and uploaded to the robot. The AIBO is certainly typical in this regard. The only native debugging facility available is a text stream over wireless Ethernet. This makes run-time debugging of vision processing code extremely difficult, though not more difficult than in many other robotic applications.

Although general purpose robots come in all shapes and sizes, there is a push in certain areas of the industry to make them appear more like a human. This means that as robotics progresses we will increasingly see legged robots with independent control of the direction of vision. The legged nature of the AIBO adds many problems to vision processing because it is difficult to determine precisely the view-plane of the camera. We expect this problem to be increasingly common as legged robots replace wheeled ones. The independent control of the head also adds strategic difficulty to the vision system which must now decide where to point the camera and what to look at.

Therefore the AIBO is fairly typical not only of the current state of robotics, but of the future direction as well. The AIBO is also less expensive than many custom-built robots so this adds to the attractiveness of the platform for many general-purpose robotics researchers.

If RoboCup is to achieve its stated goal of developing a robotic team of humanoid soccer players that can beat the current world champion team by the year 2050, then vision will certainly play a critical role. The current standard of vision in the league and in the RoboCup community is fairly elementary. Objects are identified primarily by colour and systems are used that are in no way tolerant to changing lighting conditions. Nevertheless, RoboCup is state of the art in its general purpose vision. There have been vision solutions developed for specific tasks (such as driving) that work well but do not attempt a general purpose solution. If RoboCup achieves its goal then a robot will be able to play soccer under human conditions — that is with natural lighting, in variable weather, in a field surrounded by advertising and a crowd. If a vision system is developed that will work under these conditions then it will work in any conditions for any task.

RoboCup also provides a valuable context in which to compare techniques against other research. There is a great deal of valuable theoretical work done

[5]Refer to the Glossary on Human-Computer Interaction on Page 183.

in machine vision that is nevertheless very difficult to use in robotic applications. The reason for this is that the work is published in academic papers. If I desire to test the work of another researcher against mine then I must first redevelop their code for my context: it has to run on my robots and in my testing environment. Researchers do not have time to do this for a large number of alternative algorithms. RoboCup removes some of these limitations by providing a common robot and a common testing context. It is relatively easy to see which vision system (or which component thereof) performs better simply by running the two pieces of code side-by-side. This feature of RoboCup has accounted for much of its success in general as it is true not only for vision but other areas of robotics as well. Many researchers find the direct comparison of algorithmic quality valuable. The competition merely provides added incentive.

Part I

Computational Complexity of Image-Analysis Algorithms

Chapter 3

Basic Concepts and Related Work

We divide the image analysis literature into two overlapping areas. Firstly we classify some algorithms and techniques as *low-level*. In this category we place algorithms such as noise elimination and filtering, edge and corner detection, image segmentation, texture analysis and shape recognition. These techniques are fundamental to *high-level* analysis routines that attempt to find features for a specific analysis problem (for example, finding the eyes in a face recognition problem). The low-level analysis routines are fundamental in that they are used across a wide variety of different problems, and are combined in different ways by the high-level routines. We therefore expect that the greatest impact on runtime performance, in a general way across the image-processing literature, will occur when we examine these low-level routines for the purpose of improving the computational complexity.

The set of low-level image processing tasks is actually fairly small. It is true that there is a vast number of algorithms in the literature for each separate task — but the tasks themselves seem to be fairly common. This further justifies our hypothesis that an improvement in the computational cost of the low-level tasks will reap a large reward across a wide variety of high-level image-processing applications.

In Table 3.1 we list each of the low-level tasks along with a brief description and the computational complexity that is generally associated with the task. The exact computational complexity, of course, will vary with different algorithms and we will examine the relevant variations in detail.

Low-Level Task	Example	Runtime Cost
Image filtering	Noise filtering	$O(n)$
	White balancing	$O(n)$
	Histogram equalisation	$O(n)$
Image segmentation	Colour classification	$O(n)$
	Connected region	$O(n \log n)$
Edge detection		$O(n)$
Vectorisation	Straight line detection	$O(n \log n)$
	Corner detection	$O(n \log n)$
	Shape recognition	Depends on shape — $O(n^5)$ for general conic section
Texture analysis	Connected region	Depends on number of known textures
	Feature analysis	

Table 3.1: The most common low-level image processing algorithms and their associated runtime costs.

In this thesis we focus our attention on three of these low-level tasks: image segmentation, edge detection and vectorisation. We propose that low-level algorithms should be robust enough that no pre-processing (image filtering) step is required before the algorithm is executed. Each pre-processing filter effectively doubles the amount of data that the processor must cope with on each incoming image. Due to the limited nature of available processing power in mobile robotics it would seem wiser to develop algorithms that do not require a pre-processing step. This has been part of our design philosophy.

Furthermore we note that the analysis of textures is not a common problem for robotic vision tasks. While we recognise there could be some valuable applications for texture analysis in robotic vision (for example, visual inspection of uneven, realistic terrain) we leave this task to future work. At the present time there does not seem to be a great need for these algorithms in general purpose robotics.

3.1 Image Segmentation

The task of image segmentation has formed the backbone of many robotic vision algorithms. This has certainly been the case in RoboCup where teams have followed closely the same paradigm since 2000 when Bruce, Balch and

Veloso defined it [16]. Image segmentation has been widely studied in the machine vision community in general. Many novel methods have been applied there [127, 125, 84, 129, 86] and it is easy to understand why. If the colours of known or interesting objects in the environment are distinct, then the colour can be easily used for the task of object-recognition. This is a very inexpensive object recognition process that does not require a high resolution camera or a fast processor. For this reason robotic developers often stipulate colours even when this imposes an artificial limitation on the user or the robots environment. For example, the Workpartner [54], developed by the Helsinki University of Technology is a gardening assistant robot that will follow its owner around the common household garden. It is calibrated to follow a red item — the designers obviously felt that red would be an uncommon colour in a garden environment — so the owner of the robot must wear a red shirt or jumper every time they wish to use it.

The shortcomings of an image segmentation approach to computer vision are substantial. Image segmentation performs poorly in conditions where the lighting condition is unknown or dynamic, but we will leave this discussion to the appropriate part of the thesis.

An image segmentation algorithm labels each pixel within an image as one of a predefined set of colours. For example, in RoboCup, the important colours are orange, green, white, yellow, pink, blue and red. In this context an image segmentation algorithm should assign one of these colour labels to each of the pixels in the image. Refer to Figure 3.1 for an example.

This task is not as simple as it first appears. Most colour spaces are large[1]. For example, in the YUV colour space[2] each pixel is made of three components: Y (intensity), U (chromatism 1: red-green) and V (chromatism 2: blue-yellow). Each of these components is represented by 1 byte (a value between 0 and 255) so the total number of colours that can be represented is $256^3 = 16,777,216$. The simplest possible solution to this classification problem is a lookup table that maps each of these possible values to one of the classified colours. Such a table would be 16 Mb in size assuming that there are no more than 255 colour classes. A table of this size is unrealistic not only to store but also to build when

[1]Refer to the Glossary for a fuller discussion on colour spaces on Page 181.

[2]We will use the YUV colour space throughout this thesis as it is the one used by the camera on the AIBO.

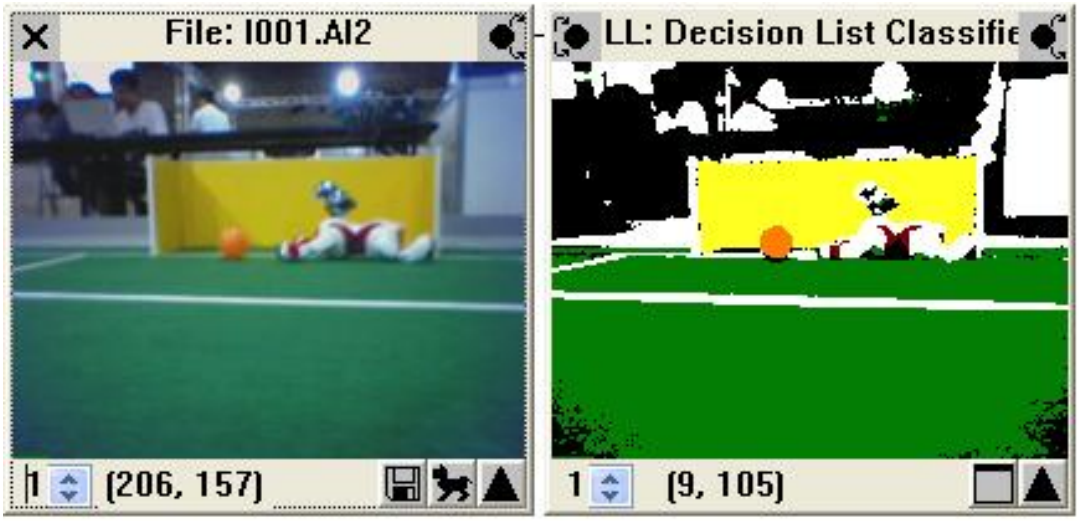


Figure 3.1: Image segmentation assigns a colour class to each of the possible colours in the original image. Objects in the environment that appear yellow, for example, are labelled as belonging to the class YELLOW.

you consider the arduous calibration task of assigning a colour class to each of the 16 million colour values.

Consequently, there is a large body of research on the best way to train and store a classifier that can do this task well. Research has explored machine learning and statistical discrimination techniques like linear discriminates [19], decision trees [27], artificial neural networks [132] and instance-based classifiers (k-nearest neighbours and other non-parametric statistics). At the time of writing, support vector machines are extensively used by the community [104, 105].

There is definitely a trade-off in building a good classifier. If you are willing to store large amounts of data and take a great deal of time to train the classifier, then a fast and accurate runtime solution can be found. In this respect a lookup-table of characteristic vectors [16] remains the best solution. However, the job of training such a classifier is a major undertaking requiring many hundreds of images and many hours of laborious human-supervised work. This is obviously undesirable. The trade-off then is to sacrifice runtime speed and accuracy for a smaller representation that is easier to train.

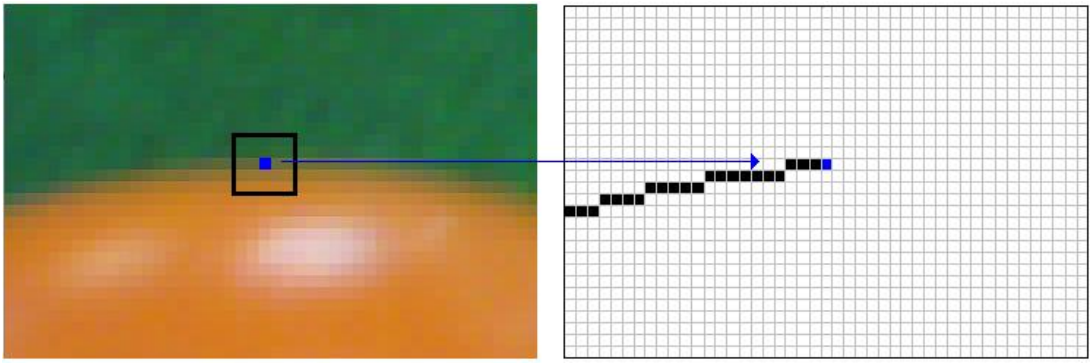


Figure 3.2: The process of detecting edges within an image involves sliding a window over every pixel. The pixels within each window are examined to determine if the centre pixel is an edge. Different edge detection algorithms vary on this examination step.

3.2 Edge Detection

An edge detection algorithm may be expressed as a function f with input image I and resultant image I':

$$f(I) = I'$$

The result I' will be a binary raster image where each pixel (x, y) is either "on" if (x, y) represents an edge in the source image or "off" if it does not. While the specific mechanics of different edge detection techniques vary, basically they all operate on a similar idea. A "window" (of width and height w) is slid over the pixels in I. Each window is examined to determine if the centre pixel is an edge. Different algorithms have different methods of examination. Refer to Figure 3.2.

Edge detection routines have a reputation for being very slow. It is easy to understand why when you consider that with a window size of w, and n pixels in the image, the entire runtime of the algorithm is $O(wn)$. Obviously w is constant, but in the context of image processing this constant is significant. Typically w will be small — say three or five — but this does not alleviate the problem. An edge detection algorithm with a window of size three will need to examine 2.3 Mb of data if the image is 1024x768 pixels.

Even though edge detection is considered a fundamental step by the image

processing community [50], it has been largely ignored by the robotic vision community due to these performance restrictions. There are some applications which require edge detection (such as simultaneous localisation and mapping in unknown environments [35, 93]). However where possible the technique is avoided [52]. Even when it seems unavoidable in the desired application people have used alternative (non-vision based) sensors such as laser range finders [53] or ladar[3] [42].

In the image analysis community, which cares less about runtime performance, the edge detection problem is well studied [57]. The most common edge detection routines are well known. The Canny edge detection algorithm [22] is considered a standard method and is often used in benchmarks of other algorithms. The Canny method applies a Gaussian blur to a grey-scale image[4] and then computes the directional derivatives for each row and column in the window. The derivatives are compared to a user-supplied threshold to determine edge pixels. This is quite an expensive computational process. We describe this edge detector more fully in Chapter 5.

There are other well known methods. Sobel [119] and the Roberts Cross [50] are also quite common methods that are less computationally expensive than Canny but more susceptible to noise. There are also statistical methods [62], genetic algorithms [15] and neural network approaches [121]. A large body of literature also exists on the best way to compare the effectiveness of different edge detection algorithms [57, 1]. In all of this literature, however, we see concern for the algorithmic robustness of the various techniques, but very little for the runtime efficiency. This is simply because the literature has been developed primarily by the image analysis community which does not have the rigid runtime restriction imposed by autonomous robotics.

Another reason that edge detection is difficult to use in a robotic environment is that edge detectors generally do not cope well with blurry images. In a robot with a directional camera (especially a legged robot), the speed at which the camera is moving can cause a significant amount of blur which obscures the edge information. This can be seen in Figure 3.3 where the orange ball does not even look vaguely circular and there exists no clear edge information in the

[3] See the Glossary on Ladar on Page 183.

[4] The grey-scale image corresponds to the Y channel in YUV images, but other colour spaces such as RGB and HSI would have to be converted to grey-scale before edge detection could be applied.

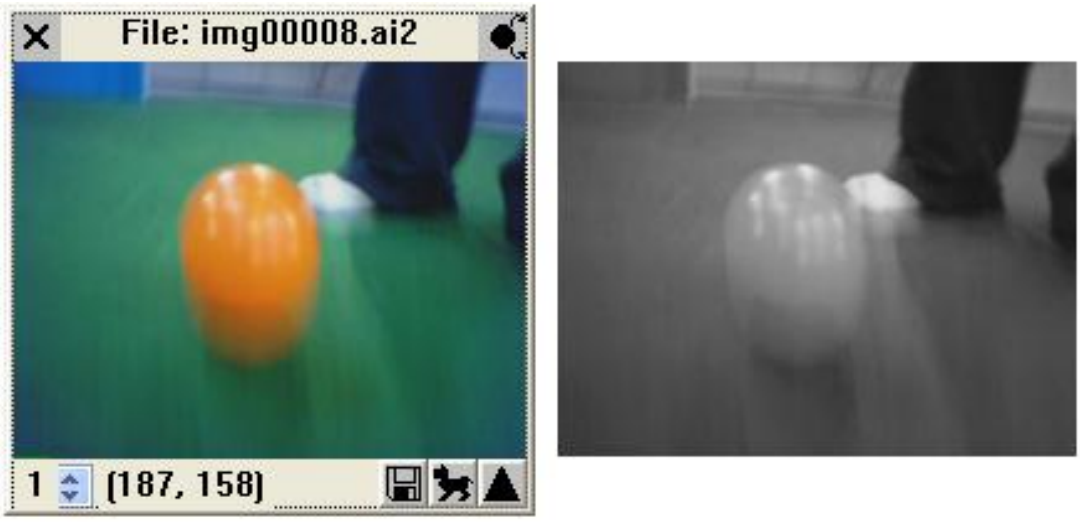


Figure 3.3: The effect of camera instability on image processing can be quite marked. In the above image the boundary between the ball and the field is almost indistinguishable in the Y-channel. This would make edge detection very difficult.

black/white (Y-channel) image between the ball and field. Some research has been done on restoring edge information in blurry images [102], but this kind of analysis is extremely computationally expensive. The edge detection methods we present in Chapter 5 are the first fast edge detection routines that cope well in the context of blurry images (under certain conditions) while maintaining a speed that is usable in the context of real-time image processing. We do this by including the colour information in our approach.

3.3 Vectorisation

The task of vectorisation is usually the next step after edge detection. The edge detection algorithm results in a binary image which, generally speaking, is not useful in itself for image processing tasks. It is much more useful to know that the image contains a line from point p_1 to p_2 than to have a list of pixels in raster form. It is therefore the job of vectorisation to convert the raster image (usually the binary edge image) into vectorised form. When we speak of vectors in this context we do not refer to the same concept that we find in linear algebra. A vector image is a list of parameterised shapes — each shape being stored in its

most succinct form. For example we may represent circles by their centre and radius, we may closely represent the generalised polynomial curve by a Bézier description [45] and we may represent a line segment by its start and end points. It is more work for a graphics engine to render such an image, but it is also much more useful for image analysis purposes.

Although many methods exist for the task of vectorisation of boundaries ([23, 47, 43, 67] for example), without a doubt the most commonly used technique is the Hough transform [65]. The transform will, in fact, locate any shape within an image, provided there is a suitable parameterisation for that particular shape. The parameterisation may be multi-dimensional but the important thing is that every possible occurrence of the shape in the image can be represented in a finite parameter space. For example, to search for an arbitrary straight line within an image we may represent the complete set of possible straight lines by two finite intervals on the parameters r and θ where θ is the angle of the line and r is the perpendicular distance of the line from the image origin. The parameter θ is bound between 0 and π, and r is bound by the size of the image. The edge points in the raster representation are processed and each possible r and θ to which the edge point could contribute is incremented in a two dimensional accumulator. The peaks in the accumulator space then represent the actual edges within the image. Refer to Figure 3.4.

The same is possible for any shape for which a finite parameter space can be found [8]. Of course, as the shape becomes more complex the parameter space will also expand. For example the Hough transform can be used to locate an arbitrary size circle in an image but must use a three dimensional parameter space — *radius* and *centre* (x and y). The general conic section can be found in a five dimensional parameter space — *eccentricity*, $focus$ (x and y) and *directrix* (as a line). As the parameter space increases, so does the computational complexity associated with the operation. The entire n-dimensions must be accumulated for each edge pixel within the image, and then after the image is processed the entire parameter space must be searched for local maxima.

There have been various attempts to optimise the Hough transform for various shapes using the particular properties of the shape in question, and these have met with some success [68, 118]. For example, it is possible to use the properties of the geometry of a circle to reduce the Hough transform for the general circle to $O(n^2)$ [69]. It is also possible to generate heuristics so that only

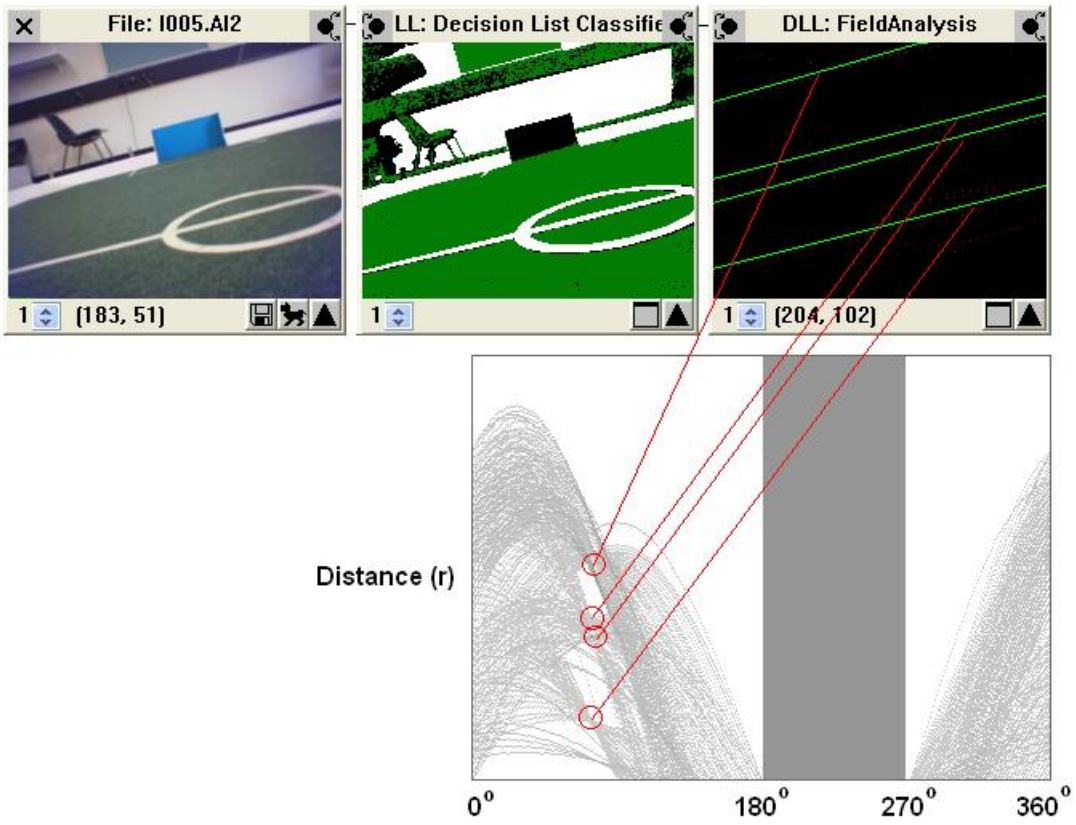


Figure 3.4: The Hough transform is one technique for straight line vectorisation. Each pixel (x, y) identified as an edge in the source image contributes to the accumulator for every possible straight line that can pass through (x, y). The accumulator space is then searched for local maxima.

a fraction of the pixels require examination [109, 85].

The Hough transform is popular because it works extremely well. It is very robust to noisy images and highly accurate. The problem, of course, for robotic vision applications is the runtime complexity. The computational complexity of the transform arises because it has a large multi-dimensional space to accumulate and then search. An approach that simply sub-samples pixels will not reduce the size of this space.

If we restrict ourselves to a straight-line vectorisation of the edges then there are other algorithms that are a great deal faster. The Douglas-Peucker algorithm [140] is commonly used in the geographical information systems literature for this task. It accepts as input an ordered list of pixels, E, between any two points: the origin pixel $o = (o_x, o_y)$ and the target pixel $t = (t_x, t_y)$. Usually this list of pixels will represent the boundary of the shape we wish to vectorise. The list is searched for the pixel p that is furthest from the line $\vec{ot}$. If this pixel is further than a threshold then the algorithm recursively searches $\vec{op}$ and $\vec{pt}$, otherwise the straight edge $\vec{ot}$ has been found. The poly-line formed by the recursion represents the vectorisation of E.

The Douglas-Peucker algorithm, in the form we have stated it above, is also $O(n^2)$ on the number of pixels in E. This has been improved to $O(n \log n)$ by using a binary search heuristic instead of iterating over the entire list E at every level of recursion [61, 60]. This represents a substantial improvement over the $O(n^2)$ complexity of the Hough transform. There are other algorithms that work along similar principles and also run in $O(n \log n)$ time [111]. We have managed to improve the Douglas-Peucker algorithm further to run in linear time and will present this work in Chapter 6.

3.4 Our Contribution

Most high-level image processing routines are built by using a combination of lower-level processing techniques. If we are to reduce the runtime cost of many high-level routines (to the point where it is feasible to use them in a mobile, autonomous robotics environment) then the best place to start is with the low-level routines. We believe that a small contribution in any one of these routines is very valuable as there is only a small subset of low-level image processing techniques from which most higher level techniques are assembled. Consequently we exam-

ine several low-level routines in the following chapters with the view of increasing performance and, where possible, reducing the computational complexity.

Of course, sometimes it will be necessary to trade accuracy for the performance we require. We note, however, that this is not as significant a trade as it first appears. The image analysis literature has primarily focused on accuracy over performance because this made sense in the context of the tasks for which it was being used. If you are analysing a CAD drawing, for example, then you would obviously be happy for the algorithm to take a little longer and do a good job. However, in the context of real-time vision, one small mistake in a particular frame will quickly be corrected by the next incoming image less than one second later. When viewed in this light the trade-off is reasonable.

Nevertheless we cannot afford to sacrifice too much accuracy. If vision is to be a useful sensory input then it needs to compete with much more specialised sensors such as laser range finders, and our image processing algorithms will still need to be accurate.

Chapter 4

Fast and Accurate Image Segmentation

We present in this chapter a colour classifier that is suitable for the task of image segmentation. Our classifier is equivalent in speed to a lookup table but is considerably more compact than any other classifier available. Also, in contrast to other classifiers, its stored representation is intuitive and easy to understand.

4.1 Our Colour Classifier

We may represent a colour classifier as a function $colour\_class : Y \times U \times V \rightarrow Colour$ that given a triplet (y, u, v) produces a colour class where $Colour$ is a member of a discreet set of colour classifications, COLOURS. This function may be easily represented as a single characteristic function for each member of COLOURS. For example, let $class\_orange : Y \times U \times V \rightarrow \{true, false\}$ return $true$ when the pixel (y, u, v) belongs in the class ORANGE.

We note that the set of all pixels in the colour space that we would like to classify ORANGE cannot be accurately described by any linear discriminator (refer to Figure 4.1). This is because the ORANGE area is not rectangular. This makes the individual characteristic functions difficult to define.

It is clear from this that the only way for a classifier to exactly represent each colour class is by a comprehensive lookup table. We certainly lose accuracy whenever we attempt to define the class ORANGE by any linear discriminator. We are, however, quite unconcerned with this loss of accuracy. The reason why such accuracy is unimportant is illustrated in Figure 4.2. The two highlighted

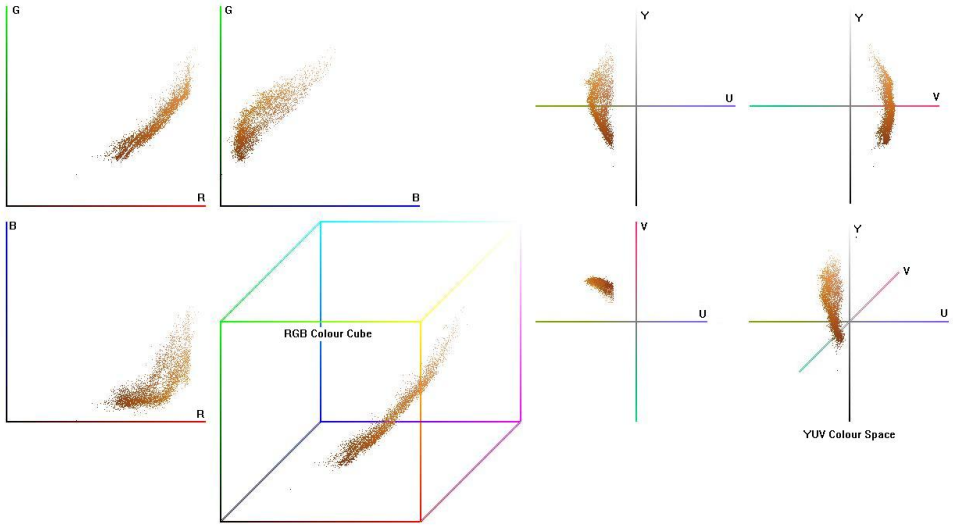


Figure 4.1: This Figure shows all of the (orange) ball pixels in Figure 4.2 mapped to the RGB colour space (left) and the YUV colour space (right). Notice that the area of space defined as ORANGE is not rectangular.

pixels in the image on the left have exactly the same (y, u, v) tuple even though our eye clearly perceives one as red and one as orange due to the context of the image. Should that pixel be classified as orange or red? Clearly the human eye, to some extent, defines colour by what it expects to see given the surrounding context of the image. An ideal classifier would therefore classify a pixel as class ORANGE only when the predominant colour of the surrounding shape is ORANGE. The problem, of course, is that image segmentation is usually the first step in object recognition. The image analysis algorithm does not yet know the context of the object in which the pixel lies. For this reason even a comprehensive lookup table will classify pixels incorrectly.

For this reason we are content with a classifier that recognises the core of each colour class[1]. Thus a linear description will be a suitable representation. The characteristic function of each colour class can therefore be represented by the projection functions on each of the dimensions Y, U and V. For example, the three projections Y_orange, U_orange and V_orange can be used to approximate $class\_orange$ in the following way:

[1]For reasons that we will examine in Chapter 8 we would ideally like to ignore pixels that could fall into more than one class depending on surrounding context.

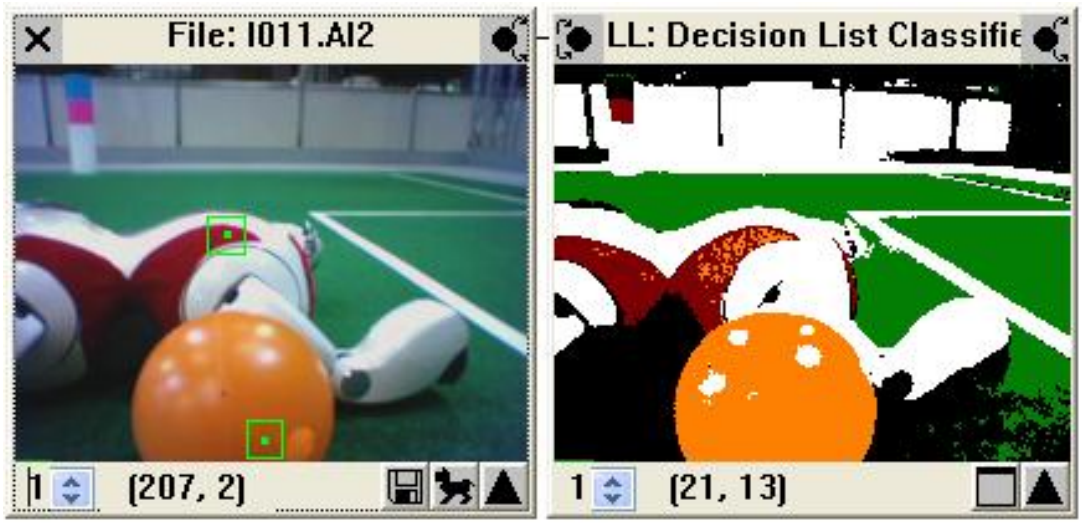


Figure 4.2: The highlighted pixels both classify to orange because they have the same component (YUV) values. Despite this, the human eye clearly percieves one as red and the other as orange because of the context of the image.

$$class\_orange(y,u,v) \approx Y\_orange(y) \wedge U\_orange(u) \wedge V\_orange(v). \quad (4.1)$$

Each of the characteristic projection functions has a domain of 256 values and thus can be feasibly stored in a lookup array that stores 1 if Y_orange is *true* and 0 otherwise. Thus there is a lookup table of 256 values for each of Y_orange, U_orange and V_orange which we store in 3 arrays. We can store the lookup for up to 32 characteristic functions in these arrays if they are of correct width[2]. We do this by putting the lookup for the first characteristic function in the first bit of the value, the second function lookup in the second bit, and so on. If more characteristic functions are required then we simply increase the size of the data type.

The final *colour_class* function is then essentially represented by a C++ bitwise AND-operation:

$$colour\_class(y,u,v) = Y[y]\&U[u]\&V[v]. \quad (4.2)$$

[2]32 bits is a convenient width because it represents an int data type on a modern (32 bit) processor.

The final step is to compute the $\log_2$ of the result to determine the ID of the colour class. This entire method of classification is shown in Algorithm 4.1, which is what the Machine Learning or Data Mining community would refer to as a Decision List.

Algorithm 4.1 Decision List classification.

Input: Arrays Y[255], U[255] and V[255] where the first bit in Y[n], U[n] and V[n] is the lookup of the characteristic function for the nth colour class. The colour tuple to classify is (y,u,v).

Output: The colour class of (y,u,v).

```
1: val = Y[y] & U[u] & V[v]
2: count = 0
3: while ¬(val & 0x01) ∧ (count < 32) do
4:     val = val >> 1
5:     count++
6: end while
7: return count
```

This implementation of our algorithm is similar to that presented by Bruce *et al.* [16]. The innovation of our approach is that each colour class may be represented in the array more than once because Algorithm 4.1 retrieves an index to the class, not the class itself. There are two advantages to this system over Bruce *et al.* The first is that our algorithm allows us to discriminate non-rectangular regions in the colour space using a decision list format. The second is that the representation of the calibration file is very easy to understand and edit (even by hand). Both of these advantages are discussed fully in the following section (Section 4.2, see especially Figure 4.4). The Bruce *et al.* algorithm does not permit more than one linear discriminant for each colour class, and therefore does not permit non-rectangular colour regions. This makes the Bruce *et al.* algorithm unsuitable for use in both of the calibration techniques (robust and sparse) that we present in Section 4.2.

Our entire classifier is at most 3060 bytes in memory and runs very quickly. Table 4.1 compares our method with some of the other classifiers that are being used in the RoboCup competition. It is easy to understand why our technique is so much faster than the others. If classification is treated as a spacial problem (k-Nearest Neighbours) then the classifier is required to compute Euclidean

Algorithm	Avg time on AIBO per frame (ms)	Amount of memory consumed (bytes)
Our method	1.71	3060
Complete lookup table	1.41	16777216 (16Mb)
k-Nearest neighbours	Depends on k but very slow (1ms / k)	Depends on k but fairly small (4 bytes / k)
Support vector machines	1.93	262144 (256Kb compressed)

Table 4.1: Comparison of our Decision List classifier with other methods that are available.

distances that involve a square root operation. If it is treated as a decision tree then it may process up to 20 levels of conditional statements before a decision is reached. Our method has a runtime cost only marginally larger than the fastest possible solution of the lookup table.

4.2 Classifier Calibration

In general, calibration for classification is a supervised learning task; given a set of sample pixels with known colour class, derive a classifier to assign a colour class to future pixel values. It is important to recognise that we may not encounter every possible (y, u, v) tuple in the training set, so the classifier must generalise. There are two types of calibrations possible for the task of classification: *robust* and *sparse*.

Let P be a training set of n images and $Orange_i$ be the set of pixels that we wish to classify as ORANGE (for example) within the i^{th} image. Then an ideal robust classification for the class ORANGE is:

$$class\_orange(y, u, v) = true \iff (y, u, v) \in \bigcup_{i=0}^{n} Orange_i. \tag{4.3}$$

That is, if a pixel with values (y, u, v) is recognised as ORANGE in *any* of the images in the training set, then the classifier should label this pixel as class ORANGE for any future images. Of course, an ideal robust classifier may not be possible. For example, as we have discussed, the same (y, u, v) tuple may be assigned to two different classes in the training set (even within the context of the same image). An ideal classifier may also suffer from over-fitting [138]. In practice, we weaken the condition by asserting that the classifier should label a

pixel as class ORANGE if it is recognised as ORANGE *more often* than any other colour class.

By contrast, an ideal sparse classification for the class ORANGE is:

$$class\_orange(y,u,v) = true \iff (y,u,v) \in \bigcap_{i=0}^{n} Orange_i. \tag{4.4}$$

That is, we wish to label a pixel (y,u,v) as ORANGE only if it is recognised as orange in *all* of the images in the training set. Again, sometimes it is necessary to weaken this condition. In practice we label a pixel as ORANGE if it is recognised as orange in *most* of the images in the training set.

There are both benefits and drawbacks to each of these two calibration methods. We argue for a sparse classification in Chapter 8, while the majority of existing systems use a robust classifier. For the purposes of this chapter it does not matter which type of calibration we wish to use — the calibration method will have only a very minor alteration from one to the other.

Each of our characteristic projection functions, as described above, is capable of storing any pattern of 255 bits. However, for the task of calibration we restrict this to a continuous block of 1's anywhere within the domain of the function. We label the lowest positive bit for a particular projection (y, u or v) and class (COLOUR) as $Min_{proj,COLOUR}$. Similarly we label the highest positive bit $Max_{proj,COLOUR}$. This means that each characteristic projection function is essentially testing the clause

$$(Min_{projection,COLOUR} \leq x) \wedge (x \leq Max_{proj,COLOUR}). \tag{4.5}$$

Therefore each characteristic function may be written

$$\begin{aligned} class\_{}_{COLOUR}(y,u,v) = {} & (Min_{y,COLOUR} \leq y) \wedge (y \leq Max_{y,COLOUR}) \\ & \wedge (Min_{u,COLOUR} \leq u) \wedge (u \leq Max_{u,COLOUR}) \\ & \wedge (Min_{v,COLOUR} \leq v) \wedge (v \leq Max_{v,COLOUR}). \end{aligned} \tag{4.6}$$

It is therefore this representation, in the form of a decision list [139], that we expose to the user. A typical calibration file has the following format:

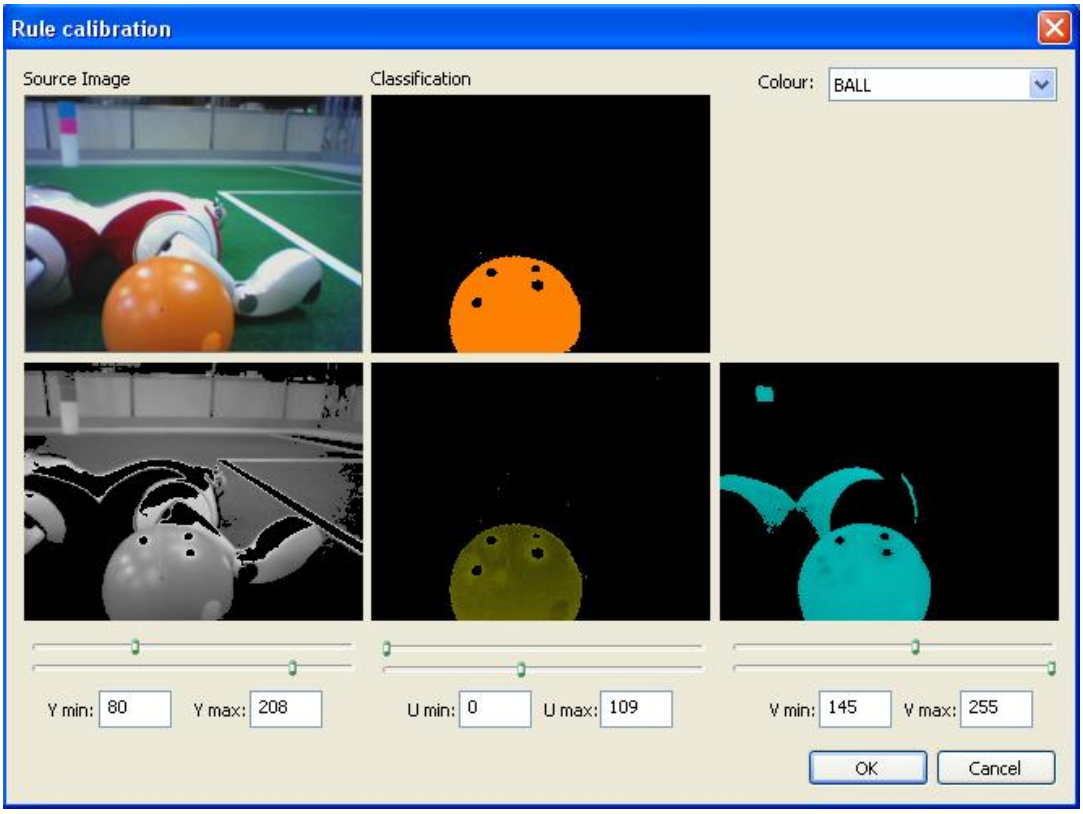


Figure 4.3: We may calibrate for each colour by separately examining the Y, U and V components and finding the projections for each characteristic function. This image shows our manual calibration tool allowing the user to select the Y (greyscale in the image), U (yellowscale in the image) and V (bluescale in the image) projections for the class ORANGE separately.

```
Colour_ID_1  Min_Y Max_Y Min_U Max_U Min_V Max_V
Colour_ID_2  Min_Y Max_Y Min_U Max_U Min_V Max_V
...
```

Of course, we are limited in the number of characteristic functions we can apply by the size of the selected data type, as explained above. In our case we are limited to 32 characteristic functions which, of course, may be increased if a larger data type is used. The results of applying one such characteristic function can be seen in the screenshot in Figure 4.3. Here the Y, U and V channels are calibrated separately to produce the overall characteristic function.

Although the rule format restricts us to linear discrimination within the colour space, significant flexibility is achieved by the decision list format as

Figure 4.4 illustrates. Image (a) represents the class ORANGE that we want our classifier to learn. In Image (b) the bounding rectilinear area minimally surrounds the class ORANGE. This is the optimal linear discriminator that completely contains the class, but it does not give a very good solution. There is a large area that our classifier will label ORANGE that is not orange. By using a decision list format we can do much better. We first find a characteristic function for the two shaded areas in Image (c), and label these pixels NOT_ORANGE or UNKNOWN. The while loop in Algorithm 4.1 will only continue evaluation until a characteristic function returns *true*. Therefore if the shaded areas in (c) are tested before the rule in (b), then pixels within them will not be classified as ORANGE even though the linear discriminator in (b) would have classified them so. In this way it is possible to obtain relatively good classifications of colours.

Of course, if we are training a sparse classifier instead of a robust one, we will not be interested in a box that completely surrounds the colour class. Instead we will want a discriminator that contains the core of each class. In this case linear discriminators are more than adequate. Of course we may require more than one characteristic function to adequately describe the core of each colour class (Image (d) in Figure 4.4).

4.2.1 Supervised Learning of a Calibration

It is very difficult to calibrate a robust classifier by hand although a sparse classifier is certainly possible if the right tools are used. Nevertheless the preferred method is to use a supervised learning algorithm. Even this can be quite time consuming and labour intensive. Our general approach is not significantly different to others although it has obviously been adapted to our particular classifier and calibration. Nevertheless we describe it here for completeness.

The basic approach is to build a sufficiently large training set and use covering algorithms to learn the decision list. Generally, given the training set T of n labelled instances (y, u, v,COLOUR), the supervised learning technique attempts to find the best characteristic function for some colour class and adds it to the decision list. All instances in T covered by the recently produced rule are removed from T and the algorithm repeats. Of course, there can be significant variation depending on what method is used to assess the value of a characteristic function.

We experimented with three algorithms of this class. The first is PRISM [24].

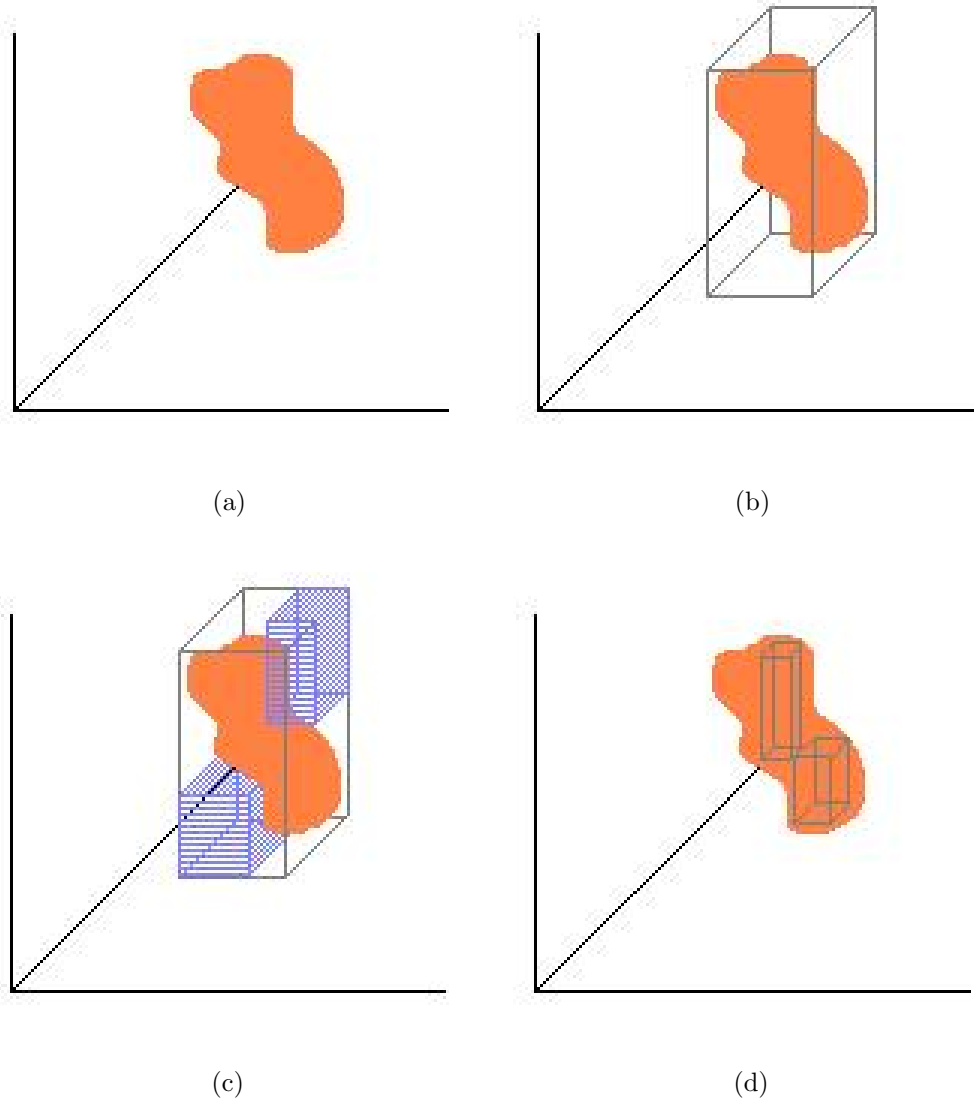


Figure 4.4: Colour classes cannot be described by orthonormal rectangular regions in the colour space (a). Therefore classification by linear discrimination is typically bad because it labels many pixels that are not in the class (b). We may use linear discrimination in combination with decision lists to do a better job (c). If the shaded regions are labelled *not* in the class, and are higher in the list, pixels within them will not be labelled as belonging to the class. More than one characteristic function can be used for each colour class (d). This is particularly useful for sparse classification where only the core of each colour is required.

This algorithm evaluates rules by their accuracy: that is, by the simple ratio p/t where p is the number of instances correctly classified by the rule and t is the total number of instances covered by the rule. A variant on this is to evaluate rules by their information gain, namely, $p[\log_2(p/t) - log_2(P/n)]$ where P is the total number of examples of the class in T. Our experiments showed that classifiers trained with these algorithms are inaccurate for this application.

At this point we make two observations. Firstly, we are not restricted to one characteristic function per colour class and, secondly, the rules can be ordered any way that is convenient for us. Clearly our classifier will execute more quickly if the correct characteristic function for a particular pixel is high in the decision list because of the while loop in Algorithm 4.1. We notice then that it makes sense to select the more frequent characteristic functions and place them at the beginning of the list. This required a statistical approach for which we used PART as implemented by Weka [139].

Chapter 5

Inexpensive Edge Detection

Edge detection is a critical aspect of many image processing applications. Nevertheless, as we have argued, it is often ignored in computer vision applications due to the high runtime cost associated with sliding a processing window over an entire image. In this chapter we present two of our edge detection algorithms. The first of these algorithms is an optimisation on gradient edge detection methods such as Canny and Sobel.

While our optimisation considerably improves the performance of the Sobel algorithm, and is an order of magnitude faster than Canny, it is still computationally expensive. With this in mind we present a second alternative that enables us to delay edge detection until it is required (*late* edge detection). By delaying the edge detection phase we have found it possible to use edge detection as a fundamental tool in our image processing pipeline because only the areas of the image where edge detection is required will be examined for edges.

Comparing the results of edge detection in an objective manner is a difficult problem because ground truth is subjective, depending on the image and the observer [57]. Consequently we will compare our edge detectors with other algorithms only in terms of computational cost, and not in terms of quality. At the same time we assert, in a subjective way, that the algorithms we present here seem to work very well, and we present several images for comparison.

5.1 Optimised Edge Detection

Let I be a grey-scale image and $I(m, n)$ be the value of the pixel in the m^{th} row and n^{th} column. Then the Canny edge detector can be described by the

following 4 step process:

1. Use a Gaussian filter to reduce image noise and remove unwanted texture and detail:

$$I'(m,n) = G(m,n) * I(m,n) \tag{5.1}$$

where G is the Gaussian function

$$G(m,n,\sigma) = \left(\frac{1}{\sqrt{2\pi\sigma^2}}\right)^{\frac{n^2-m^2}{2\sigma^2}}. \tag{5.2}$$

2. Compute the magnitude $R(m,n)$ and orientation $\theta(m,n)$ of the gradient of $I'(m,n)$ via the x and y partial derivatives ($P(m,n)$ and $Q(m,n)$ respectively). We show the calculation here using a 2x2 window, but it can be extrapolated to whatever size window is desired:

$$\begin{aligned} P(m,n) &\approx (I'(m,n+1) - I'(m,n) + \\ &\quad I'(m+1,n+1) - I'(m+1,n))/2. && (5.3)\\ Q(m,n) &\approx (I'(m,n) - I'(m+1,n) + \\ &\quad I'(m,n+1) - I'(m+1,n+1))/2. && (5.4)\\ R(m,n) &= \sqrt{P(m,n)^2 + Q(m,n)^2}. && (5.5)\\ \theta(m,n) &= tan^{-1}(Q(m,n)/P(m,n)). && (5.6) \end{aligned}$$

It should be noted that $P(m,n)$ and $Q(m,n)$ can be calculated using any of the gradient operators (Roberts, Prewitt, Sobel [50]). We show here the operator used by Canny in the original thesis.

3. Compare each $R(m,n)$ with its two neighbours along direction $\theta(m,n)$. If $R(m,n)$ is greater than its two neighbours then $I''(m,n) = M(m,n)$ otherwise $I''(m,n) = 0$.

4. Finally, threshold $I''(m,n)$ to remove noise. $I''(m,n)$ is the final edge representation:

$$I''(m,n) = \begin{cases} I''(m,n) & \text{if } I''(m,n) > T \\ 0 & \text{otherwise.} \end{cases} \tag{5.7}$$

While the Canny edge detector works extremely well and is very popular, it is easy to see why it is unsuitable for our current needs. The entire process must be repeated for every pixel in the image. We note that some of the simpler methods such as Robert's Cross and Sobel are content with performing Step 2, computing $R(m,n)$ and thresholding on this value without ever considering the Gaussian blur in Step 1. However, the trade-off for this is either that the window size must be larger (Sobel), or the edge detector becomes very sensitive to noise (Robert's Cross).

Our edge detection uses an idea similar to Sobel's: that is, we also are content with Step 2 of the process outlined above and compensate by increasing the window size to five. However, to minimise the time spent in each window we only consider pixels in the same row and column as the centre (m,n). Thus the window we use is not a square but a cross. Ours is still a gradient computation: the difference in pixels at the centre of the window is computed and compared to the average difference between other pixels in both the row and column respectively. If the difference is much higher (compared to either the horizontal difference or the vertical one), then this pixel lies on an edge. The determining threshold value is the calibration for the edge-detector[1].

Although both our routine and the traditional edge detection algorithms are linear time, there is a big difference in the order of the hidden cost. In traditional edge detection algorithms, each pixel in the window must be compared to each of its neighbours. For a window size w, the constant in such algorithms is $2w(w-1) = O(w^2)$ (thus, quadratic on w, and usually very large). For example, a window size of five leads to 40 colour comparisons for each window evaluation and therefore 40 colour comparisons per pixel in the image. Contrast this to our technique where, with size w, the constant is $2w = O(w)$ (linear in w). This means that in the above example, our algorithm will only do 10 colour

[1] A C++ implementation of the edge detection algorithms described in this chapter is provided for reference in Appendix A.

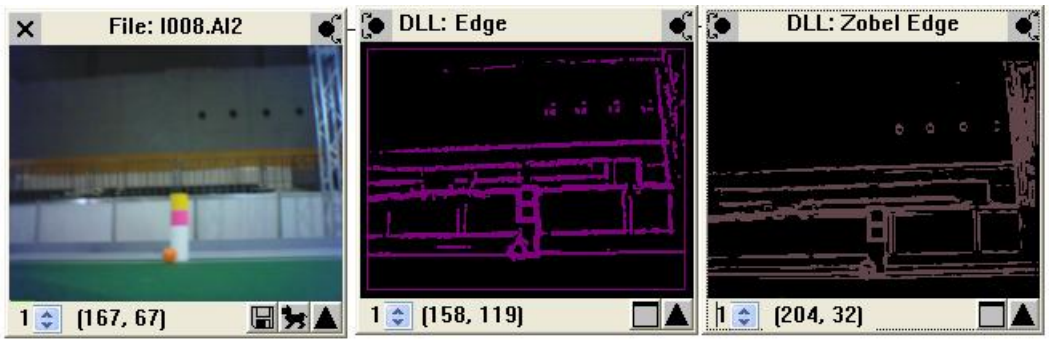
	Min Runtime (ms)	**Max Runtime (ms)**	**Avg Runtime (ms)**
Sobel	75	129	84.2
Our Algorithm	14	21	20.9

Table 5.1: The runtime performance of our edge detection algorithm compared to the very common Sobel algorithm on a set of 100 images with a window size of 5.

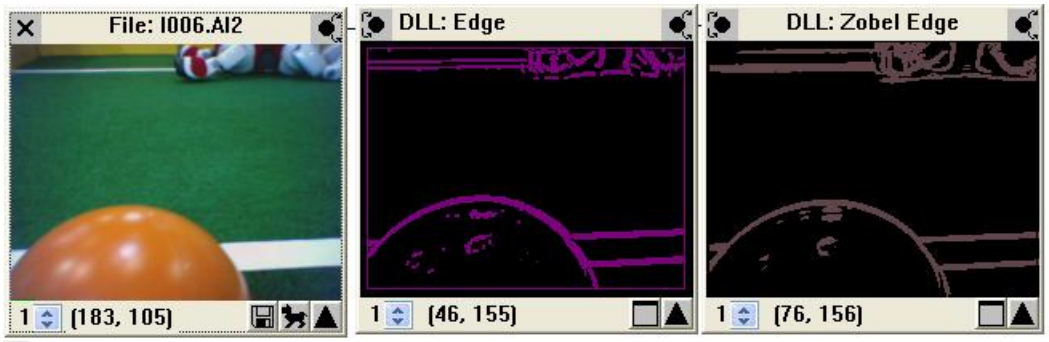
comparisons per pixel and will therefore execute in a quarter of the time of a traditional edge detection algorithm. There is a very minor trade-off in terms of the algorithm quality. Table 5.1 compares the runtime cost of our algorithm with that of a traditional algorithm (Sobel's), a set of 100 images with a (reasonable) window size of five.

Of course, as we have mentioned, comparison to ground truth is impossible as edges within an image are somewhat subjective. We present in Figure 5.1 several images showing the result of our method against Sobel for visual inspection.

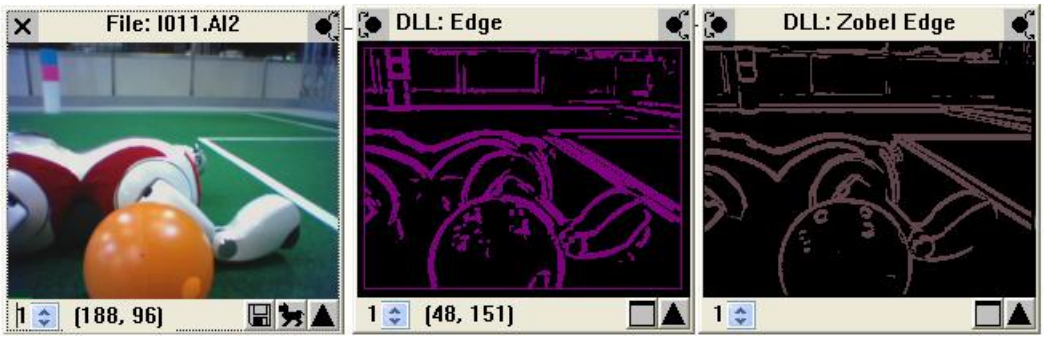
Any choice of edge detection algorithm must be implemented carefully to make it feasible to run in real-time. We emphasise an important optimisation that must be performed to make even our improved algorithm suitable. As the window slides over the image, we store and re-use colour differences that will occur in comparisons more than once (in fact, proportional to the window size). Consider the pixels at $(10, 10)$ and $(11, 10)$. The difference in colour between these two pixels is calculated for the first time when the window (of size five) is centred on $(9, 10)$ and used in the average to contrast with the difference between pixels at $(9, 10)$ and $(10, 10)$. As the window moves along the tenth row, this difference must be used five times. In a standard edge detection algorithm this difference would be computed on only the grey-scale image. However, we wish to use the entire colour-space (as we will discuss in Section 5.2.1) so the cost of each comparison is much higher. This expense arises because each comparison involves a three dimensional distance calculation (with a square root). We minimise the cost in two ways. Firstly, like Robert's Cross, we substitute Manhattan distance (which contains no floating point operations) for Euclidean distance. Secondly, we minimise this cost by keeping a circular buffer of the calculated differences between both the last five pixels and, looking ahead, the next five. Since we



(a)



(b)



(c)

Figure 5.1: A visual comparison of the results of our edge detection (purple) with the Sobel algorithm (grey).

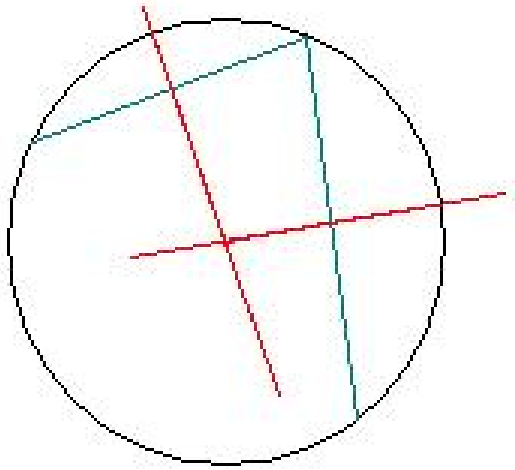


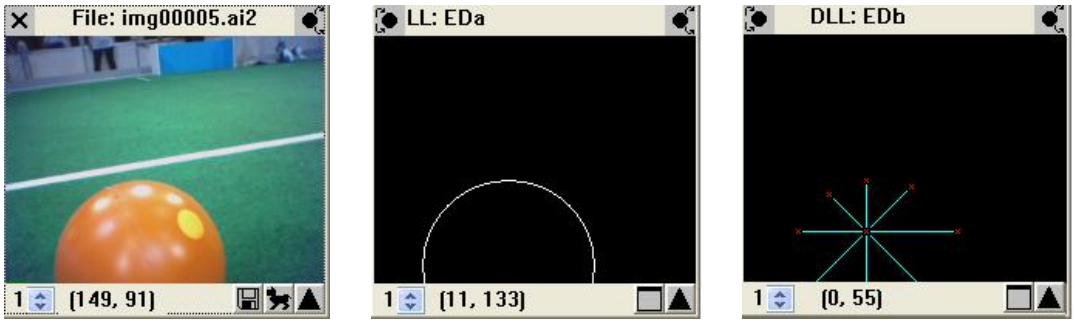
Figure 5.2: Vectorisation of a circle does not require all the edge points to be known. Any three points on the boundary of the circle can be used to locate its centre by the perpendicular bisectors method as shown in this figure.

iterate across rows first, and then columns, it is also necessary to maintain a circular buffer for each column in the image, in addition to the one for the current row. By using this technique we calculate the difference between each two pixels only once per pixel.

5.2 Late Edge Detection

Even given the above optimisations, edge detection remains very slow. One way that we can improve performance is to do edge detection only on sections of the image where it is required. The obvious problem is how to identify interesting sections of the image. Edge detection is usually an early step in the image processing pipeline so it is unlikely that we will have a large amount of contextual data on which to base such a decision. But, assuming that we can identify *some* points within the boundary of interesting objects, it is then possible to use edge detection to locate the edges of that object and use them for feature extraction.

We will discuss the identification of points within interesting objects in Chapter 8. For now, we will simply assume that we have identified $p = (x, y)$ as a pixel that is contained within an object for which we need to find the edges. We discuss here two techniques depending on the amount of edge information that



(a) (b) (c)

Figure 5.3: Full edge detection of an image (a) results in a full list of pixels that compose the edges in the image (b). Partial edge detection locates only a subset of these points (c).

is required.

Complete edge detection is a technique we use when a complete description of the edge of the object is required. This renders a traditional edge detection on a relevant object within the image. However, it is not always necessary to find the complete edge of each object as we will see in Chapter 8. For example, we are only required to know three points on the edge to find the correct parameterisation of a circle (see Figure 5.2). In this instance we use a *partial edge detection* that can find n points on the boundary without actually tracing the entire boundary. The difference in result between the two algorithms is illustrated in Figure 5.3. We describe the partial edge detection first, because the complete one will build on some of these techniques.

5.2.1 Partial Late Edge Detection

The idea of partial edge detection is that, given some seed point $p = (x, y)$ that we know to be within the boundary of an object, we wish to find a set E of n pixels that lie on the boundary. To do this we cast n evenly spaced rays out from p, examining each pixel on the ray as we come across it. Each pixel is compared with its neighbours to check the gradient change in intensity. By examining the

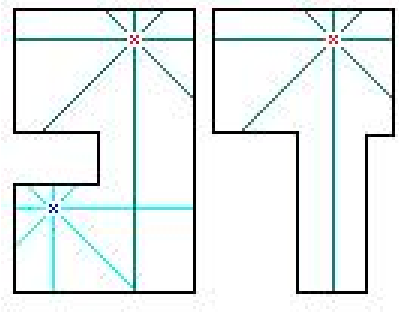


Figure 5.4: Partial edge detection on a convex shape (such as the ones in this figure) may not produce enough points to correctly identify and parameterise the shape. In this case further sample points may be used, or complete edge detection may be required.

pixels along a ray in this way, we are essentially performing a Sobel gradient comparison in the same way as we do when we use our cross shaped window. Edges are therefore detected well when they are approximately orthogonal to the ray — which is most of the time if the shape is convex.

Of course, if the shape is not convex then we may not be able to gather enough information to parameterise it in this way (see Figure 5.4). In this case, casting rays from a second point p_2 (or more) may sometimes be sufficient for parameterisation. More complex shapes will require a different method (which we will again examine in Sections 5.2.2 and 8.3).

Edge Detection on Blurry Images

One very useful feature of this technique is that we can apply edge detection to blurry images without the need for complex preprocessing (such as in Perona [102]), by incorporating colour data. Refer to Figure 5.5. Although a ray cast from p along the positive x axis will not contain sufficient data in the grey-scale channel to detect the edge of the ball, there is sufficient data present in the colour channels to compensate. As we progress along the ray, we compare each pixel intensity with its neighbours in the usual manner. This will detect any non-blurry edges. We also compare each pixel's U and V channels[2] with the *original* source pixel p. With the right comparison function and threshold this

[2]Any channels that carry colour information will do. For example, this corresponds to H and S in the HSI colour space.

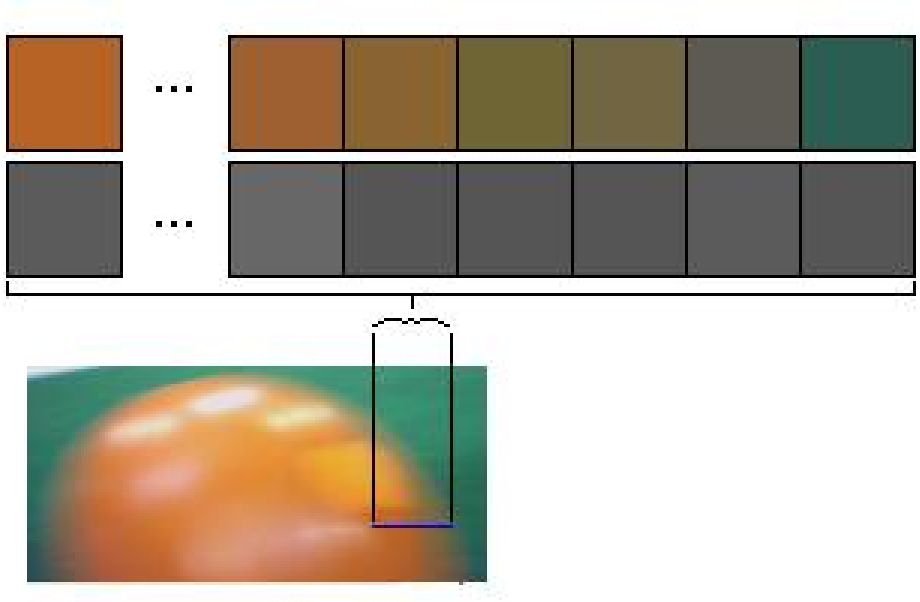


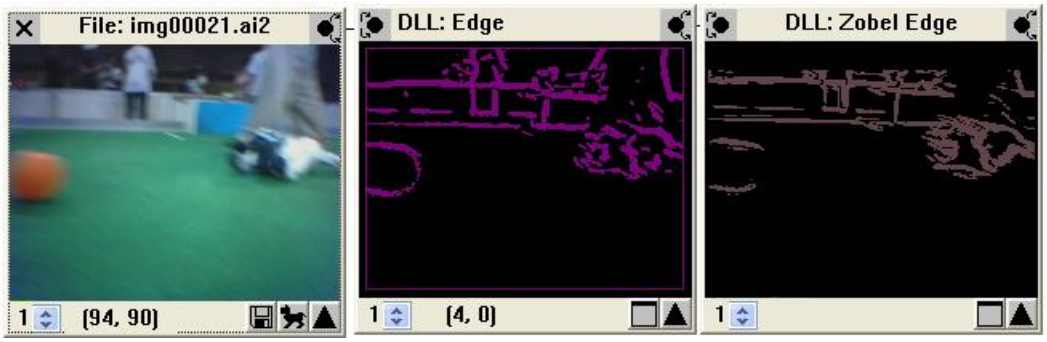
Figure 5.5: Edge detection is difficult on blurry images. This figure shows a pixel-by-pixel analysis of the horizontal line indicated in blue. The first row of pixels shows that there is significant variation along this edge (even though it is blurry) in the colour data. This would be almost impossible to detect by examination only of the Y channel shown in the second row of pixels.

will reveal any blurred edges because the colour information will change much more dramatically along a blur than due to natural changes of colour information within an object[3]. We have found a simple Manhattan distance function to be sufficient for our purposes:

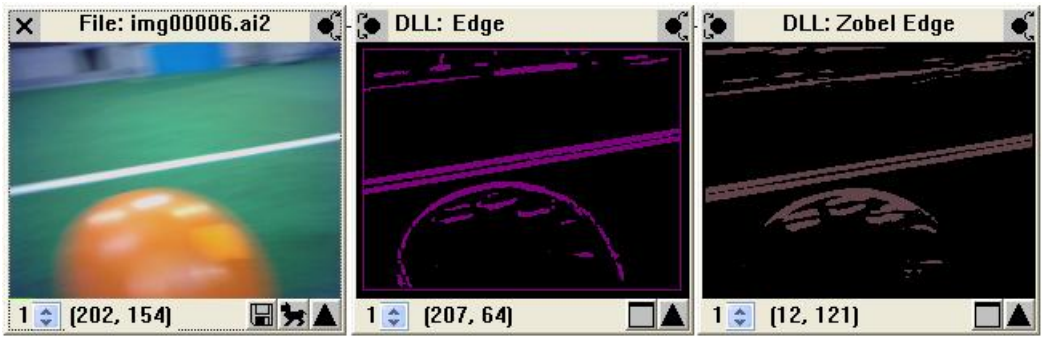
$$\Delta = |U_1 - U_2| + |V_1 - V_2|.$$

We present in Figure 5.6 several examples that illustrate the effectiveness of our edge detection on blurry images. Again we compare with the popular Canny algorithm.

[3]Of course, this presumes that our object is a single colour. In the general case this is a fundamental limitation of our technique.



(a)



(b)

Figure 5.6: Our edge detection algorithm (purple) remains effective even when images are blurred due to instablility in the camera. Traditional algorithms (such as Sobel, shown here in grey) have problems with blurry images.

5.2.2 Complete Late Edge Detection

Sometimes it is not adequate to know only a sample of the boundary points. For example, for vectorisation[4] we require an entire list of spatialy connected edge pixels that represent the boundary of an object. Although our method for this complete late edge detection is slower than a partial edge detection, it is still significantly faster than an early edge detection that must be performed on the whole image.

Let $B(p, I) \rightarrow p'$ be a standard border following algorithm that, given a pixel p on the border of an object in a raster image with borders marked I, returns the next pixel around the border of an object p'. Our late edge detection algorithm is then defined by Algorithm 5.1.

Algorithm 5.1 Complete late edge detection.

Input: A source pixel p that is within the spatial boundary of an object to identify in image (with no marked edges) I.

Output: The complete list of pixels E that make up the boundary of the object.

1: Trace *any* ray from p to find an edge as described in Section 5.2.1 and label this pixel s.
2: Let the current pixel be c.
3: **while** $c \neq s$ **do**
4: Apply any edge detection window to locate borders around c.
5: $c = B(c)$
6: **end while**

Of course in Line 4 we may use the window that we presented in Section 5.1. We show some images in Figure 5.7 that illustrate the results of this algorithm. The edge detection is complete in that a full list of pixels that compose the border of a particular object are discovered, but the algorithm does not need to examine any unnecessary pixels to do this. Irrelevant sections of the image are never examined because the algorithm uses a border follower.

One of the problems associated with this technique arises if the edge detector is unable to form closed contours. This issue can be somewhat avoided by a sensitive calibration (that is, one that forms thick edges). However, occasionaly we will be forced to abandon an attempt at identifying the edge of the object.

[4] See Chapter 6.

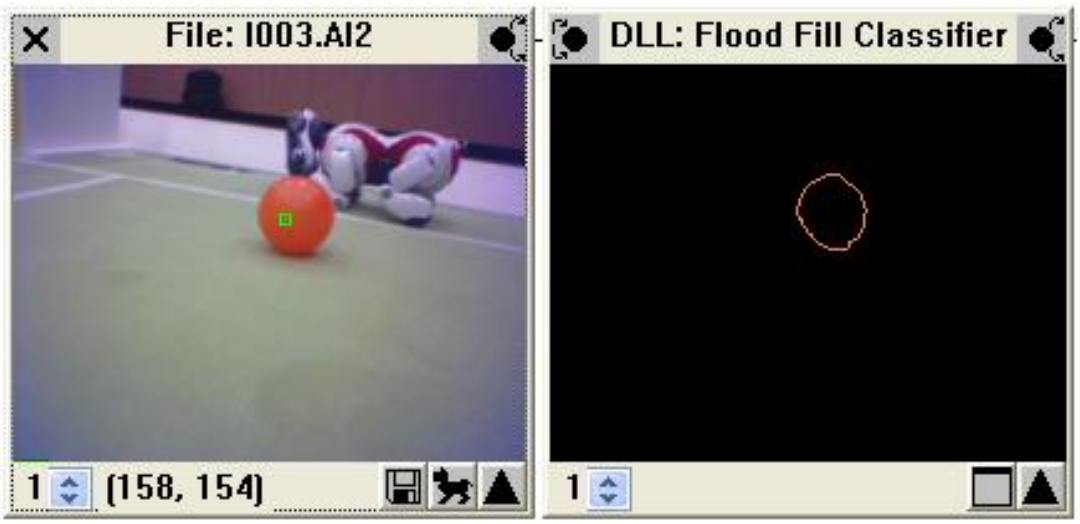


Figure 5.7: A complete late edge detection on a single object within an image reveals the boundary of that object, without examining any unrelated areas of the image. In this figure we show a complete edge detection on the ball, starting from the pixel indicated by green in the source image.

By bounding the pixels in the edge of each object we can abort unsuccessful attempts. We will leave a discussion of the impact and frequency of this for our chapter on object recognition (Section 8.4).

Border Following

There are many standard border following algorithms that we can apply to B in Algorithm 5.1, all of which are extremely fast ($\Theta(n)$ on the number of pixels in the border). We describe here one of the simplest for completeness. This is a standard algorithm that operates in 8-connected space, but it can be easily modified to work in 4-connected space.

Let $D(p, d)$ be a function that returns the next pixel from p in direction d. Let the directions be defined as in Figure 5.8.

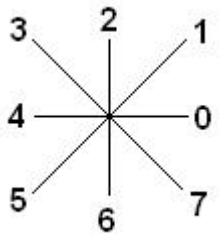


Figure 5.8: Direction definition for the border following method shown in Algorithm 5.2.

Algorithm 5.2 A simple 8-connected border following algorithm.

Input: Initial pixel p that lies on the border of an object in image (with edges marked) I. The direction returned from the previous call dir.

Output: The next pixel p' in a counter-clockwise direction around the border. The direction to use in the next call dir'.

1: Initialise $dir = 7$ on first call.
2: **if** $dir\%2 = 0$ **then**
3: $\quad dir = (dir + 7)\%8$
4: **else**
5: $\quad dir = (dir + 6)\%8$
6: **end if**
7: **while** p is not on border **do**
8: $\quad p = D(p, dir)$
9: $\quad dir = (dir + 1)\%8$
10: **end while**
11: $dir' = (dir - 1)\%8$

Chapter 6

Linear Time Vectorisation

The task of straight line vectorisation is that of converting a raster description of the edges within an image to a list of straight lines represented in vector form ($\vec{ot}$ where o is the origin and t is the target of the line). Therefore vectorisation algorithms pre-suppose an edge-detection step, which we examined in Chapter 5.

Algorithms for the task of vectorisation usually have the added complexity of determining not only the best vector representation of the points in the boundary, but also how many edges are present in the image and to which edge each point in the pixel list belongs. By far the most common method for this is the Hough Transform [65] which runs in $\Theta(n^2)$ for this task. There are heuristics which run more efficiently than this requiring $O(nlog_2(n))$ time [140, 61, 60]. We have discussed these methods in Chapter 3.

In this chapter we present an algorithm for the task of line vectorisation that improves the $O(nlog_2(n))$ runtime of Hershberger and Snoeyink [61] to $O(n)$. The basis for this improvement is a method that allows us to classify raster representations of edges as either *straight* or *non-straight* in $O(1)$ time rather than $O(n)$. Our algorithm for constructing the raster lists requires slightly more memory though still runs in $O(n)$ time.

Our procedure determines if a list of pixels is the raster representation of a straight line in constant time. It is embedded in a common procedure to build poly-line representations of boundaries. We first build ordered lists of pixels which traverse the boundaries in an image. For each list we determine quickly if it is a straight line or not and, if not, we divide the list in two. For each half we then determine if it is a straight line (recursively) until we locate all straight edges. This general algorithm is applied in many image processing tasks and

is commonly referred to as the Douglas-Peucker algorithm, especially in the Geographical Information Systems literature [140]. Its pseudo-code is presented as Algorithm 6.1.

Algorithm 6.1 The Douglas-Peucker straight-line vectorisation algorithm.

Input: Sequence E of pixels from pixel $o = (o_x, o_y)$ to pixel $t = (t_x, t_y)$.
Output: A vectorisation of the line represented by E.

1: **if** classifier says E is straight line **then**
2: return $\vec{ot}$
3: **else**
4: return Douglas-Peucker(first half of E) $\bigcup$ Douglas-Peucker(second half of E)
5: **end if**

The Douglas-Peucker algorithm is commonly improved by locating the pixel furthest from the straight edge connecting o and t while it iterates over the pixels, and then splitting the edge at that point (Line 4). Obviously we do not wish to interate over the entire list of pixels at each level of recursion so we will use the form presented in Algorithm 6.1. There is a minor improvement in the speed of the original Douglas-Peucker algorithm by using the more efficient form, but it does not offset the improvement we have gained by reducing the algorithm to $O(n)$ complexity. As we are not concerned with how many straight-edge segments we obtain in our vectorisation, there is no significant difference in the quality of the result obtained (refer to Figure 6.1).

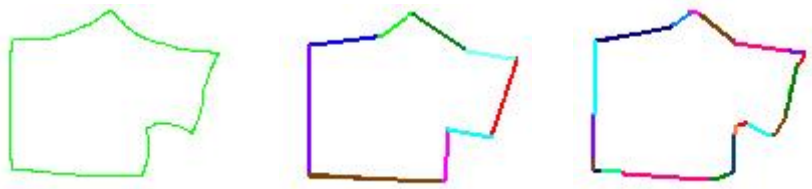
6.1 Definitions

Before we detail our method we must include some common definitions that we will use. Some of these are standard notation and are placed here for completeness, and some are new.

$\lceil x \rceil$ The ceiling of x. That is, the smallest integer above x.

$\lfloor x \rfloor$ The floor of x. That is, the largest integer below x.

$[x]$ The rounding of x. That is, the closest integer to x.



(a) (b) (c)

Figure 6.1: A comparison of our algorithm (c) with the original Douglas-Peucker algorithm (b) on the pixels that form the edge of part of an AIBO uniform (a). We are not concerned that our vectorisation finds more straight-line segments, so the final result is equivelant in quality, though not identical.

${}^{m}C_{n}$ The binomial coefficient *m choose n*. Some literature denotes this as either $\binom{m}{n}$ or C_n^m.

BLOCK A continuous row or column of pixels. Refer to Figure 6.2.

DIAGONAL The crux of two pixels that are 8-connected but not 4-connected. Refer to Figure 6.2.

Edge A list of 8-connected pixels from an origin point o to a target pixel t. Edges are composed of BLOCKS separated by DIAGONALS.

Optimal edge An edge is optimal if it contains the minimum number of pixels necessary to get from o to t. The number of pixels in an optimal edge will be equal to the number of pixels along the long side of the optimal box (see below).

Straight edge An edge is straight if it is the rasterisation[1] of a straight line from o to t. All straight edges are optimal, but not all optimal edges are straight. There is only one straight edge between any pair of points.

Optimal box The parallelogram with corners at o and t containing all optimal edges from o and t. Refer to Figure 6.3.

[1]Refer to the glossary on rasterisation on Page 183.

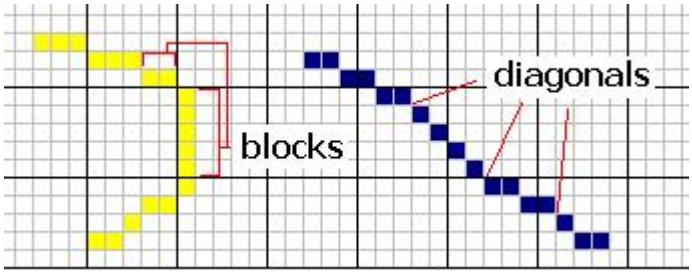


Figure 6.2: A block is a continuous row or column of pixels. A diagonal is the crux of two pixels that are 8-connected but not 4-connected. Edges are composed of blocks that are separated by diagonals.

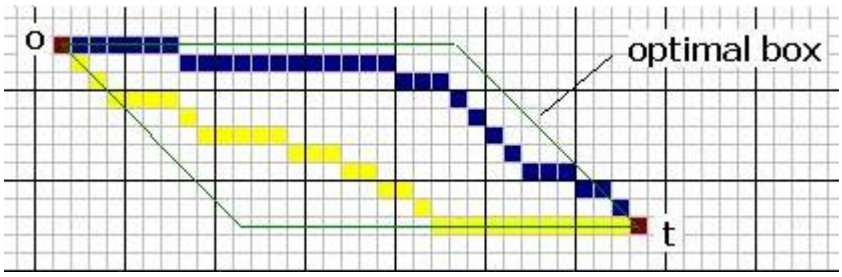


Figure 6.3: There is more than one shortest path (optimal edge) between most pairs of points (o and t) in 8-connected space. The lines shown here in blue and yellow are two such examples. The optimal box (shown in green) is the parallelogram that contains all such optimal edges.

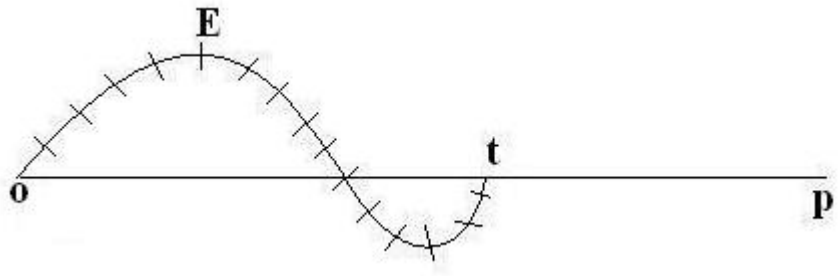


Figure 6.4: If the origin, o, and destination, t, of a journey are known, as well as the time taken, j (shown here as tick marks along E), then it is easy to determine if we have travelled in a straight line. If the distance from o to t is less than $j\|\vec{v}\|$ then we must have travelled along a curved path.

6.2 Classifying Line Segments

The basis for our improvement is the ability to classify a given list E of pixels, as either forming a straight line or a non-straight line in $O(1)$ time.

The intuition behind our algorithm can be illustrated by the following analogy. Imagine that we are travelling at a constant velocity $\vec{v}$ from a starting point $o = (o_x, o_y)$ to a target point $t = (t_x, t_y)$. If after j units of time we arrive at a final position t and find that the distance from o to t is less than $j\|\vec{v}\|$, we certainly did not move in a straight line. Consider Figure 6.4 where the tick marks along the curve from o to t represent evenly spaced points along that curve. If we know the first point and the last point of our list E and the number, n, of points in the list, then it is easy to tell whether the list represents the straight edge. The point p in the figure represents the expected endpoint of the straight edge with n units from o to t. Since the actual end point t falls far short of p, we conclude that E does not represent the straight edge. In fact, we not only determine whether the pixels in E form a straight line, but we have a measure, or indication, of how close they are to a straight edge (the distance from o to t approaches the distance from o to p as the pixels in E fall straighter).

Unfortunately the 8-connected space [32, 110] of pixelised images complicates this algorithm because many Euclidean properties do not hold — in particular the Pythagorean identity. In general there are many shortest paths that connect two points in 8-connected space. Figure 6.3 illustrates this. Both the blue and yellow lines are shortest paths from o to t and, in fact, any optimal edge is a

shortest path. The number of shortest paths in 8-connected space between o and t therefore relies on the gradient (slope) of the straight line connecting o to t. When this line is a horizontal or vertical line, or when it has a slope of 1 or -1, there is exactly 1 shortest path and the optimal box collapses to that line of pixels.

In general, there are ${}^{l}C_{s-1}$ shortest paths where $l = max(|o_x - t_x|, |o_y - t_y|)$ and $s = min(|o_x - t_x|, |o_y - t_y|) + 1$. This is because when travelling from o to t along a shortest path we must never increase the distance remaining to t so there are only two types of possible moves. Either we may step along the long axis of the optimal box or we may step diagonally towards t and cover one pixel on the short side of the optimal box at the same time as one on the long side. We must make s diagonal steps (or we will not end at t) and we have l opportunities to do it.

Although we cannot determine from this information whether a pixel list represents a straight edge, we can certainly rule out some lists. Any list that has more than the optimal number of pixels (l) does not represent a straight line segment. We also know that as the gradient approaches 0, 1, -1 or ∞ we are less likely to mis-classify using this heuristic (since there will be fewer possible shortest paths).

We now focus on determining if an optimal edge, E, (that is, a shortest path from o to t) is the straight edge (that is, the raster representation of the vector from o to t). Let us call the unique straight edge from o to t our target edge, T. That is, we wish to determine if a given edge E is equal to T. We are now able to make certain statements about the nature of the target edge. We assert the following propositions. Some of these are known properties of lines [36, 141], while we leave proofs of the other propositions for Section 6.3.

Proposition 1. *The target edge T that connects o to t in the optimal box must have the following properties:*

Claim 1: It will have l pixels where $l = max(|o_x - t_x|, |o_y - t_y|) + 1$ (the long side of the optimal box).

Claim 2: It will have s blocks where $s = min(|o_x - t_x|, |o_y - t_y|) + 1$ (the short side of the optimal box).

Claim 3: It must therefore have an average block length $a = l/s$.

Claim 4: There will be at most two block lengths in any rasterisation of a line: q long blocks of length $L_L = L_{\text{LONGBLOCK}}$ and p short blocks of length $L_S = L_{\text{SHORTBLOCK}}$. There may, of course, be the degenerate case where $L_L = L_S$.

Claim 5: If $L_S \neq L_L$ then $L_L - L_S = 1$.

Proposition 2. *Any rasterisation of a line follows a finite repeating pattern of blocks.*

Proposition 3. *The task of rasterisation is equivalent to the task of finding p, q, L_L and L_S ($L_L = L_S$ or $L_L - L_S = 1$) and distributing these blocks evenly between o and t.*

From these propositions we can derive certain qualities of our target edge, T.

The average block length is a, and we know that long and short blocks must be different in length by at most one. By this we can deduce that $L_S = \lfloor a \rfloor$ and similarly $L_L = \lceil a \rceil$. Let $k = \lfloor a \rfloor$. Thus $pk + q(k+1) = l = sa$ and, since there are s blocks in total ($p + q = s$) we can find p and q. Thus $q = l - sk$ and $p = s - q$. There will also be $\lfloor p/q \rfloor$ lines of length L_S for each line of length L_L and they must be evenly distributed. This set of properties, $I = \{s, t, p, q, L_L, L_S, \textit{the distribution of blocks}\}$ rigidly defines our target edge T.

This solves the classification task of determining if E is the target edge (that is, E is a raster representation of a straight line). However, computing I for a list of pixels E requires $\Theta(n)$ time. If we need to do this in every recursive call of the Douglas-Peucker algorithm we obtain $\Theta(n)$ cost per level of recursion because there are two calls in each level operating on half the number of pixels in E. The binary split gives $log_2(n)$ levels, so the total complexity would be $\Theta(nlog_2(n))$ in the worst case.

To avoid this problem we need to keep some data for the next recursive calls to be able to determine the set I in $O(1)$ time after the first call. We know, when we are building the edge, that if we encounter more than two different block lengths then it is not a straight edge[2]. This means we can avoid presenting our classifier[3] with any line which does not conform to the restriction of having

[2]We may allow for noise in the data by permitting a small variation in block lengths.

[3]The classifier is the procedure that tests if we have a representation of a straight line (Line 1 of Algorithm 6.1).

two block lengths where $L_L - L_S = 1$ or $L_L = L_S$[4]. We now only need to check that the blocks are distributed as we expect. We do this by comparing the standard deviation in distance between blocks of length L_L calculated for the target list T to the actual value calculated on the list E when we built it. We use a formula called the calculator formula for standard deviation[77] that requires only $\sum_{i=a}^{b} v_i$ and $\sum_{i=a}^{b} v_i^2$ and n (where v is the distance between blocks of length L_L and the interval of the list is a to b with n points). The calculator formula for standard deviation D is[5]:

$$D = \sqrt{\frac{(\sum_{i=a}^{b} v_i^2 - \frac{(\sum_{i=a}^{b} v_i)^2}{n})}{n-1}}.$$

Constant time is achieved because the calculation requires only that we know $\sum_{i=a}^{b} v_i$ and $\sum_{i=a}^{b} v_i^2$ and m. The sums $\sum_{i=0}^{k} v_i$ and $\sum_{i=0}^{k} v_i^2$ were calculated and stored for each pixel k in the list when it was constructed. If a recursive call operates on an arbitrary interval a to b then it requires $\sum_{i=a}^{b}$ for v_i and v_i^2. These values can be evaluated with one subtraction operation: $\sum_{i=a}^{b} = \sum_{i=0}^{b} - \sum_{i=0}^{a-1}$ for both v_i and v_i^2.

The complexity of the algorithm, including the recursive Douglas-Peucker, is therefore $\Theta(n)$. The algorithm is $\Theta(n)$ at the top level of recursion but no more than two operations at level 2, no more than four at level 3 and so on until there is no more than $2^{log_2(n)-1}$ at the deepest possible level. The sum $\sum_{i=1}^{log_2(n)} 2^{i-1}$ is less than $2n$ so the algorithm is $\Theta(n)$ time in the worst case.

Note that, although we have used a statistical formula, our method is deterministic and not statistical in nature. The distribution of blocks in the target edge will render a unique standard deviation. All other optimal edges will have a different distribution of blocks and therefore a different standard deviation between them. We only use the standard deviation to compare the distribution of blocks in a given edge, E, with that in the target edge, T. If the two distributions are equal then $E = T$.

[4]Note: the fact that a pixel list has two block lengths that differ by one does not make it a straight line. We can rearrange all the long blocks to one end of the line and all the short ones to the other to make a curve. We must take into consideration the distribution of the blocks in the line.

[5]We can increase or decrease tolerance on D to account for noise in the image.

6.2.1 An Alternative (Faster) Classifier

While $\Theta(n)$ time is impressive, it is in practice slowed down by the necessity of computing the standard deviations which involve time consuming floating point manipulations[6]. It is possible to improve this algorithm considerably (reduce the constant under the O notation, not improve the bound) by using a different measure to determine if the edge segment is straight when it falls inside the optimal box. The technique for this speed-up relies on the integral of the edge, as any straight edge from o to t divides the area of the optimal box exactly in two. Since we are required only to track $\sum_{i=0}^{k} y_k$ for each pixel k to calculate an integral between any two points in the list, this is a very fast and efficient method[7].

The drawback is that there exists a small number of edges that are not straight edges but are of optimal length and also possess the correct values for p, q, L_S and L_L and that also divide the optimal box exactly in two (see Figure 6.5). Notice the two block distributions in this figure. The non-straight (but optimal) blue edge divides the optimal box in two but the distribution of the blocks is incorrect. Our standard deviation method would detect this case.

In practice these cases turn out not to happen very often so this alternative classifier is often a better choice as it is both quicker than the robust one and requires less memory for each pixel. Although it is easy to think of example pixel lists where the integral classifier fails, our experiments show that this is not common in real life situations.

6.3 Proof

In this section we will prove the propositions stated in Section 6.2.

6.3.1 Proof of Proposition 1

Claims 1 and 2 are known properties of lines [141] and Claim 3 is evident from 1 and 2 (refer to Section 6.2). Thus our proof of Proposition 1 will show Claims 4

[6]There is a hidden constant of around 12 under the O notation.

[7]We still are required to make sure that the classifier is not presented with an edge that violates the $L_L - L_S = 1$ or $L_L = L_S$ relationship. Otherwise any edge that divides the area in two will classify as straight.

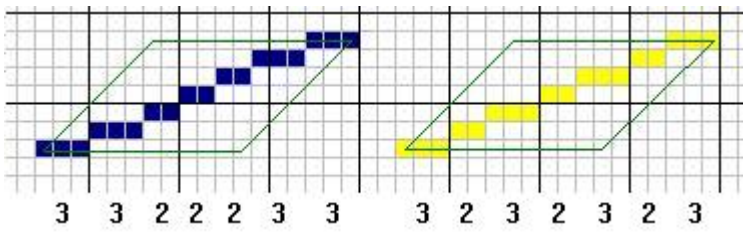


Figure 6.5: This figure shows a counter-case for the alternate straight edge classifier. An optimal edge with correct values for L_L and L_S will not necessarily be the straight edge, even if it divides the optimal box in two. The edge shown here in blue is optimal, has $L_L = 3$ and $L_S = 2$ and also divides the optimal box in two. It, however, is not the straight edge (shown in yellow). These situations are rare in real life images.

and 5. Furthermore we shall see that the proof of Claim 5 follows directly from the proof of Claim 4.

We wish to show that any rasterisation of a straight line has at most two block lengths: L_L and L_S where $L_L = L_S$ or $L_L - L_S = 1$.

Imagine that we have an arbitrary straight line drawn on a sheet of paper. Suppose now that we lay an overhead transparency which has been prepared with a regular grid of unit squares over the line. The process of rasterisation now consists of colouring the squares which the line passes through. This is a deterministic process that results in a raster that looks as close to a straight line as possible. That is, there is no swap, addition or removal of a coloured box that results in a closer approximation to the line that also leaves no gaps.

Without loss of generality, let us restrict the angle of the line relative to the grid squares to $0 < \theta < \frac{\pi}{4}$. The proof for all the lines with other angles [8] follows by symmetry (rotation and reflection). Figure 6.6 illustrates this process.

As we rasterise, we shall only colour at most one box in each column because $0 < \theta < \frac{\pi}{4}$ is more horizontal than vertical. In this setting it is not possible to have any vertical blocks of boxes forming part of the line. If the line only touches one box in a particular column then that box is coloured. When the line touches two boxes in a single column we must decide which one to colour by selecting the box in which the line segment is greatest. This is equivalent to selecting the top box if the line segment cuts it at a point greater than half way

[8] We treat 0 and $\frac{\pi}{4}$ as special cases in Section 6.3.4.

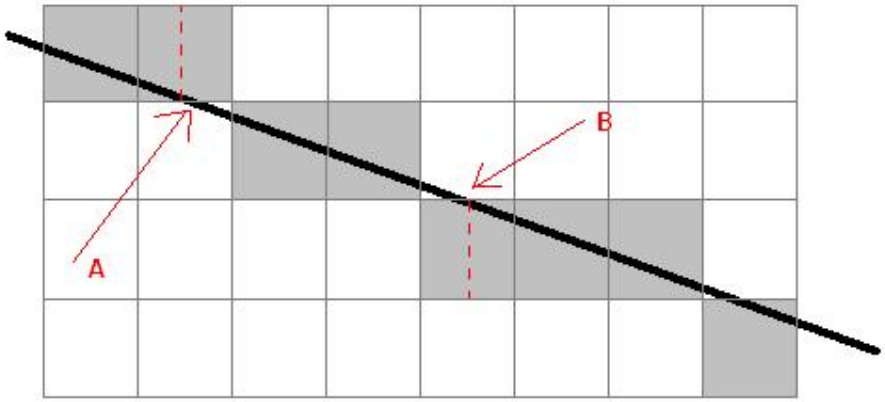


Figure 6.6: The task of rasterising a straight line is that of selecting which pixels to colour from the set of all pixels that the line touches. Pixels are selected if the line enters them from one side, or from less than half way along the length of the top. Pixel A was not selected because the line enters it more than half way along its length. Pixel B was selected because the line enters it less than half way along its length.

along its base, or the bottom box otherwise. If the line segment cuts it exactly half way along the base then the bottom box is arbitrarily selected. Again, refer to Figure 6.6.

It is now possible to derive an expression for the length of a block of pixels which will be a function of the angle and position at which the line enters its first pixel. Refer to Figure 6.7.

Suppose that the line enters the first pixel along the top at distance d from the left edge. Then $d < \frac{1}{2}$ or else the pixel would not be coloured. If the line enters the pixel along the left edge, this is equivalent to a negative value for d along the top edge (refer to Figure 6.7.) Again, $d \geq -\frac{1}{2}$ or the pixel before it would have been coloured. The value l is the distance from the start of the first pixel to the point where the line exits the row of pixels. We can therefore bound l:

$$\begin{aligned} d = -\frac{1}{2} : \quad l \geq \quad & \frac{1}{\tan\theta} - \frac{1}{2}. \\ d = \frac{1}{2} : \quad l \leq \quad & \frac{1}{\tan\theta} + \frac{1}{2}. \end{aligned}$$

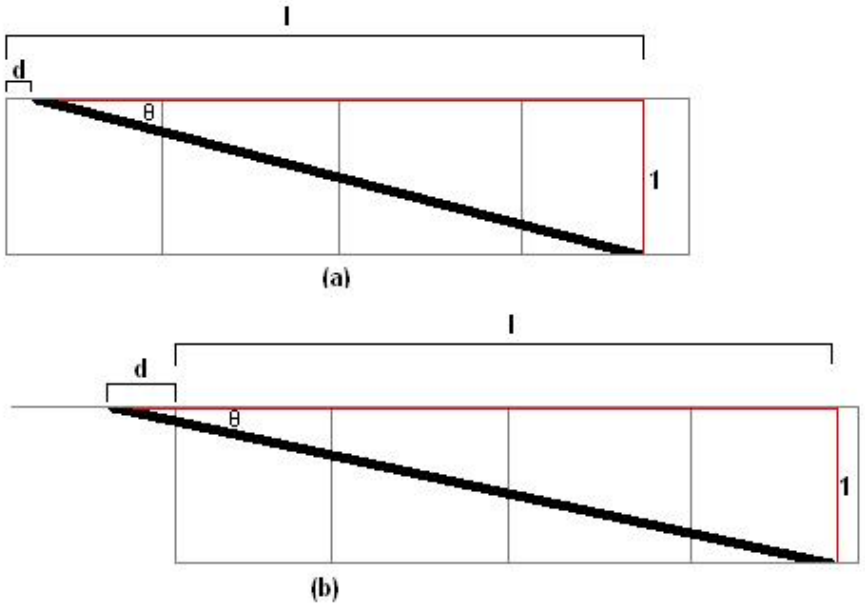


Figure 6.7: Deriving an expression for the length of a block. The line may enter the first pixel of a block from either the top (a), which gives a positive value for d, or the side (b), which gives a negative value for d.

The number, n, of pixels in the block is $[l]$ which we may also bound:

$$[\frac{1}{\tan\theta} - \frac{1}{2}] \leq n \leq [\frac{1}{\tan\theta} + \frac{1}{2}]. \tag{6.1}$$

This is true for all blocks in the line since the slope θ is constant with respect to the grid. Thus any block-length L must satisfy it with $n = L$. Since $n \in \mathbb{Z}$ and Equation (6.1) defines an interval of at most 1, there are at most two possible block lengths, $L_S = [\frac{1}{\tan\theta} - \frac{1}{2}]$ and $L_L = [\frac{1}{\tan\theta} + \frac{1}{2}]$ and if $L_L \neq L_S$ then $L_L - L_S = 1$.[9]

6.3.2 Proof of Proposition 2

We move on now to prove that the blocks must be distributed in a regularly recurring pattern. Before we can do this, however, we must show that we do

[9]We do not claim that two block lengths must exist. Consider the degenerate case of a line with $\theta = \pi/4$ and initial $d = 0$. Then every subsequent d will also be 0. Thus only one block-length (of 1) will exist.

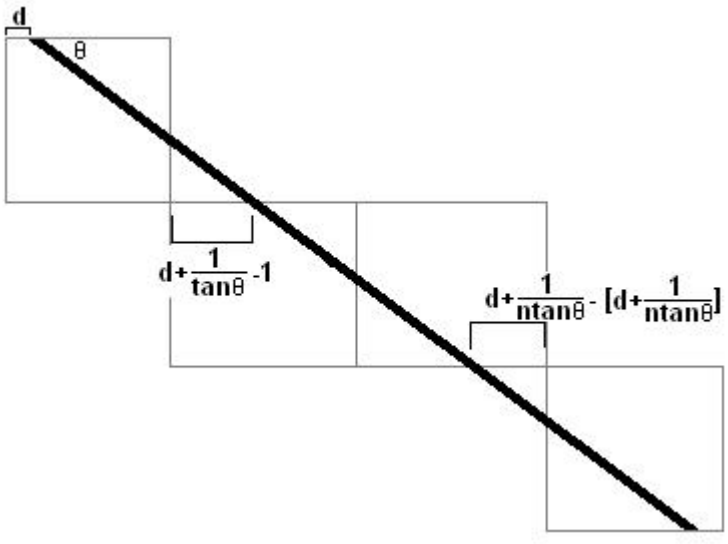


Figure 6.8: Any straight line rasterisation will have a repeating pattern of blocks. If a line enters the first pixel of a block at d, then the rasterisation will repeat if it can be shown to enter the first pixel of some other block at d also.

not lose generality by assuming $\tan\theta \in \mathbb{Q}$, or in other words, $\tan\theta = \frac{s}{l}, s, l \in \mathbb{Z}$. Obviously not very many values for θ make $\tan\theta$ rational. It has been shown that it is possible to replace every occurrence of a real number with a rational number to an arbitrary precision. Indeed, since computing devices are capable only of representing rational numbers this must be done. If a computing device is ever invented that is capable of representing (as well as operations on) arbitrary real numbers then our method will no longer be correct. We may, in that case, expect to see rasterisations of lines with no repeating pattern of blocks. While we confine ourselves, however, to digital computers and their rational representations of numbers, we are able to define $\tan\theta = \frac{s}{l}, s, l \in \mathbb{Z}$.

The points o and t given to the digital computer to define the line are such that $|o_x - t_x|$ and $|o_y - t_y|$ are integers. Therefore $\tan\theta = \frac{|o_y - t_y|}{|o_x - t_x|}$ is rational. From this definition we are able to derive L_L, L_S, p and q as in Section 6.2.

Suppose, as above, that the line enters the first pixel d from the left of the box at angle θ (again d may be negative if the line enters the left side of the box). The angle of the line will not change, so if we can show that it reaches another block at some later stage and enters it also at distance d from the left, then we must be in a repeating cycle. Refer to Figure 6.8.

If the line enters the first pixel along the top side at d from the left then it will enter the second block (not the second pixel) at $entry = \frac{1}{\tan\theta} + d - [\frac{1}{\tan\theta} + d]$. In general:

$$entry = \frac{n}{\tan\theta} + d - [\frac{n}{\tan\theta} + d].$$

We wish to prove that at some stage, for every θ, we will return to the original d. That is, we wish to show:

$$\begin{aligned} \frac{n}{\tan\theta} + d - [\frac{n}{\tan\theta} + d] &= d \\ \frac{n}{\tan\theta} &= [\frac{n}{\tan\theta} + d]. \end{aligned}$$

It is clear that this equation has infinite, periodic solutions if $\frac{n}{\tan\theta} \in \mathbb{Z}$ because $-\frac{1}{2} \leq d < \frac{1}{2}$. This is true if $\tan\theta \in \mathbb{Q}$ which we have shown to be the case. The rasterisation of the line therefore repeats the same pattern of blocks infinitely.

6.3.3 Proof of Proposition 3

We show now that the blocks within each repeating cycle will be evenly distributed.

We would not expect the rasterisation of a straight line, within the repeating pattern, to place all of the long blocks up one end and all of the short blocks down the other. In fact, we would expect the blocks to be exactly evenly distributed. We define an even distribution of p short blocks and q long blocks as a continuous sequence of blocks where at most $\lceil \frac{p}{q} \rceil$ short blocks are placed adjacently and at most $\lceil \frac{q}{p} \rceil$ long blocks are placed adjacently. We shall then prove that if we were to place either one too many short blocks or one too many long blocks together then we would produce a suboptimal rasterisation result.

Long Blocks

We aim to show that it is not possible to place $\lceil \frac{q}{p} \rceil + 1$ long blocks together without making the rasterisation result suboptimal. We prove this by contradiction. Figure 6.9 shows a block of length $\lceil \frac{q}{p} \rceil + 1$ and we show that such a block results in a contradiction. Figure 6.10 shows the last pixel in the block. If the pixel was

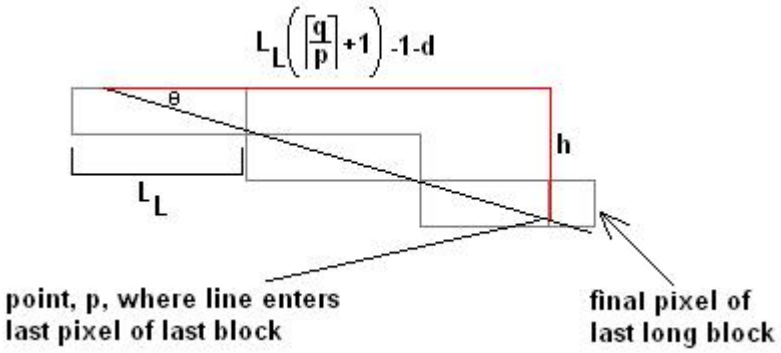


Figure 6.9: Placing too many long blocks together produces a suboptimal rasterisation result. The point at which the line enters the last pixel of the final long block will be less than $\frac{1}{2}$ and therefore the block should be short.

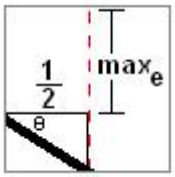


Figure 6.10: The distance max_e is defined as the maximum distance (from the top) at which the line can enter the side of the pixel in order to make this pixel part of the rasterisation. It is contingent on the angle of the line (θ). If the line enters at a distance greater than max_e from the top of the pixel then this pixel will not be part of the rasterisation.

truly necessary for that block, the line must cover it for at least half its length. We will show that it actually uses less than half of that pixel by showing that it enters its vertical side at a point further away than $max_e = 1 - \frac{\tan\theta}{2}$. This distance is an overall distance of $\lceil \frac{q}{p} \rceil + 1 - \frac{\tan\theta}{2}$ from the top of the first block. Trigonometry calculations show that the actual point at which it enters the last pixel of the final block is $(L_L(\lceil \frac{q}{p} \rceil + 1) - (1 + d)) \tan\theta$ from the top of the first block where $-\frac{1}{2} \leq d < \frac{1}{2}$.

Showing that this intersection is further away than max_e reduces to showing:

$$(L_L(\lceil \frac{q}{p} \rceil + 1) - (1 + d)) \tan\theta \quad > \quad \lceil \frac{q}{p} \rceil + 1 - \frac{\tan\theta}{2}. \tag{6.2}$$

Before we are able to show that Equation (6.2) holds, we summarise some

facts we have now proved:

1. We have shown that we do not lose generality by letting $\tan\theta = \frac{s}{l} \in \mathbb{Q}, s, l \in \mathbb{Z}$.

2. We know that $l > s$ and $\frac{l}{s} \notin \mathbb{Z}$. If $\frac{l}{s} \in \mathbb{Z}$ then there is only one block-length[10]. If there is only one block-length then the blocks must be distributed evenly.

3. We have shown that long blocks have length $L_L = \lceil \frac{l}{s} \rceil$ and short blocks have length $L_S = \lfloor \frac{l}{s} \rfloor = k$.

4. It follows (as in the derivation in Section 6.2) there will be $q = l - sk$ long blocks and $p = s - q$ short blocks.

Equation (6.2) now becomes:

$$\begin{aligned}
(\lceil \frac{l}{s} \rceil (\lceil \frac{q}{p} \rceil + 1) - (1 + d)) \frac{s}{l} &> \lceil \frac{q}{p} \rceil + 1 - \frac{s}{2l} \\
\lceil \frac{l}{s} \rceil (\lceil \frac{q}{p} \rceil + 1) - (1 + d) &> \lceil \frac{q}{p} \rceil \frac{l}{s} + \frac{l}{s} - \frac{1}{2} \\
\lceil \frac{l}{s} \rceil (\lceil \frac{q}{p} \rceil + 1) &> \frac{l}{s} (\lceil \frac{q}{p} \rceil + 1) + d + \frac{1}{2} \\
\lceil \frac{l}{s} \rceil - \frac{l}{s} &> \frac{d + \frac{1}{2}}{\lceil \frac{q}{p} \rceil + 1}
\end{aligned} \tag{6.3}$$

To show Equation (6.3) holds, we represent l in modulus form as $l = zs + r$ where z is the modulus of l and s, and r is the remainder:

$$\begin{aligned}
l &= zs + r. \\
\frac{l}{s} &= \frac{zs + r}{s}. \\
\lceil \frac{l}{s} \rceil &= z + 1. \\
\lfloor \frac{l}{s} \rfloor &= z.
\end{aligned}$$

[10]Refer to Section 6.2. The length of a long block is $L_L = \lceil \frac{l}{s} \rceil$ and the length of a short block is $L_S = \lfloor \frac{l}{s} \rfloor$. If $\frac{l}{s} \in \mathbb{Z}$ then $L_L = L_S$.

The last two statements are true because $\frac{l}{s} \notin \mathbb{Z}$. So:

$$\begin{aligned} \lceil \frac{l}{s} \rceil - \frac{l}{s} &= z + 1 - \frac{zs + r}{s}, \\ &= \frac{s - r}{s}. \end{aligned} \tag{6.4}$$

We may also derive a similar expression for $\lceil \frac{q}{p} \rceil$:

$$\begin{aligned} q &= l - s\lfloor \frac{l}{s} \rfloor, \\ &= l - sz, \\ &= r. \\ \lceil \frac{q}{p} \rceil &= \lceil \frac{q}{s - q} \rceil, \\ &= \lceil \frac{r}{s - r} \rceil. \end{aligned}$$

So:

$$\frac{d + \frac{1}{2}}{\lceil \frac{q}{p} \rceil + 1} = \frac{d + \frac{1}{2}}{\lceil \frac{r}{s-r} \rceil + 1}. \tag{6.5}$$

Substituting (6.4) and (6.5) back into Equation (6.3) we get the following target:

$$\frac{s - r}{s} > \frac{d + \frac{1}{2}}{\lceil \frac{r}{s-r} \rceil + 1}. \tag{6.6}$$

Note that $\frac{s-r}{s} > (d + \frac{1}{2})\frac{s-r}{s}$ because $-\frac{1}{2} \leq d < \frac{1}{2}$. Rewriting this we know that $\frac{s-r}{s} > \frac{d+\frac{1}{2}}{\frac{r}{s-r}+1}$. Because we have enlarged the denominator, we know that $\frac{d+\frac{1}{2}}{\frac{r}{s-r}+1} \geq \frac{d+\frac{1}{2}}{\lceil \frac{r}{s-r} \rceil + 1}$ (note that $d + \frac{1}{2} \geq 0$) and thus Equation (6.6) is also true.

We have therefore shown that if $\lceil \frac{q}{p} \rceil + 1$ long blocks are placed together we violate the rules of rasterisation.

Short Blocks

We use the same strategy in the case of short blocks to show that if we were to place $\lceil \frac{p}{q} \rceil + 1$ short blocks together then the line would *exit* the last pixel of the

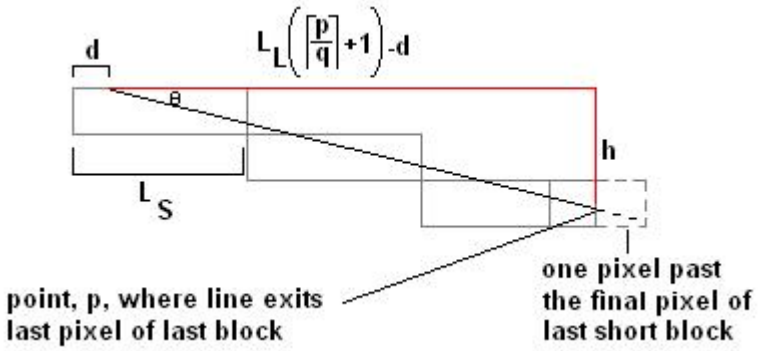


Figure 6.11: Placing too many short blocks together causes a contradiction with the rules of rasterisation. The proof is similar to the one for long blocks shown in Figure 6.9.

final block at a distance that would add another pixel to the block (thus making it a long block). Refer to Figure 6.11.

Again suppose that the line enters the first pixel of the first block along the top at distance d from the top left, $\frac{1}{2} \leq d < \frac{1}{2}$. We can calculate the point at which the line will exit the final pixel of the final block as being $(L_S(\lceil\frac{p}{q}\rceil + 1) - d)\tan\theta$ from the top of the first block. We would like this value to be *less than* the distance which would no longer require another pixel added to the block (refer again to Figure 6.10). That is, we would like it to be less than $\lceil\frac{p}{q}\rceil + 1 - \frac{\tan\theta}{2}$ so we wish to show:

$$(L_S(\lceil\frac{p}{q}\rceil + 1) - d)\tan\theta \quad < \quad \lceil\frac{p}{q}\rceil + 1 - \frac{\tan\theta}{2}. \tag{6.7}$$

This is equivalent to:

$$
\begin{aligned}
(\lfloor\frac{l}{s}\rfloor(\lceil\frac{p}{q}\rceil+1)-d)\frac{s}{l} &< \lceil\frac{p}{q}\rceil+1-\frac{s}{2l},\\
\lfloor\frac{l}{s}\rfloor(\lceil\frac{p}{q}\rceil+1) &< \frac{l}{s}(\lceil\frac{p}{q}\rceil+1)-\frac{1}{2}+d,\\
\lfloor\frac{l}{s}\rfloor-\frac{l}{s} &< \frac{d-\frac{1}{2}}{\lceil\frac{p}{q}\rceil+1},\\
z-\frac{zs+r}{s} &< \frac{d-\frac{1}{2}}{\lceil\frac{s-r}{s}\rceil+1},\\
-\frac{r}{s} &< \frac{d-\frac{1}{2}}{\lceil\frac{s-r}{s}\rceil+1},\\
\frac{r}{s} &> \frac{\frac{1}{2}-d}{\lceil\frac{s-r}{s}\rceil+1}. \qquad (6.8)
\end{aligned}
$$

Notice that $\frac{r}{s} > (d-\frac{1}{2})\frac{r}{s}$ because $-\frac{1}{2} \leq d < \frac{1}{2}$. Rewriting this we know that $\frac{r}{s} > \frac{\frac{1}{2}-d}{\frac{s-r}{s}+1}$. Because we have enlarged the denominator we know that $\frac{\frac{1}{2}-d}{\frac{s-r}{s}+1} \geq \frac{\frac{1}{2}-d}{\lceil\frac{s-r}{s}\rceil+1}$ (note that $\frac{1}{2}-d>0$) and so Equation (6.8) is also true.

We have therefore shown that if $\lceil\frac{p}{q}\rceil+1$ short blocks are placed together we violate the rules of rasterisation.

6.3.4 Special Cases

We have proved our three propositions for all lines with θ in the interval $0 < \theta < \frac{\pi}{4}$. We must now show that our propositions are true for lines with $\theta = 0$ and $\theta = \frac{\pi}{4}$. This is a trivial proof. Lines with $\theta = 0$ are horizontal and therefore contain only one block which is the size of the entire line. Therefore all of our propositions are true in this case. Lines that have $\theta = \frac{\pi}{4}$ are diagonal in which every pixel is its own block. In this case there is only one block-length (of 1) and therefore all the propositions are also true in this case.

Since now we have proved our propositions for all lines with θ in the interval $0 \leq \theta \leq \frac{\pi}{4}$, the proof for all other lines follows by symmetry (rotation and reflection).

Algorithm	Number and Size of Images	CPU time profiles (ms)		
		Best	Worst	Average
Algorithm 1	1000 (176x144)	82	181	124
	1000 (352x288)	204	426	297
	1000 (704x576)	501	932	613
$O(n \log n)$ version [61]	1000 (176x144)	99	211	139
	1000 (352x288)	251	498	312
	1000 (704x576)	507	1457	992
Hough Transform	1000 (352x288)	118	950	355
	1000 (352x288)	369	1286	863
	1000 (704x576)	975	3110	1862

Table 6.1: The performance of our vectorisation algorithm compared to our implementation of the $O(n \log n)$ algorithm and the optimized Hough transform.

6.4 Runtime Performance

We have demonstrated a method to classify a list of pixels as either *straight* or *non-straight* in constant time. This classifier when used in line 1 of the Douglas-Peucker algorithm results in the ability to locate straight edges within an image, and build vectorised poly-lines for the boundaries of objects, in linear time. Classical algorithms for this task run in $O(nlog_2(n))$ or even $O(n^2)$. While $O(nlog_2(n))$ to $O(n)$ does not seem to be a large improvement, in the case of image analysis for robotic vision, the algorithms must often perform well on inexpensive hardware, even for large n. Even a modest size image will have approximately 100,000 bytes of data so the improvement will be significant. Some test results are shown in Table 6.4 which compares the runtime of our algorithm over a series of images to that of an $O(nlog_2(n))$ implementation of an edge classifier[61] and an implementation of the Hough Transform. The images used in these tests were from our image archive where the number of straight lines in each image varies. The average, however, is approximately 4.

The performance increase obtained from the improvement from $O(nlog_2(n))$ to $O(n)$ time comes at the cost of some extra memory required to store information about each pixel in the edge, as the pixel list is built. The trade-off,

[9]Note that we do not see a direct correlation between the n in each runtime efficiency and the size of each image because the efficiency is calculated for n edge pixels not n pixels per image. We can, however, see an approximate correlation because a larger image will have more edge pixels. The same images (simply resized) were used for each set of tests to illustrate this.

however, is not large as each pixel requires only an extra two values to be stored. If the alternative, integral based classifier is used then only one extra value must be kept per pixel, in which case the cost of the trade-off is minimised.

Part II

Vision for Dynamic and Unpredictable Conditions

Chapter 7

Basic Concepts and Related Work

In this section we deal with the two areas where vision systems for mobile autonomous robotics must display versatility and the ability to cope with dynamic conditions. The first of these, robustness to variable illumination conditions, is emerging as the primary area of interest in robotic vision at this time. Surveys of recent conferences show that a large amount of research is currently being undertaken to solve the problem of how to adapt a vision system rapidly to unknown conditions. Research in all of the leagues in RoboCup is also following this trend. Every league either has a stated goal, or has some kind of plan to move away from controlled illumination conditions into natural light. For example, the Four-Legged league has run a variable illumination challenge for the previous two years while the Small-sized league has already moved to uncontrolled illumination conditions.

The second area in which vision systems for mobile autonomous robotics must be versatile is in its object recognition — particularly when posture or gesture recognition is necessary. Standard posture and gesture recognition systems assume that the viewed object is always readily identifiable in each image and is in some kind of standard position relative to the camera. A mobile robot cannot guarantee its own position in the environment, let alone relative to some other object which it has determined to analyse.

We will discuss both of these areas in turn.

7.1 Variable Illumination Conditions

Machine vision systems have been traditionally used well in environments where illumination conditions can be carefully controlled. One such example is that of electronics manufacturing where computer vision has been used extensively for over a decade to automatically examine circuitry for small flaws [101, 88]. In such an application, the circuit board will be placed in a known position in front of the camera, under known illumination conditions and compared to a reference image. While such a controlled environment is possible in the context of manufacturing robotics, once a robot is placed in a less predictable environment, the techniques applied to these controlled environments quickly become unusable. It is the ultimate goal of general purpose robotics research to develop robots that will be able to function alongside humans in the real world. This is truly a much more complex and dynamic environment.

7.1.1 Light Conditions

There are two dimensions that uniquely specify an illumination condition: light intensity and colour temperature. Light intensity is a measure of the *amount* of light that is reflected from a particular surface. It varies according to the intensity of the light source and in inverse proportion to the square of the distance between the surface and the light source. It is measured in lux[1]. The colour temperature of light is so named because as solid objects are heated they emit light[2]. A solid object heated to 1000^o Kelvin, for example, emits most of its light in the infra-red spectrum but some visible red light will also be present. As objects become hotter, the wavelength of the light emitted becomes shorter and therefore the colour changes. At 3000^o Kelvin, a bright yellow colour is emitted and at even higher temperatures the object will emit green and then blue light. Each wavelength in the colour spectrum of light is uniquely defined by a particular temperature. White light is the addition of light of every wavelength.

Lighting conditions may vary in an environment in several different ways.

Changing light source: The most obvious way an illumination condition may

[1] $1\ lux = 1\ lumen/m^2$. A lumen is the amount of light emitted along any particular angle by a light source of intensity 1 candella radiating equally in all directions.

[2] This process is independent of the material properties of the object, assuming the heat does not also catalyse a chemical reaction such as oxidation.

vary is if a light source changes — it may become brighter or duller, change colour, move its position or appear or disappear altogether. Of course the light source does not actually have to change, it only has to change *relative* to the robot. Illumination intensity varies in inverse proportion to the square of the distance to the source, so if the robot moves the illumination intensity will vary accordingly.

Natural (outside) environments encapsulate exactly such a light source. The sun moves across the sky in the course of a day and this does not simply alter the direction of the light source. As it moves across the sky different wavelengths of light are refracted through the atmosphere. This is why, towards the evening and morning the sky looks red while in the middle of the day it appears blue. Furthermore, in the early morning and late afternoon, the light from the sun must pass through a greater amount of the atmosphere to reach us (see Figure 7.1). This means that more of its energy will be absorbed by the atmosphere and therefore the intensity of the light that reaches us will vary. Although the sun is a constant light source, when coupled with the natural rotation of the earth and its atmosphere, the sun effectively is a light source that alters both in illumination intensity and colour temperature. This makes vision in natural lighting conditions difficult.

Specular reflection: In any visual scene there will be both direct light (from a light source) and ambient light. Any ambient light present is the result of one of the light sources being reflected on objects in the visual scene. The quality and colour of the ambient light therefore is largely determined by the properties of the objects in the environment. Highly reflective surfaces, for example a mirror, can effectively create another light source, while highly absorbent (black) surfaces will not reflect any light at all.

While we do not normally consider the effect of ambient light due to the presence of direct light, on occasion the reflective properties of objects in the environment can be quite important. Consider Figure 7.2 which shows a close up of a RoboCup beacon against a white background. Notice how the white background has a pink and yellow tinge to the colour around the edges of the beacon. This is due to the beacon's specular reflection of the

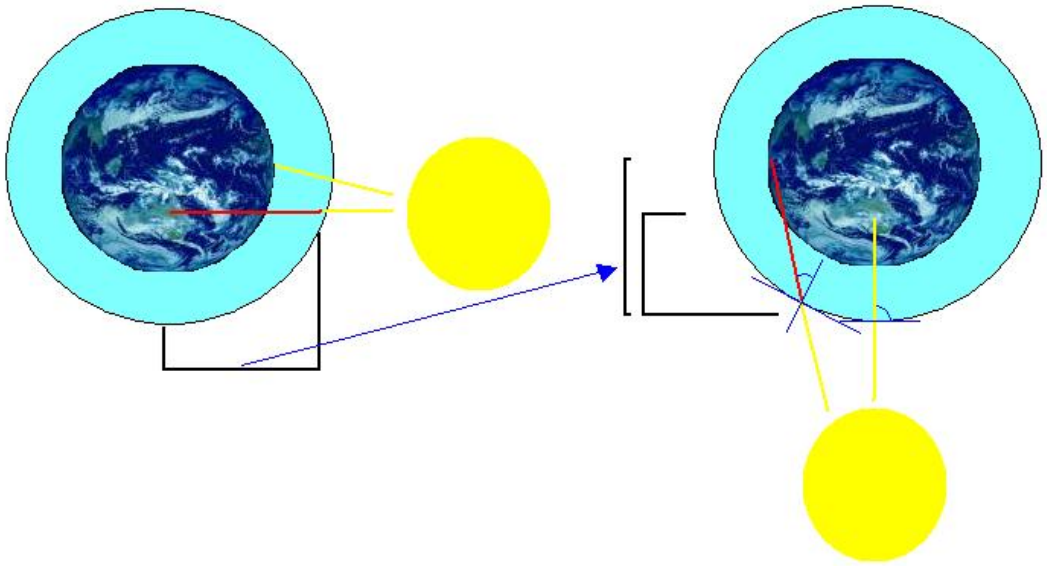


Figure 7.1: The rotation and atmosphere of the earth effectively make the sun a variable illumination source. In the morning or evening the light from the sun must pass through more of the atmosphere to reach us, making it less intense. Refraction in the upper atmosphere also absorbs some wavelengths of light while reflecting others making the sun appear to change colour (towards red) in the evening and morning.

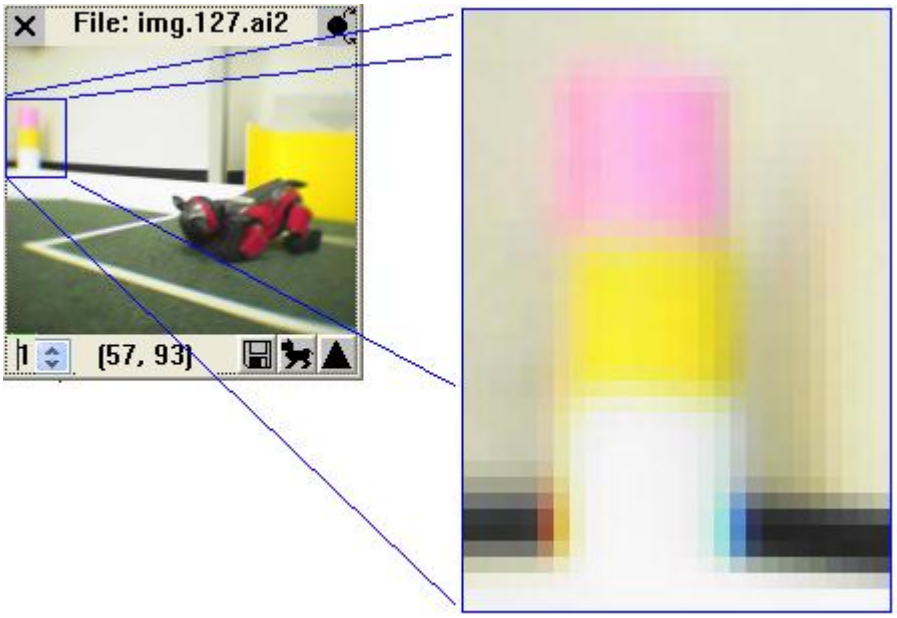


Figure 7.2: Specular reflection can alter the appearence of nearby objects. Notice the pink and yellow tinge on the white wall behind the beacon.

direct, overhead, lights. The beacon is effectively acting as a weak yellow and pink light source in this scene.

Shadows: The presence of shadows can obviously dramatically affect a perceived illumination condition. In effect shadows increase the importance of ambient light — an obstacle has blocked direct light from a source, so the only light reflected on our object will be ambient light. Of course, this makes the exact effect of shadows on a scene very unpredictable. It depends not only on the nature of the object casting the shadow, but also on the reflective properties of the other objects in the environment.

Although in most current research environments the presence of shadows is eliminated by controlled illumination, most real world environments routinely contain objects that cause shadows. Any vision system designed to work outdoors, for example, must allow for clouds to temporarily obscure the sun.

The Effect of Light Conditions on Perception of Colour

To detail adequately the properties of the link between lighting conditions and the perception of colour, it would be necessary to investigate the physical characteristics of the modern CCD device (digital camera). One should not simply assume that the camera is performing accurately in all lighting environments — especially if it has modern features such as auto-white balance, focus and shutter speed. In fact, the quality of a CCD device across different lighting environments is a largely subjective assessment anyway. Who is to say that each person's eyes (by which we judge the quality of images) function in exactly the same way? Research in this area is well beyond the scope of this thesis. We therefore restrict our discussion by assuming that the CCD device is performing accurately and the image our software is presented with is an accurate reflection of the environment, although we do recognise that this is a theoretical restriction.

There is a large body of work investigating the effect of lighting conditions on the perception of colour [83, 55, 13], and it has long been recognised that there is no linear relationship between a variable illumination parameter and the result on the perceived colour — no matter what colour space[3] is used. Some colour spaces were originally developed with the intention of decoupling the colour intensity from the perceived colour. The YUV and HSI colour spaces are examples of this. The thought was that a particular object should retain its colour properties under different illumination intensities. In other words, as the illumination condition became brighter or duller, the intensity value of a colour would change (Y in the YUV space, and I in the HSI space) but the two colour values would remain the same (U and V in YUV, H and S in HSI).

Ideally, given a colour value $C = (y, u, v)$ and an illumination condition, I, we would like a function $f : C \times Y \rightarrow C'$ that will transform C into the colour it would be in some standard condition (C'). Although the properties of the YUV and HSI colour spaces do reflect their ideal to a certain degree, they are no where near mathematically precise enough to be used for such purposes. Even if they were useful in such a way, as the illumination intensity in a scene drops, the importance of ambient light increases — indicating that the reflexive properties of the objects in the environment will play an important role in determining perceived colour. This is something that no colour space could compensate for. Figure 8.2 in Chapter 8 illustrates the effect of varying illumination conditions

[3]Refer to the Glossary on colour spaces on Page 181.

on the colour space.

7.1.2 Related Work

Several methods have been proposed to deal with variable illumination conditions [3, 100]. If there is some *a priori* knowledge of the geometry of the objects present within the environment then this can be used to identify the objects in a colour-independent way. The pixels in each object can then be examined and a classification automatically learned for the current lighting condition [21]. There is nothing particularly bad about this approach except, as we have noted in Chapter 3, geometric image processing is often a time consuming operation and so such an approach may be unsuitable for real-time robotics environments. If the objects in the environment are geometrically simple enough, this remains a viable method and, indeed, such methods have been used successfully to dynamically update colour calibrations [51, 49]. Such a process is a very popular method for self calibration in robotics. Self-calibration still occurs off-line but has the advantage that it does not require human supervision.

Others have improved on this technique by defining colour classes as sections of the colour space relative to some reference colour [71]. These techniques first assume that a reference colour can be tracked using the above geometric methods, and then assume that the variable illumination has affected all colours equally. Given these assumptions other colour classes may be tracked relative to the reference colour. It is a dangerous assumption to make, however, that different colours are affected in the same way by the same change in illumination. Euclidean distances in colour spaces can be stretched or compressed by a change in illumination condition [91].

If nothing is known *a priori* about the geometric properties of the environment then the traditional approach is to use a colour constancy method. Colour constancy refers to the invariance of perceived colour of different surfaces under variable illumination. The literature in this area is huge and goes back many years so we restrict our discussion to the major works and to those relevant to machine vision for robotics.

Several colour constancy approaches have been suggested where incident light is reconstructed and used to adjust the perceived colours of objects [9, 10, 80]. Forsyth proposes exactly this approach for machine vision in conditions where the illumination parameters are known (or measurable), but not necessarily

static [46]. Such approaches are statistical in nature because they treat colour classes as probability distributions within the colour space. Mayer *et al.* apply this approach to self calibration of a machine vision system [90], but no-one has yet successfully applied such an algorithm to a real-time vision system. This is because the calculations involved are extremely computationally complex. Even when processing power increases to be able to handle it well, the approach will not necessarily be effective on mobile, autonomous robots. It is relatively easy to measure the lighting conditions at the point in the environment where the robot is physically located through the use of on-board sensors, but how would the robot measure the lighting conditions at the physical location of the viewed objects? Furthermore the material properties (such as reflexivity) for each object in the environment could not necessarily be assumed.

Another colour constancy approach is to track the change in position of each colour class through the colour space as the illumination varies. In this way, these approaches are similar to those of Jüngel *et al.* discussed above [71] except that they do not require any prior geometric information concerning objects in a particular colour class. Such an approach can be shown to work well when illumination conditions vary steadily [4] and has the advantage that it avoids complex geometric calculations. However, the assumptions underlying this model of colour constancy make it difficult to apply to a robotic vision system. The first assumption is that the lighting conditions will never change dramatically — only gradually. For example, such a system would work well in an outdoor environment as the sun sets. Slowly the environment becomes darker and the colour temperature shifts towards the red end of the spectrum. It would not work well on a day when there were fast moving, small clouds in the atmosphere that repeatedly cast shadows and then moved on. It would not work well either in an indoor environment where lights can be suddenly switched on and off. The second assumption is that every single colour class will be present in roughly constant quantity in each image. How can you track a colour class through variable illumination conditions if there are no examples of the class in the image? An object that disappeared from view and then reappeared sometime later would be completely unrecognised by this type of system.

7.1.3 Our Contribution

Our contribution to illumination independent object recognition is a vision system that works quickly enough to run in the real-time processing environment of a mobile, autonomous robot, is independent of the illumination condition and able to adapt quickly to changing conditions. While our method does require a calibration, and does not provide complete illumination invariance, it is robust enough to changes in illumination intensity and colour to function under a wide variety of conditions without re-calibration. For example, the same calibration can be used to detect objects under direct sunlight, in natural shade, under fluorescent lights (blue colour temperature, low intensity) and under halogen lamps (yellow colour temperature, high intensity). Our method is not only illumination independent, it is actually computationally *less* expensive than the standard pipeline discussed in Section 1.2. We describe our method in Chapter 8.

7.2 Versatile Posture Recognition

Recognition of the posture and gesture of an object is an important task for many robotic vision systems. For example, domestic robots should be able to recognise and respond to human actions in their environments [44]. Competitive robots (as in RoboCup) should be able to respond to the actions of opponent robots. Accordingly, there are many techniques available in the literature for recognising the posture of a known object within a visual scene. Much of this work focuses on recognition of facial or hand gestures (for example, [20]) as there are immediate applications of this in many areas including user-interface design, face recognition and security monitoring systems. Indeed, this is a very important application for robotic vision systems as well because robotic interaction with humans is becoming increasingly common [12, 11]. It is not our intention to explore all of the literature in this area because it is simply too broad for the scope of this thesis. We instead address the work that is relevant to robotic vision systems.

We have noted that posture and gesture recognition is an important task for mobile robotics. Indeed, several commercial robots have successfully implemented facial, posture and gesture recognition systems in their software, for example the AIBO Mind software from Sony [4]. There are some important differ-

[4] http://www.aibo.com

ences, however, between these existing systems and what is required in an ideal robotic vision system. For example the AIBO Mind software can only recognise the face of its owner if the face is a particular distance from the camera and looking straight at it. Indeed, this seems to be a restriction of many posture recognition systems as well. For example, a security monitoring system at an airport monitors crowds of people, examining people's behaviour for anything suspicious [143]. However, in this system the camera is fixed and all people are viewed from the same perspective. If the camera is above a doorway then people are also always viewed from the same distance and orientation as they pass through the doorway. A mobile, autonomous robot that is free to move around its environment will not necessarily have such a fixed perspective. Gesture and posture recognition are difficult without a fixed perspective and distance to the viewed object.

Another problem with implementing gesture and posture recognition is that work in this area tends to be very domain-specific. For example, recognising a facial gesture (posture) relies on domain knowledge such as how to locate the eyes and nose within an image [40]. It is not our contention that these tasks could be made simpler — in fact, it is hard to see how you could perform these tasks without the computational expense associated with the extraction of complex features such as the eyes and nose. Although sometimes complex feature extraction is necessary, we postulate that because these techniques are so well used and known, many simpler methods get overlooked when the recognition task is itself more simple. For example, we will show in Chapter 9 that it is not necessary to extract the features of a coffee cup handle to detect the pose of a coffee cup relative to the viewer. This task can be performed at a lower level.

Even at a high level, posture recognition can often be simplified by performing a low-level analysis before high-level feature extraction is performed. One technique for performing exactly the same task as our main example — recognising the postures of Sony AIBO robots for the task of playing robot soccer — relies on learning the shapes of coloured patches on the soccer uniform of the AIBO [29, 136]. There are other techniques for posture detection as well. Some are based on perimeters of objects [116] and some on the comparison of surfaces to other planar surfaces (circles, squares etc.) [40]. Each of these techniques has in common a complex pre-processing phase that must be performed in order to extract the necessary features (usually lines and edges). Some tasks do require

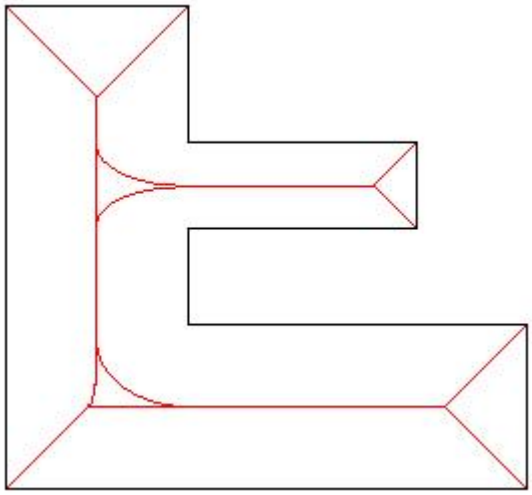


Figure 7.3: The medial axis of an object is the set of points that are equi-distant from any edge.

high-level processing. Others, however, benefit from a lower-level posture recognition algorithm. Shapes on an AIBO's soccer uniform, for example, are much more readily identified by symmetry than by edge analysis and the posture of an AIBO can be determined much more quickly this way. Simple hand gestures too, are more inexpensively identified by symmetry than by finding features such as knuckles and fingers.

7.2.1 Posture Recognition by Symmetry

Symmetry is an important property of many real-world objects. When viewed from certain perspectives, many objects appear (relatively) symmetrical while, when viewed from a different orientation, they do not. Therefore the degree of symmetry in an object is often a useful indicator as to its relative posture [74]. However, in order to analyse the symmetry of a raster image, some computationally useful representation of the image must be obtained. The most common such technique is the medial axis [79, 28] although others have been suggested [122, 142].

The concept of a medial axis is fairly intuitive to understand. Conceptually, the medial axis of a polygon is the set of points that are equi-distant from any edge. In Figure 7.3 the medial axis of the black polygon is shown in red.

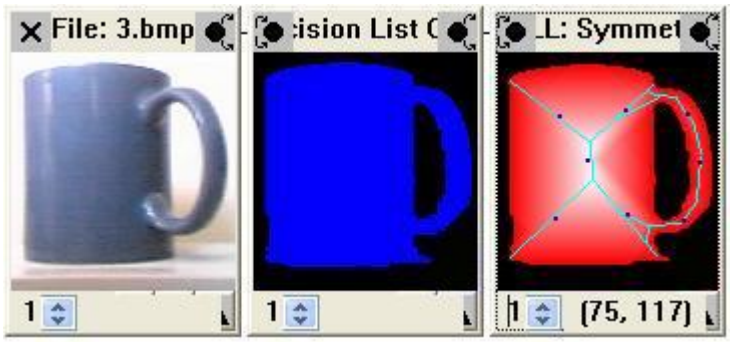


Figure 7.4: The medial axis can be found by a distance transform as shown here. Each pixel classified blue in the middle image is assigned a value based on its distance to the edge of the blue blob. Higher values are shown in the right image in white, lower values in red.

There are several ways to compute the medial axis. It may be found by a pixel stripping method [28], or similarly, by a distance transform on the edges of the polygon [79]. Refer to Figure 7.4 for an example of this method. The distance transform of the cup is shown in the red image on the right, where the distance of each pixel from the edge of the polygon is indicated by its colour. You can also use a vectorisation of the boundary of the polygon to inexpensively compute the medial axis using the set of maximally inscribed circles [30]. Of course, this requires an efficient vectorisation algorithm, such as the one we have presented in Chapter 6.

Once the medial axis has been found (or some other simplified representation of the object), there are many suggested methods for using it for the task of posture and gesture recognition [128, 41, 134].

7.2.2 Our Contribution

Our contribution to this field of posture and gesture recognition is an algorithm that works by symmetry and is capable of quickly determining an object's relative pose and gesture. It works independently of where the object is located in the environment — provided it is close enough for sufficient detail to be present in the image, and it is computationally inexpensive, allowing it to be used in real-time vision systems. While our method is certainly not a general solution to the posture recognition problem, it does provide a computationally inexpensive,

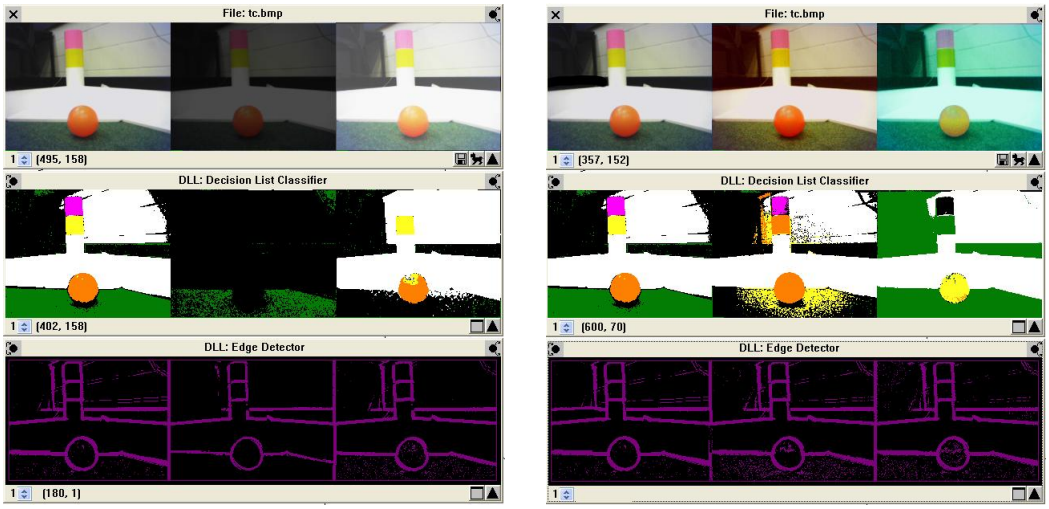
perspective independent system that can be used in a wide variety of situations. We first illustrate our technique using the tasks of analysing the posture of opponent robots in a game of robotic soccer. Secondly we will perform posture recognition of hand gestures, and finally we apply our method to the analysis of maritime signalling flags. Even though these are widely varied domains, all lend themselves to our symmetry analysis approach to object recognition. We describe our method in Chapter 9.

Chapter 8

Illumination-Independent Object Recognition

In this chapter we introduce an efficient object recognition system that is robust to changes in colour intensity and temperature. To do this, we build on the work we presented in Part I of this thesis, particularly that of optimised edge detection (Chapter 5) and sparse colour classification (Chapter 4). The aim of our algorithm is to obtain a description of the boundaries of objects detected in the image. We have already shown one way that such a list will be useful in the task of object recognition in Chapter 6. We will demonstrate another in this chapter.

Our algorithm will provide the basis for object recognition in a wide variety of illumination conditions without re-calibration. It can do this because it does not rely solely on colour classification in order to form blobs. We have presented a sparse classifier (Section 4.1) that is very accurate, but only on pixels that are at the core of each colour class. It essentially refuses to make a decision for most other pixels, labelling them as UNKNOWN. We use this in combination with the late edge detection methods we presented in Section 5.2 to enable us to find the border of objects without performing either a robust classification, or a complete edge detection. We present two variations of our method. One variation runs extremely quickly but is only able to find simple shapes. The other variation is slightly more processor-intensive but will recognise an arbitrary shape.



(a) (b)

Figure 8.1: This figure shows the result of varying illumination intensity (a) and colour temperature (b) on a common scene in RoboCup. The top row of images is the source image, the middle row is a robust classification and the bottom row is our edge detection routine. Note that although variation in illumination conditions is detrimental to colour classification, it does not particularly affect edge detection.

8.1 The Basis of Illumination Independence

It is well accepted that edge detection algorithms are far more robust to changes in the temperature and intensity of light than colour based segmentation algorithms. This is easy to illustrate. Consider Figure 8.1 which illustrates the effect of varying the illumination intensity (a) and temperature (b) of a common scene in RoboCup. Although the edge information in the images is not lost until the illumination levels become extremely low, the robust colour calibration becomes useless quite quickly. We have seen in Chapters 4 and 7 how fragile colour calibrations can be.

It is apparent then that edge detection, rather than colour segmentation, is a better basis for object recognition systems if robustness to dynamic lighting conditions is a requirement. However, edge detection *on its own* will not yield sufficient information quickly enough. For example, to find the ball in the binary edge images in Figure 8.1 we would need to run a circle detection algorithm such

as the Hough transform[1]. This would be far too slow for our purposes. Instead, our method works by combining edge detection with the sparse classification technique introduced in Section 4.1.

8.1.1 Training a Sparse Classifier for Illumination Independence

We have argued in Chapter 7 that there is no way to train a robust classifier for variable illumination conditions. We may, however, train a sparse one as long as there is not *too* much variation in the conditions. Figure 8.2 shows how this is possible. As the lighting conditions vary, the perceived colour changes in a non-predictable way ((a) and (b)). However, as long as there is some overlap we may classify *only* the pixels in the overlap section as ORANGE (c). We therefore have found a *core* class ORANGE that contains pixels that are perceived as orange across both images. The classification itself (c) would be poor if we were to use it for object recognition purposes, but we do not wish to use it directly in this way.

We see now how it is possible to train a sparse classifier for dynamic illumination conditions. We simply widen our set of training images to include many images from different lighting environments and, in the manner described above, train a sparse classifier to label only the pixels that are at the core of each colour class. Of course if the lighting conditions are *too* variable then the intersection operation of the sparse classifier (Equation 4.4) will yield an empty set. In this case we should be less ambitious with our illumination conditions.

8.1.2 Combining Edge Detection with Sparse Classification

One initial algorithm for illumination-independent object recognition is shown in Algorithm 8.1. In this algorithm we use the list of labelled pixels as seed points to find the border points of each object in the image. Although we can not be sure that *every* pixel that is part of, for example, the ball, will be labelled as ORANGE, we can be certain that *some* of them will. Therefore to find the border points of an object we start at a seed point and iterate over pixels in

[1] See Chapter 6.

(a) (b) (c)

Figure 8.2: This figure shows the result of varying the illumination conditions on the colour space. For both images (a) and (b) a perfect calibration was made and every pixel that was labelled ORANGE is plotted in the three dimensional YUV space. Notice that the location of the orange colour class shifts in the space as the illumination changes. We can find the *core* of the colour class orange (c) by only classifying pixels that are labelled ORANGE in both images. This leads to a poor robust classification, but a good sparse classification.

any direction until a previously identified edge is found (Line 4). Once we find an edge we may border trace using the algorithm in Section 5.2.2 to obtain the entire list of pixels in the border.

Algorithm 8.1 Basic illumination-independent object recognition.

Input: An image I.

Output: A list of the border points E_o for each object o in I.

1: $EdgeDetect(I)$
2: $SparseClassify(I)$
3: **for all** classified pixels p in I **do**
4: Iterate across pixels in any direction until an edge pixel e is found
5: **if** e does not belong to the border of a known object **then**
6: $BorderTrace(e)$ to obtain a list of border points E
7: Assign new object o with edge E, E_o
8: **end if**
9: **end for**

This process renders an image segmentation that works in a similar fashion to other region-growing techniques [133] except that it has the advantage of being illumination-independent. Refer to Figure 8.3 where each row in the table shows the process working under a different illumination condition (but with the same calibration file). The first picture is the source image for the row, the second shows the results of a sparse classification step, the third shows the result of Algorithm 8.1 and the final image shows the object recognition step. In each case the ball, goal and beacon are found correctly, despite the large variation in illumination.

The disadvantage of this method is the runtime cost. Refer to Table 8.1 for a breakdown in the runtime cost of this algorithm, compared to the Bruce *et al.* pipeline we discussed in Chapter 1. Most of the execution time is taken in the edge detection step. Therefore we would expect that optimising the edge detection step will improve the runtime cost of our algorithm. We have already presented our methods for this improvement in Chapter 5 so we present here two algorithms that build on that work. Our algorithms significantly improve the efficiency of Algorithm 8.1 to make it feasible for use in real-time robotic environments.

Notice that Algorithm 8.1 is not fast enough to analyse 30 fps on an AIBO.

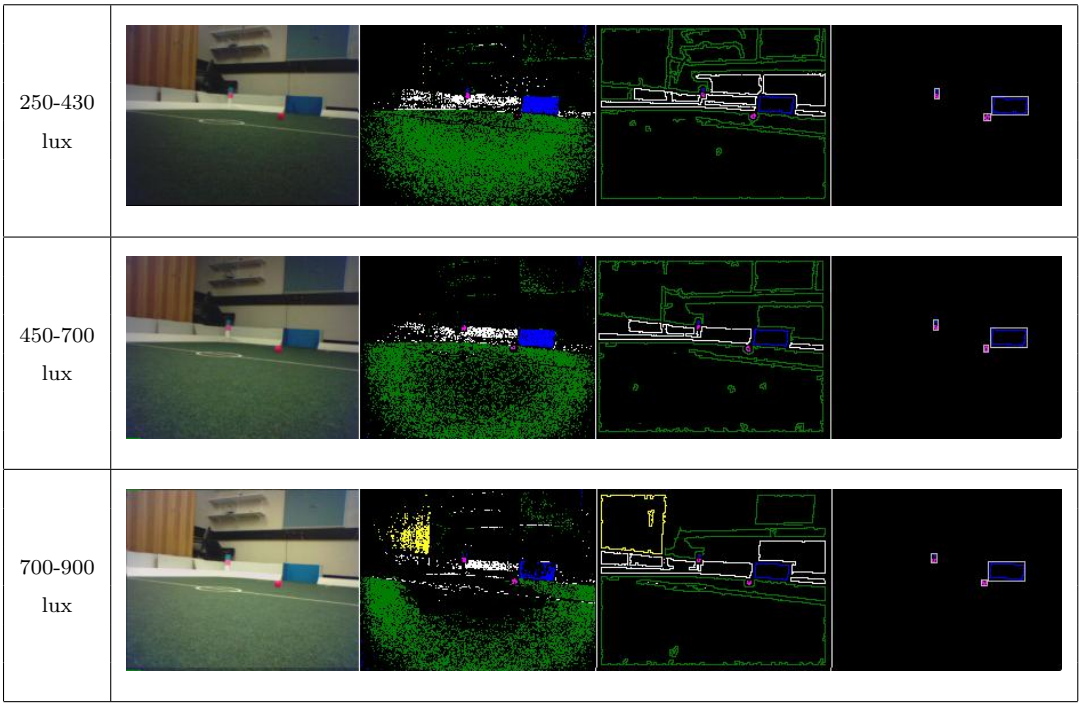


Figure 8.3: The basic algorithm for illumination-independent object recognition. A sparse classification (column 2) is used along with a border-following algorithm (column 3) to locate regions within the image. These regions can be analysed for objects of interest (column 4). Note that the system uses the same classification in each illumination condition with accurate results.

	Component	Average Time on AIBO (ms)
Our basic pipeline	Sparse classification	1.71
	Edge detection	20.90
	Border following	17.71
	Object recognition	5.87
	Total per frame	46.19
	Time taken for 30 frames (1s of data)	1385.70
The Bruce *et al.* pipeline	Robust classification	2.42
	Blob forming	9.13
	Object recognition	6.21
	Total per frame	17.76
	Time taken for 30 frames (1s of data)	532.80

Table 8.1: The runtime cost of basic illumination-independent object recognition is quite high compared to the Bruce *et al.* pipeline discussed in Chapter 1. A large component of this cost is in the edge detection and border following steps; therefore optimising these steps will improve the runtime cost ofour algorithm.

In fact, 30 frames analysed at this speed takes 1404.3ms, or just under one and a half seconds. Clearly, a large part of the cost of the algorithm is associated with the edge detection and border following steps. Therefore, we propose two modifications to Algorithm 8.1 in order to make it execute quickly enough to use in a mobile, autonomous robotic environment. Both of our alternatives use the late edge detection technique that we introduced in Chapter 5. By delaying the edge detection step until it is required we can perform edge detection only on the sections of the image in which we need it.

8.2 Detecting Simple Object Boundaries

The first of our alternatives uses the partial late edge detection that we described in Chapter 5. There are several cases where we do not require a complete description of the boundary of an object in order to parameterise it, as occurs with most simple shapes. The ball in RoboCup is a good example because it is circular. We require only three points on the boundary of the ball in order to apply the geometrical technique of perpendicular bisectors to find the centre and

radius[2]. Therefore we need only to cast three rays in different directions from a seed pixel p that we are sure is part of the ball. Of course, we may wish to cast more rays, or to start with more than one seed point, in order to check if we have actually found the ball or just some other object that also looks orange. This is an extremely fast ball-finding algorithm.

There are other shapes as well that can be found in this way. We present in Algorithm 8.2 our fast algorithm for determining the parameters of a rectangle provided the angle of orientation is known *a priori*. If the angle of the horizon in the image is known then this algorithm can be applied to find (for example) the beacons or goals in RoboCup or a great many rectangular shaped objects in the real world.

We use the method outlined above to find a set of points, P, that are on the boundary of the rectangle. The points are then rotated in space so that the rectangle is rectilinear (Line 2). Each vertical column and horizontal row are then examined to find the lines on which the most points in P lie (Lines 6-21). These lines are labelled as the sides of the rectangle (Line 23). Finally the four corners are rotated back to the frame of reference of the image (Line 24). If the aspect ratio of the rectangle is also known *a priori*, the algorithm adjusts the rectangle based on the side that had the weakest support value (S). Similar algorithms may be employed to detect any regular polygon.

If enough original sample points are chosen in the initial set of points P, then the algorithm is quite tolerant to noise in the image and even eliminates points where the edge has been determined incorrectly (see Figure 8.4 in the next section).

8.2.1 Runtime Performance of Simple Object Detection

In Table 8.2 we examine the runtime performance of simple object recognition[3]. The most expensive components of this pipeline are in edge point detection and shape recognition, but the pipeline is still a dramatic improvement on the basic pipeline in Algorithm 8.1. Indeed this pipeline is not only illumination-independent, it is *faster* than the Bruce *et al.* pipeline. We could potentially

[2]We discussed this technique previously in Section 5.2.

[3]Note that the two object recognition components in Table 8.2 should not be directly compared. Much of the work that happens in object recognition in the Bruce *et al.* pipeline is performed in the shape recognition phase in our pipeline.

Algorithm 8.2 Efficient rectangle parameterisation.

Input: A set of points P that lie roughly on the boundary of a rectangle of unknown aspect and size but known orientation, θ.

Output: The parameters of the rectangle R as four corner points — p_{tl}, p_{tr}, p_{bl} and p_{br}.

1: **for all** $p \in P$ **do**
2: Rotate P by angle $-\theta$
3: **end for**
4: Find the bounding, rectilinear rectangle of P and assign to R.
5: Let $maxS_1 = 0$, $maxS_2 = 0$
6: **for all** x in R **do**
7: Let $S =$ the sum of points that lie on or near x
8: **if** $S > maxS_1$ **then**
9: $maxS_2 = maxS_1$
10: $maxS_1 = S$
11: **end if**
12: **end for**
13: Let $x_1 = MIN(maxS_1, maxS_2)$, $x_2 = MAX(maxS_1, maxS_2)$
14: Let $maxS_1 = 0$, $maxS_2 = 0$
15: **for all** y in R **do**
16: Let $S =$ the sum of points that lie on or near y
17: **if** $S > maxS_1$ **then**
18: $maxS_2 = maxS_1$
19: $maxS_1 = S$
20: **end if**
21: **end for**
22: Let $y_1 = MIN(maxS_1, maxS_2)$, $y_2 = MAX(maxS_1, maxS_2)$
23: $R = \{(x_1, y_1), (x_2, y_1), (x1, y2), (x2, y2)\}$
24: Rotate R by angle θ around the origin.

	Component	Average Time on AIBO (ms)
The Bruce *et al.* pipeline	Robust classification	2.42
	Blob forming	9.13
	Object recognition	6.21
	Total per frame	17.76
	Time taken for 30 frames (1s of data)	532.80
Our pipeline using only simple object recognition	Sparse classification	1.71
	Edge point detection	6.71
	Simple shape recognition	5.29
	Object recognition	1.64
	Total per frame	15.35
	Time taken for 30 frames (1s of data)	460.50

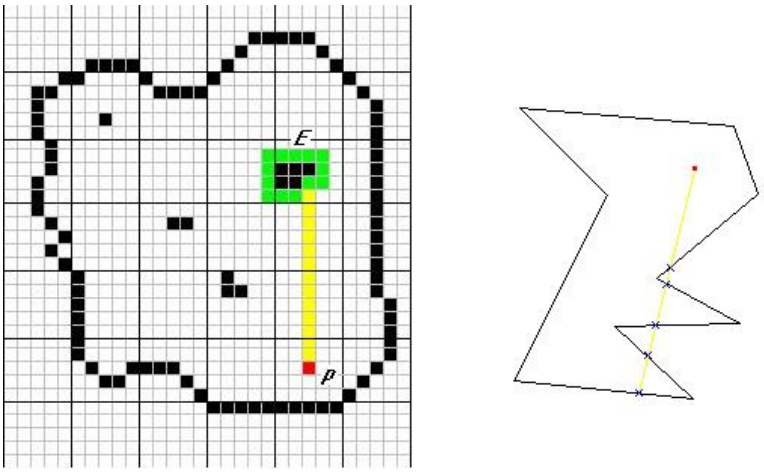
Table 8.2: By using our partial late edge detection in conjunction with the sparse classification, our object recognition pipeline is not only illumination-independent, it executes *more* quickly than the Bruce *et al.* pipeline.

analyse 60 fps at this speed.

8.3 Detecting Complex Object Boundaries

The second of our alternatives uses the complete late edge detection that we described in Section 5.2.2. The basis of this method was that a point on the boundary of an object would be found in the same way as above, casting a ray from a seed pixel p until we reach the edge of the object. We noted in Section 5.2.2, particularly in Figure 5.4, that it may not be enough to simply cast rays from seed pixels to determine the parameters of a complex shape. We therefore proposed a combination of a boundary-following algorithm and a partial late edge detection that could be used to discover and trace the boundary of an object in linear time, Algorithm 5.1.

Algorithm 5.1 will, under most conditions, return a list E of pixels that represents the boundary of the object in which the pixel p resides. Due to noise in the image, on occasion, it is possible for E to locate an island, rather than the object. Refer to Figure 8.4 (a) where the yellow pixels illustrate the ray cast from p (the red pixel) and the green pixels indicate the edge, E, that has been



(a) (b)

Figure 8.4: One of the main problems with our method is that "islands" can be found and traced instead of the border of the object (a). We use a standard polygonal interior test to detect this situation (b).

located. The correct boundary of the object has not been found in this case due to noise in the image.

We may use the standard polygonal inclusion test [98] to detect this situation (Figure 8.4 (b)). A ray cast from any point to a point on the edge of a polygon cuts the polygon an odd number of times if that point is *inside* the boundary of the polygon. We wish to see if our original point p is inside the polygon represented by E. Therefore the algorithm is simple. As E is constructed (Algorithm 5.1) we count the number of times a pixel $q = (x, y)$ is placed in E with $q_x = p_x$ and $q_y < p_y$. If this number is odd then the original point p is inside the polygon E, otherwise it is outside.

Another problem that may be encountered due to noise in the image is that the edge detector does not return a closed contour: that is, there may be gaps in the list of pixels that comprise the edge of the object. If the edge detector is calibrated to be sensitive (that is, for thick edges — refer to Section 5.1) then this is not a frequent problem, as the results in Section 8.4 illustrate. When it does occur it is possible to detect the situation by bounding the number of pixels that may comprise the edge of an object. In this case we discard the edge entirely and the object is missed in that frame.

Once the boundary has been correctly determined, it is usually useful to vectorise it. This can be done in linear time using the algorithm that we presented in Chapter 5. Once a vectorisation has been obtained there are several useful and fast analysis algorithms. We will leave our discussion of these techniques for Chapter 9.

8.3.1 Runtime Performance of Heterogeneous Object Detection

Most object recognition tasks will blend a mixture of objects that can be detected using simple object recognition, and objects that must use complex object recognition. Table 8.3 shows the runtime cost of such a system. Images in these tests were from the RoboCup domain and had a mixture of balls, beacons and goals (representing simple objects) and opponent AIBOs (representing complex objects).

We make several observations on Table 8.3, particularly with respect to the introduction of complex objects (AIBOs). In order to analyse AIBOs we must identify their skeleton (as we will see in Chapter 9) so the processing that was done on blobs (in the Bruce *et al.* pipeline) and on complex shapes (in our pipeline) is enough to identify a complete and ordered list of pixels along the border of the AIBOs' uniforms. We do not, however, include a full AIBO recognition algorithm in either pipeline.

This processing represents one extra step for the Bruce *et al.* object recognition component than for our pipeline. This is reflected in the slower processing time for object recognition for the Bruce *et al.* pipeline than was seen in Tables 8.1 and 8.2. Our object recognition has no extra overhead because this work is done in the complex shape recognition component.

Different sample images were used for this set of tests (so that AIBOs could be included) so minor variations from the data in the above tables are to be expected. Classification is marginally slower in these images because there are more potential colours to classify and therefore a longer decision list.

We see from this table that our pipeline is essentially equivalent in speed to the Bruce *et al.* pipeline, being less than 1% slower. Therefore we have managed to provide an illumination-independent object recognition system for minimal extra cost.

	Component	**Average Time on AIBO (ms)**
The Bruce *et al.* pipeline	Robust classification	2.73
	Blob forming	8.99
	Object recognition	14.71
	Total per frame	26.43
	Time taken for 30 frames (1s of data)	792.90
Our pipeline	Sparse classification	2.51
	Edge point detection	7.98
	Simple shape recognition	5.06
	Complex shape recognition	10.44
	Object recognition	1.34
	Total per frame	27.33
	Time taken for 30 frames (1s of data)	819.90

Table 8.3: The final runtime cost of the two pipelines including analysis of both simple and complex objects. Our pipeline executes less than 1 ms/frame slower than the Bruce *et al.* pipeline but provides an illumination-independent object recognition system.

8.4 Accuracy of our Method

In this section we discuss the accuracy of our object recognition methods presented in this chapter. We have collected 500 images that contain scenes that may be expected in a typical RoboCup game. The image database is divided into two sections. The first 200 of the images are taken from a moving AIBO (and thus contain varying degrees of blur), the other 300 are taken from an AIBO standing still. The images vary in lighting condition (200-1500 LUX, with dynamic shadows) though they are similar enough that a suitable sparse classifier has been found (see Section 8.1.1).

Table 8.4 shows the accuracy of our system. There are several things to note here. Firstly, the system performs well — especially given the variable lighting conditions. Even in images taken from a moving camera the system performs well enough to use as the primary sensory input of a soccer playing robot.

The second thing to notice is that we may determine the extent of the closed contour problem (Section 8.3) by comparing the accuracy of our system on simple objects with the accuracy achieved when we detect these same objects uning our algorithm for complex objects. Typically, we only fail to detect objects due to this problem in less than 10% of images taken from a stationary camera. This accuracy is sufficient for the posture recognition task we will describe in Chapter 9. We fail to detect objects approximately 25% of the time when the images contain blur. This is to be expected — complete edge detection in blurry images remains a difficult problem.

Table 8.4 shows only the positive accuracy (that is, objects in each image that were correctly identified). There were an insignificant number of false positives — less than 10 for all objects in all images — however, this result is not significant to the discussion in this chapter. It is the object rules that we describe in Chapter 11 that rule out false positives.

	Object	Occurences in 300 Images	Recognised (simple)	Accuracy (%)	95% Confidence Interval	Recognised (complex)	Accuracy (%)	95% Confidence Interval
Stationary Camera (300 Images)	Blue Goal	123	117	95.12%	89.77% to 97.15%	109	88.62%	81.80% to 93.10%
	Yellow Goal	119	114	95.80%	90.54% to 98.19%	107	89.92%	83.20% to 94.14%
	P/B Beacon	57	56	98.25%	90.71% to 99.69%	51	89.47%	78.88% to 95.09%
	B/P Beacon	83	79	95.18%	88.25% to 98.11%	76	91.57%	83.60% to 95.85%
	P/Y Beacon	34	31	91.18%	77.04% to 96.95%	29	85.29%	69.87% to 93.55%
	Y/P Beacon	62	57	98.39%	91.41% to 99.71%	54	87.10%	76.55% to 93.31%
	Orange Ball	212	203	95.75%	92.13% to 97.75%	192	90.57%	85.88% to 93.81%
	Red AIBO	87	-	-	-	63	72.41%	62.23% to 80.71%
	Blue AIBO	112	-	-	-	86	76.79%	68.16% to 83.64%
Moving Camera (200 Images)	Blue Goal	97	86	88.66%	80.83% to 93.55%	73	75.26%	65.82% to 82.77%
	Yellow Goal	86	76	88.37%	79.90% to 93.56%	68	79.07%	69.32% to 86.33%
	P/B Beacon	52	45	86.54%	74.73% to 93.32%	31	59.62%	46.07% to 71.84%
	B/P Beacon	35	29	82.86%	67.32% to 91.90%	23	65.71%	49.15% to 79.17%
	P/Y Beacon	39	34	87.18%	73.29% to 94.40%	27	69.23%	53.58% to 81.43%
	Y/P Beacon	41	34	82.93%	68.74% to 91.47%	29	70.73%	55.52% to 82.39%
	Orange Ball	156	139	89.10%	83.24% to 93.08%	111	71.15%	63.60% to 77.69%
	Red AIBO	49	-	-	-	31	61.22%	47.25% to 73.57%
	Blue AIBO	54	-	-	-	29	53.70%	40.61% to 66.31%

Table 8.4: The accuracy of our object recognition system in a set of 500 images: 300 taken from a stationary camera, and 200 contain some degree of blur. The images were taken over a variety of lighting conditions.

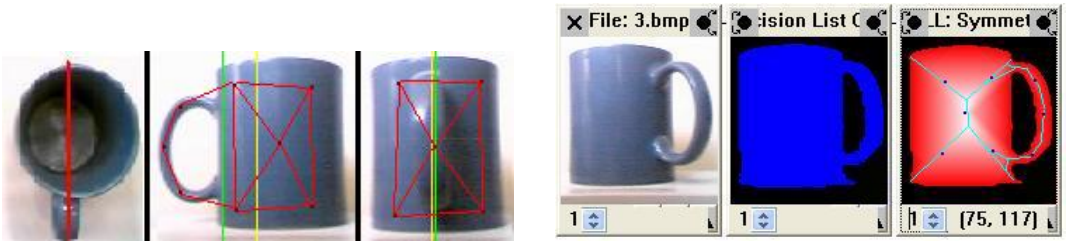
Chapter 9

Versatile Posture Recognition

In Chapter 8 we presented our pipeline for illumination-independent object recognition. This pipeline enabled us to quickly find the description of the boundaries of objects in an image (for complex edge detection). However, what we were to do with those boundaries was left unanswered. In this chapter we combine that pipeline with the fast vectorisation technique that we presented in Chapter 6 to present an efficient posture and gesture recognition system for complex objects such as robots and humans. Our technique is not a general solution to all posture recognition problems, but it is useful across a wide variety of domains and does have some advantages over existing systems. It is a very efficient algorithm that provides a robust posture recognition that is tolerant to changes in the distance to the viewed object.

Our technique will build on work by Chin, Snoeyink and Wang [28] in that we will be using the medial axis of objects to determine their symmetry. However, we employ a slightly different definition of the medial axis. The medial axis is, conceptually, a minimal representation of an object. It is defined by Lee [79] as the boundary of the Voronoi diagram of the edges of a polygon.

While our definition is different, it is still an identical transform. We define the medial axis as the raster representation of that polygon where every pixel inside the polygon is assigned a weight equal to its Manhattan distance to the nearest edge of the polygon. The set of pixels with locally maximal weights will be identical to the medial axis defined by Lee. This set is what we define as the *skeleton*. There is an example of the medial axis in Figure 9.1 (b) shown in shades of red, while the resulting skeleton is shown by the blue lines.



(a) (b)

Figure 9.1: The posture of many objects can be calculated from the symmetry of their skeleton (a). The skeleton (the blue lines) is computed from the medial axis (shown in red) of the blob (shown in blue) (b).

9.1 Object Symmetries

Most real world objects have at least one line of symmetry. Our technique can use one or more lines of symmetry. However, we have found it to be particularly successful in the case when we identify the posture of objects that have exactly one symmetrical axis.

For example, consider a common coffee mug as in Figure 9.1 (a). When viewed from the top the mug has a single line of symmetry passing through the centre of its handle. If the mug is viewed from directly in front (that is, with the handle facing us) or from directly behind (with the handle behind the cup) then the cup will look symmetrical to us along the vertical axis. If, however, we view the cup from the side then the handle will create an asymmetry which we can use to identify the posture of the object.

We use a technique very similar to the computation of the medial axis [28] to identify the structure of an object. We define the skeleton of the object as the set of all locally maximal pixels in the medial axis. This skeleton can then be easily analysed for symmetry using the techniques presented below. A symmetrical skeleton would indicate that the object is being viewed from either in front or behind. The further from symmetry the skeleton is, the larger the rotation of our viewpoint relative to the object is (refer again to Figure 9.1 (a)).

The first step when computing the skeleton is to segment the image to identify coloured regions that belong to our object. Colour segmentation is an extremely common task in visual processing systems so our approach has no extra overhead

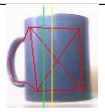
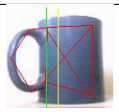
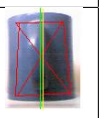
Actual Orientation	0^o	45^o	90^o	120^o	150^o	180^o
						
Perceived Orientation	-1^o	49^o	86^o	118^o	153^o	179^o

Figure 9.2: The actual orientation of the coffee cup is compared to the perceived orientation from our posture technique. Our algorithm is easily accurate enough to be useful for a robotic vision system.

compared to other systems because of this step. We have previously described a method for efficiently doing this first step in Chapter 4.

After the areas of the image (we will refer to them as blobs) that correspond to the object are identified, we can find the medial axis by stripping pixels from the boundary of the blob (or, similarly, by a distance transform) until the medial axis is evident [28] (see Figure 9.1 (b)). We can, however, significantly improve on this technique. We have presented a linear time algorithm for computing the poly-line representation of a given list of pixels in Chapter 6, and in Chapter 8 we presented a linear time algorithm for finding a list of pixels that represents the edge of a blob, without doing edge detection on the entire image. Applying these methods results in an algorithm that is an order of magnitude faster than the original stripping technique because not every pixel needs to be examined.

The final step is to identify the skeleton, S, from the medial axis, A. We care only about the nodes of the skeleton — the lines are drawn on the images in the figures to aid human understanding. To find S, we assign each pixel in the medial axis $p \in A$ a support value V_p based on how deep it is within the bounds of the blob (that is, the lighter areas in Figure 9.1 (b) are assigned higher weights). The local maxima within A are found by comparing each value V_p with the values of the other pixels in A that are close to p. If a unique local maximum is found within a given area then that pixel is added to S. However, sometimes there will be a number of pixels in an area with the same support value. In this case, our algorithm selects one pixel as a representative and adds that to S. By placing the nodes V_p in a heap data structure, the entire process is achieved in linear time relative to the number of pixels in the blob.

9.1.1 Symmetries in the Medial Axis

Using skeletons for the task of object recognition is common [97, 39]. Usually, after the medial axis is found a match is sought between the medial axis and some learned or stored pattern. Our new technique also differs at this point.

The medial axis is analysed for symmetry by comparing its median point to the median point of the entire colour-segmented blob. This is a simple task that can be performed efficiently in one or more dimensions[1]. The median point on the skeleton is calculated from the set of all points in the skeleton while the median point of the blob is calculated from the set of all pixels in the blob. We refer to the median points as M_S= (M_{S_x}, M_{S_y}) or M_B= (M_{B_x}, M_{B_y}). The median of all the x-projections for all points in the skeleton S is denoted as M_{S_x}. Similarly the median of all the y-projections for all points in the skeleton is M_{S_y}. In the same way the medians for the x and y projections of all points in the blob are M_{B_x} and M_{B_y}.

The relationship between M_S and M_B reveals the orientation of the viewed objects. Refer to Figure 9.2. This figure illustrates how the distance between M_{S_x} and M_{B_x} varies with the orientation of an object with one axis of symmetry. M_{S_x} is represented by a green line in these images, while M_{B_x} is the yellow line.

If the distance between M_{S_x} and M_{B_x} is large then the object is at its most non-symmetric orientation relative to the viewpoint. In fact, there is a direct correlation between the perceived angle of orientation (θ) that is proportional to the inverse sine of the distance between M_{S_x} and M_{B_x}. The correlation fits closely the equation

$$\theta = \arcsin(d/max_d). \tag{9.1}$$

The variable max_d represents the distance in pixels between M_S and M_B at 90^o orientation. This calculation can be adjusted appropriately for objects with more than one axis of symmetry. Furthermore, our technique is largely

[1]Strictly speaking the median point in 2 dimensions of the set $Q = \{(x_1, y_1), ...(x_n, y_n)\}$ is the point $P = (x, y) \in Q$ that minimises $f(x) = \sum_{i=1}^{n} \parallel P - Q_i \parallel$ where $\parallel P - Q_i \parallel$ is the Euclidean distance between the points P and Q_i. Finding the spatial median is intractable [7] so this is not the definition we use. Rather we define $\parallel P - Q_i \parallel$ to be the *Manhattan* distance between P and Q_i. In this way the median point may be computed in linear time on the x and y components independently. We use medians rather than means because medians are more robust estimators of central tendency than means [135].

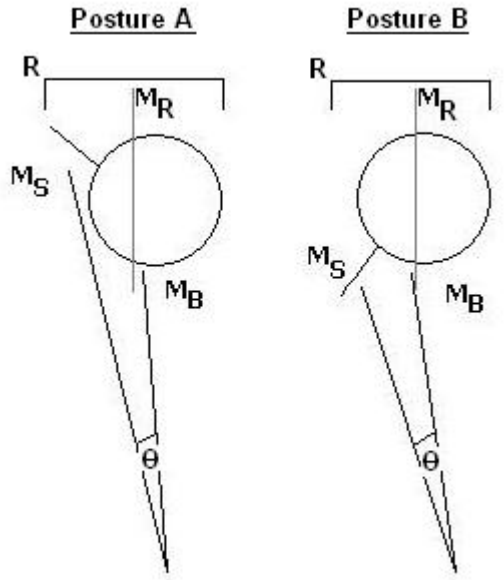


Figure 9.3: For certain objects, more than one orientation may yield the same distance between M_S and M_B. In this case we may compare the position of either M_S or M_B to the bounding rectangle, R to discriminate between possibilities.

independent of the distance to the viewed object. The distance between the two median points, with respect to max_d, may be normalized against the overall size of the blob. This means that the computationally expensive process of image normalization, which is common in posture recognition systems [40], is not required for our technique to work. Normalization can be performed as the final step on several pre-calculated points, instead of the initial step on the entire image.

Equation (9.1) fits well when the viewing angle is in the range $-90^o < \theta < 90^o$. If the actual angle of orientation lies in the range $90^o < \theta \leq 180^o$ or $-90^o > \theta \geq -180^o$ then the solution provided by our equation will be out of phase. Consider Figure 9.3. For each orientation within $90^o < \theta \leq 180^o$ or $-90^o > \theta \geq -180^o$ there is another within $-90^o < \theta < 90^o$.

To discriminate between these possibilities we introduce a new feature — the bounding rectangle, R, of the blob. Let M_R= (M_{R_x}, M_{R_y}) where M_{R_x} is the median of the x-projection of all pixels in R and M_{R_y} is the median of the y-projection of all pixels in R.

When our viewing angle is close to 0^o the object looks completely symmetric

to us. There is a set of orientations that produce a skeleton homomorphic to the one produced at 0^o. All parts of the object viewed from this set we will label the FRONT of the object. The parts of the object that are only visible from outside this set of angles are labelled the SIDE. In our coffee cup example, we are viewing more of the SIDE of the object as the handle moves into view.

When θ is small, the perspective of the viewpoint ensures that the SIDE of the object will have few pixels in it compared to the FRONT. This is because it is further away from the viewpoint. We call any posture visible from the set of angles $-90^o < \theta < 90^o$, Posture A. The ratio of pixels in the SIDE to the FRONT of the object when viewed in Posture A is $Ratio_A$. Each possible Posture A, however, has a mirror posture shown in Figure 9.3. Both of these postures render the same angle θ using Equation (9.1) because $\mid M_{B_x} - M_{S_x} \mid$ is equal. We call the mirror Posture B and the ratio of pixels in the FRONT to the SIDE of the object in this posture $Ratio_B$. The perspective of the camera ensures that $Ratio_B$ will be very large compared to $Ratio_A$ because the SIDE of the object is closer to the viewpoint than the FRONT.

When we examine $Ratio_B$ to $Ratio_A$ we notice several ways that the ratio affects the computation of M_{B_x} and M_{S_x}. M_{B_x} will be influenced by all of the extra pixels now visible in the SIDE of the object and thus will be biased toward that SIDE of the object. M_{S_x} will be relatively unaffected by the extra pixels (as the skeleton uses only the *local* maxima of the medial axis) but the SIDE will look larger and the skeleton obtained will reflect this. Therefore M_{S_x} will also be biased toward the visible SIDE of the object. Notice in Figure 9.3 that, although θ remains constant, M_{B_x} and M_{S_x} move significantly toward the visible SIDE of the object in Posture B and this is reflected in their relationship with M_{R_x}. Thus it is enough to compute $\mid M_{R_x} - M_{B_x} \mid$ or $\mid M_{R_x} - M_{S_x} \mid$ to determine

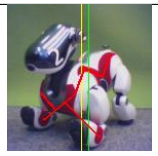
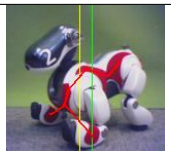
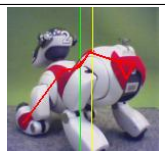
Actual Orientation	0^o	30^o	60^o	90^o
				
Perceived Orientation	-3^o	27^o	65^o	86^o
Actual Orientation	110^o	130^o	150^o	180^o
				
Perceived Orientation	103^o	121^o	145^o	-176^o

Figure 9.4: Our technique can be used to accurately determine the posture of an AIBO. The perceived orientation is, again, very close to the actual orientation of the robot, the largest error being 9^o.

whether our relative orientation lies in either the range $-90^o < \theta < 90^o$ or in either of the ranges $90^o < \theta \leq 180^o$ and $-90^o > \theta \geq -180^o$.

9.2 Application to Robot Soccer

We have been able to successfully detect the orientation of an opposing robot using our technique, regardless of the distance it is from the camera. The AIBO robot, in RoboCup uniform, has coloured patches on its body that are either blue or red (signifying the team). The patches are symmetrical along the long axis of the dog but are not symmetrical along any other axis. (Refer to Figure 9.4.) Once the image has been segmented, the first task is to determine which blobs of the correct colour belong to a particular dog. We do this by proximity clustering [56]. In this way it is possible for us to identify which blobs (representing patches on uniforms) belong to which AIBO.

Once all the blobs belonging to each individual AIBO have been identified, the orientation of the AIBO is identified by symmetry using the technique above. Refer to Figure 9.4. The medial axis of each of the blobs is computed and the skeleton obtained for the AIBO. The median point of this skeleton, M_{S_x}, is compared to the median point of all of the pixels in each of the coloured blobs, M_{B_x}. If the AIBO is in an orientation that exposes only its exact FRONT, then these

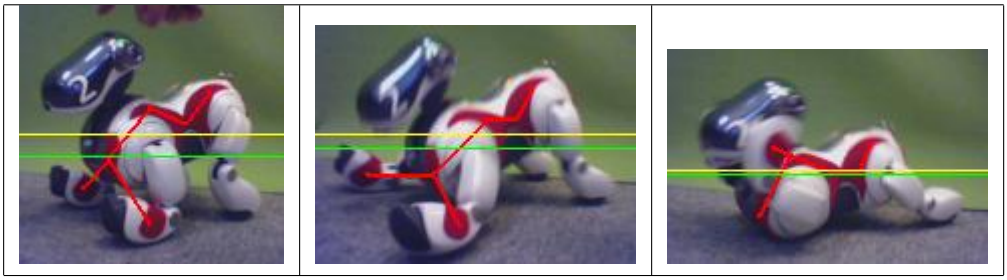


Figure 9.5: Our technique may also be used to detect the kicking motion of an AIBO. We do this by analysing the vertical symmetry, as opposed to the horizontal symmetry for detecting the posture.

two median points will closely match. However, due to the non-symmetrical nature of the other patches on the AIBO, M_{S_x} will be biased further from M_{B_x} and toward the SIDE of the AIBO in view as the perceived orientation angle becomes greater. Using this technique we can determine the orientation of the robot without any shape or posture analysis. Our technique is also independent of the distance to the AIBO because the constant max_d may be normalized with respect to the size of the blob cluster that represents the uniform patches. Furthermore, it is robust to small changes in the posture of the AIBO. In particular we are still able to analyse an opponent while it is walking, introducing only minor inaccuracy.

We have made available on our website[2] several videos of an AIBO using our technique to recognise the posture of another AIBO. In each video the first AIBO must determine the other AIBO's posture in relation to it, and move into a position to mark it. When it reaches its marking position it assumes a guard posture to indicate that it is finished. The videos show that the AIBO maintains the ability to recognise the posture of the opposing AIBO as it moves around the perimeter, and as it moves further and closer to the opponent — only occasionally making the wrong decision. We illustrated our ability in the technical challenge section of the RoboCup 2004 competition in Lisbon, Portugal.

We have used a similar technique to analyse the intent of opposing robots in the soccer game. As an AIBO kicks the ball its vertical symmetry changes. Refer to Figure 9.5. From this sequence of images we see that the M_{S_y} is significantly biased *toward* M_{B_y} as the AIBO flattens its vertical posture. Since most kicks

[2]http://www.griffith.edu.au/mipal/

	Images	Posture Recognised	Accuracy (%)	95% Confidence Interval
Red AIBO	113	94	83.19%	75.23% to 88.96%
Blue AIBO	87	69	79.31%	69.65% to 86.49%

Table 9.1: The accuracy of our posture recognition technique as applied to robotic soccer.

involve this flattening process we have an orientation-independent way of determining if the opposing AIBO is currently kicking the ball. We have also made available a video of the AIBO using this technique to correctly predict kicks from an opponent and intercept them. We have demonstrated our ability both with and without a ball present to show that we are not using the location of the ball, but rather the features of the opponent AIBO, to determine the interception.

9.2.1 Accuracy of Our Method to Robot Soccer

Table 9.1 shows the accuracy of our method as we have applied it to robotic soccer. We captured 200 images containing pictures of AIBOs in realistic RoboCup postures over several sequences of real-time data. In each sequence there is a time-delay of approximately one-tenth of a second between each image. We restricted the scenarios so that all of the images contain exactly one AIBO. There were 113 images of an AIBO in a red uniform taken over two different sequences, and 87 images of an AIBO wearing blue and also taken over two sequences. The images were all taken with a stationary camera, though some of the AIBOs in the images were moving at the time the image was captured[3]. The criteria for successful posture recognition was human inspection of the resulting skeleton overlayed on the original image (as in Figure 9.5).

The main cause of error in our technique is noisy data. The medial-axis varies with noise in the image, but only within limited bounds. Therefore it is a simple matter to judge the feasability of the data retrieved in the current frame by what we have seen in previous images. If the information in the current frame renders a posture that is unlikely based on the data we have recieved over several previous images then we simply ignore it. The frequency with which this occurs, and the effectiveness of our solution can be seen in Table 9.2. Typically, our technique will yield the correct posture in approximately 24 frames out of every

[3]This represents a realistic assumption for posture recognition. Our technique will not work on blurry images that distort the shape of the patches of the AIBO's uniform.

	Images	Posture Recognised	Incorrect Posture not Ignored	Failures (caught %)	Failures (not caught %)
Red AIBO	113	94	7	10.62%	6.19%
Blue AIBO	87	69	7	12.64%	8.05%

Table 9.2: The number of failed attempts at posture recognition because of noisy data. Some of these failures are recognised as such and ignored (caught), others are not. Failures are given as a percentage of the images in which they occur.

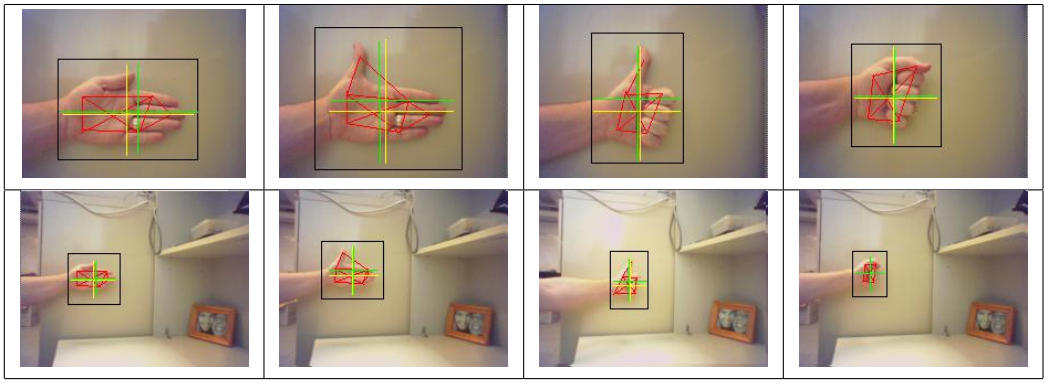


Figure 9.6: By analysing both vertical and horizontal symmetry we may accurately determine the gesture of a human hand. Our AIBO can recognise four distinct hand gestures.

30 (1 second of data). It will further ignore 4 after realising that they must be incorrect. Only two out of every 30 frames will be used incorrectly.

9.3 Gesture Recognition of a Hand

Our technique is applicable in many varied posture recognition tasks. We now illustrate how our method can be applied to differentiating between several common and simple hand gestures. Computing both the horizontal (M_{S_x}, M_{B_x}) and vertical (M_{S_y}, M_{B_y}) median values leads to a robust algorithm that is able to recognise several hand gestures. Refer to Figure 9.6 where we see that it is possible to accurately determine the difference between an open hand and a closed hand with the thumb either up or down. Again, M_{B_x} and M_{B_y} are illustrated by the yellow horizontal and vertical lines, while M_{S_x} and M_{S_y} are illustrated by green lines.

We see that in the images where the thumb is up, the horizontal line of (near) symmetry is broken and so M_{S_x} moves away from M_{B_x}. Similarly when the hand

is extended, the vertical line of (near) symmetry is broken and so M_{S_y} moves away from M_{B_y}. Using this technique we have given the AIBO the capability to recognise and respond to four simple hand gestures. Notice that it does not matter how far away the hand is from the camera, as long as it is close enough that the details can be clearly seen. This is also illustrated in Figure 9.6.

9.4 Maritime Signal Flags

The final illustration of our technique applies to maritime signalling flags. This example shows how repeated application of our technique constructs a very efficient tree to discriminate between several similar objects.

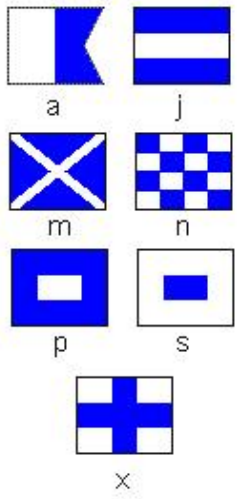
Although ships communicate more frequently with radio and modern communication devices, the maritime signalling flags have not become entirely redundant. Vessels still use combinations of flags to communicate with each other, particularly in crowded areas such as ports and channels. In the internationally agreed upon protocol there is one flag per letter of the alphabet. Signallers may choose to spell out messages, or alternatively, they can communicate many common messages by the presence of only one flag, or the combination of two. We have successfully managed to apply our technique to quickly and accurately recognise each of the alphabetical flags. Firstly, the colours in the flag are identified using image segmentation and the flags are rotated in the image so that the horizontal axis of the flag is parallel with the bottom of the image. It is not necessary to compute the skeleton for such a simple, planar recognition task, so instead of using M_S we compute the medial pixel of the blue blob (M_B) and compare it to the medial point of the entire flag (M_R).

All the flags containing only blue and white are shown in Figure 9.7 (a). These are A, J, M, N, P, S and X[4]. We have chosen to illustrate our technique on this subset because there are several flags in it that have similar symmetries: the symmetries are identical in, say, S and P (a white box in a blue square and a blue box in a white square).

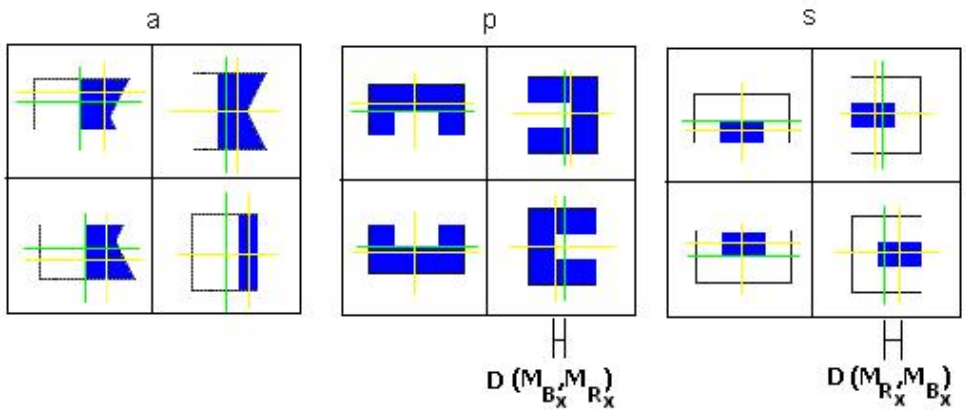
Repeated application in different regions of the bounding box allows us to analyse flags that have identical symmetries. We first take the bottom and top two thirds, then left and right two thirds. Notice that the flags which originally contained the same symmetry information become quite different. For example,

[4]The entire set of flags can be found at http://www.omniglot.com/writing/imsf.htm.

in Figure 9.7 (b) we see that the white square inside the blue box (S) is easily distinguished from the blue square inside the white box (P) because the distance from M_{B_x} (shown in yellow) to M_{R_x} (shown in green) is different in each case. By applying this technique we can quickly and correctly identify every flag in the set.



(a)



(b)

Figure 9.7: Our technique is very versatile. We show here the ability to correctly identify particular flags from a set of maritime signal flags (b) even though those flags have very similar symmetrical properties (a).

Part III

Reliance on Domain-Specific Knowledge in Object Recognition

Chapter 10

Basic Concepts and Related Work

Most object recognition systems use hard-coded, domain-specific knowledge. For example, all leagues in RoboCup rely on the ball being orange and spherical. If it were changed to be a non-uniform colour or a non-uniform shape then most object recognition systems would have to be largely re-coded[1]. We saw how devastating this was in the RoboCup challenge in the four-legged league in 2003 which required the robots to locate a black and white ball. Only eight of the twenty-four teams managed to even identify the ball and no team passed the challenge[2].

10.1 Generic Object Recognition

There is much literature on different methods for representing objects generically in such a way that they can be located within target images. Most of the recent work focuses on trying to *learn* object representations from sample images [99, 78]. As such, the generic representation usually takes the form of a statistical model [64] or some subspace representation of the features of the object [95]. These methods rely on supervised learning techniques and work well when the objects in question appear in a similar pose and at a similar distance

[1]This is certainly true for the team Griffith 2003 code. There are many other examples of systems that are programmed in this way because they are based on the Bruce *et al.* vision pipeline.

[2]http://www.openr.org/robocup/challenge2003/Challenge2003_result.html

in subsequent images. Supervised learning techniques are, as yet, unsuitable for robotic object recognition because of these reasons. Object recognition for mobile, autonomous robots requires a system that is tolerant to changes in the image such as occlusion, viewing distance, angle and posture of the viewed object. While we expect that, in the future, these problems will be addressed by the object recognition community, we focus our work on a non-learned (but still generic) description of objects.

Some of the first work done in this area was by Bergevin *et al.* who attempted to use 2-D line drawings as descriptions of commonly encountered 3-D objects [14]. The idea was that a human would draw a sufficiently detailed set of 2-D line drawings of the object, viewed from a number of perspectives and orientations, and the system would match the skeletons of objects in the images to the set of line drawings. The matching function worked in a similar way to the process we described in Section 7.2.1. While the system was somewhat successful, it proved too computationally expensive. The matching function would need to evaluate each skeleton against every line drawing — multiple perspectives of multiple objects. Also, it took a great deal of expertise in drawing, as well as requiring that the user know how the medial-axis of each object would appear from different angles.

More recent work has focused on decomposition techniques. That is, objects are represented as a composition of smaller parts that can be described more simply [92, 123]. Using this method an object can be described as the relationship between a set of easily recognisable shapes. For example, a person might be described as a circle on top of an oval with certain texture or colour properties, to recognise the head and torso.

The advantage of describing objects in this manner is that we no longer think of objects in terms of *how* they are found in the image, but rather in terms of *what* they look like. In traditional object recognition systems, all of the knowledge about each particular object of interest is coded into a program that describes how to find that object. Even a simple change in an object's properties will require a large re-coding effort under this type of system. Not only is it more intuitive to think about what objects look like, rather than how to locate them, but it is more convenient from a programming point of view as well. By developing an object recognition system that utilises a descriptive object

language we can abstract the domain-specific properties of object recognition away from the object recognition system itself.

The idea of codifying the domain specific knowledge in a goal driven and code-independent manner is not new. The German team in Robocup 2003 [82] presented an XML-based[3] specification language for agent behaviours. Our work presents a similar system applied to the task of visual object recognition.

10.2 Utilising Machine Vision Techniques

Another closely related problem, which our work also addresses, is that of how to combine and apply existing vision processing techniques to the object recognition task [103]. For example, one object may require that edges be extracted and the texture of the internal pixels analysed for identification, while a different object is more easily identified by colour segmentation and connected region analysis. Draper [37] proposes that the task of object recognition is a goal driven task and the user of the system should not need to specify which combination of vision processing techniques are applicable on a per-object basis. Instead Draper proposes that the vision processing tasks themselves can be treated as primitives and their correct combination and application learned by the system for each object, again by supervised learning techniques. While we do not attempt to address the problem of learning combinations of techniques in this thesis, we do address this problem by the static transformation of user-created object descriptions into a vision processing pipeline for each object. Our processing pipeline will be optimal in that it will only apply required vision processing techniques according to the description of each object it is required to identify.

10.3 Our Contribution

As with most vision processing techniques, applying generic object recognition to the field of mobile, autonomous robotics brings its own problems. Robots that operate in unpredictable environments must not only determine where each object is within each image, but if it is present at all. We present an XML based Object Description language (XOD) that is capable of generically describing objects in the manner outlined above. One advantage of our system is that

[3]Refer to the Glossary on XML on Page VI.

object descriptions are easily understood and written by a human. This allows a non-expert user to quickly modify or create descriptions of new objects that the robot may encounter when its environment is changed.

We also present an efficient implementation of our language that is capable of building on the work we have presented in the previous chapters to provide a vision pipeline that is both fast enough to use in a mobile robotic context, and quickly adaptable to changing environments. Our work simplifies the task of describing the objects so that the domain-specific knowledge is removed from the vision-processing module and the details of the vision processing itself do not need to be understood by the user. Objects are described by their appearance rather than by the algorithm that is required to locate them in an image, and this is done in such a way that if the domain context is switched then we can leave the underlying vision system programming unchanged.

We claim that our approach allows rapid expansion to newer or changing vision domains. We will present both the XOD language and the techniques used to translate it to C++ code. We will also present several illustrations where we have used our system to recognise vastly changed objects from those found in RoboCup. We present the XOD language and its implementation in Chapter 11.

Chapter 11

Versatile Object Definitions

In this chapter we present an XML based object description language (XOD). XOD is the first version of a language used to describe the objects we expect our robot to see in a code-independent way. As we described in Chapter 10, such a language allows the domain knowledge required to identify objects in a scene to be abstracted away from the underlying vision system code. Therefore, if the environment of the robot is changed, and new or altered objects are encountered, the vision system does not need to be recoded.

11.1 Primitives

The language works with several types of primitives — *points*, *edges*, *blobs* and *objects* — as well as collections (lists) of these primitives. When we formalise the language as a logic these primitives are the atomic terms of the logic and the relationships between the primitives of our language become the predicates of the logic. We also take advantage of the overlap between object-orientation and knowledge representation in AI to allow our primitives to have properties defined on them.

A *point* is used to represent a pixel. It is not the responsibility of the vision system to convert items from pixels into world coordinates, so a point's location (x, y) in an image is one of its properties. We can use points to specify properties of other objects such as line intersections or centres of objects.

An *edge* is simply a collection of connected points, for example a connected border between two different colours. There are two stages of processing on edges — edge detection and vectorisation. It is sometimes useful to represent edges in raster form and sometimes useful to represent them in vector form so our language allows descriptions for both representations. Our language operates on the properties of edges and the relationships between edges and blobs to find objects. Note that a property of an edge could also be an object in itself — such as the list of points that compose it. When it is required, we use the vectorisation method presented in Chapter 6 for the task of vectorisation of a straight line, and the methods presented in Chapter 8 to vectorise more complex shapes. As these can be expensive operations, they are performed only on demand.

A *blob* represents an area of connected and similarly coloured pixels as well as some other properties of these pixels (such as the bounding rectangle). Our language operates on the properties and relationships between blobs to find objects. Blobs are traditionally formed by tree-based union find operations [16], but we have found this method to be too restrictive for our purposes. Instead we use the methods that we have presented in Chapters 5 and 8 to identify objects. Our system can choose between late or early and between partial or complete edge detection as the context requires. Our blobs are defined by either the edges that compose their boundaries, or a set of points that lie on their boundary.

Thus an *object* is a named entity in our language. It can be a *blob*, an *edge* or a *point* or a set thereof. Objects are made externally available for post-processing by other modules on the robot. We may, for example, have an object called BALL or we may also have an object called FIELD_EDGE which is actually a set of vectorised lines representing the edge of the field.

11.2 Declarative Elements

The task of XOD is to represent objects of interest in a way that allows C++ code to be automatically generated for the task of locating the objects within an image if they are present. We use an XML based language because of its transportability and readability by humans and machines, but the language can

Pink/Yellow Beacon XOD	Formalised Logic
<object> <id>PY_BEACON</id> <above> <touching proximity=1> <proportional error=0.25> <blob><colour>PINK</colour></blob> <blob><colour>YELLOW</colour></blob> </proportional> </touching> </above> </object>	entity(A, B, py_beacon) ⇔ above(A, B) ∧ touching(A, B, 1) ∧ proportional_to(A, B, 0.25) ∧ colour(A, pink) ∧ colour(B, yellow)

Figure 11.1: The simplified XOD for a pink on yellow RoboCup beacon. Notice that XOD is an XML representation of a formalised logic.

be represented as logic clauses for even more readability (see illustrations and figures). The main construct in this language is the tag <object> which encapsulates the definition of an object. All objects have a property of a name indicated by the <id> tag. Many more features or properties are possible, in the style of Object-Orientation [75] or Frames [137].

To illustrate the language, assume now that the object which the vision system will attempt to recognise is to be identified via the use of colour (a common case in RoboCup), and not by lines or edges. Then the object will be encapsulated (circumscribed/bounded) by a *blob* if it is a single colour or encapsulated by two or more overlapping *blobs* of different colours. Our language allows us to search for interesting blobs both by their properties (for example the <colour> tag) and by the relationships between blobs (<above>, <in_front>, <touching>) . For example, the XOD in Figure 11.1 could be used to locate a pink blob touching the top of and proportional to a yellow blob. This definition would be useful if we were looking for the pink on yellow beacon in RoboCup four-legged league.

XOD also allows limitations on the blobs that we use, based on the properties of the blob. For example if we were looking for a large orange blob that has at least 50% of the pixels within the bounding rectangle of the correct colour then we might write something like the XOD in Figure 11.2. This figure also illustrates how the language will permit us to check relationships (or apply predicates) with other, previously defined objects such as the field edge. The truth value of these predicates is determined differently depending on the type of object being used. For example, the truth value of *below(A, FIELD_EDGE)* will need to be

Field Edge XOD	Formalised Logic
<object> <id>FIELD_EDGE</id> <edge> <source>FIELD_BLOB</source> <colour>WHITE,GREEN</colour> <colour>YELLOW,GREEN</colour> <colour>BLUE,GREEN</colour> <vectorise>line</vectorise> </edge> </object>	entity(A, field_edge) ⇔ $a \in$A ∧ bounded_by(a, field_blob) ∧ [between_colours(a,white,green) ∨ between_colours(a,yellow,green) ∨ between_colours(a,blue,green)] ∧ straight_line(a)
<object> <id>CLOSE_BALL<id> <below> <blob> <colour>ORANGE</colour> <area op=gt>5000</area> <pixels op=gt>2500</pixels> </blob> <object>FIELD_EDGE</object> </below> </object>	entity(B, close_ball) ⇔ below(B, field_edge) ∧ colour(B, orange) ∧ greater_than(area(B), 5000) ∧ greater_than(pixels(B), 2500)

Figure 11.2: This XOD illustrates the use of object properties and relationships to restrict the blobs that match an object description. Here we search for a close, orange ball that is on the field.

calculated differently depending whether FIELD_EDGE is an *edge* or a *blob*. In Figure 11.2 it is implemented as an edge vectorised to a straight line to illustrate that objects do not necessarily need to be formed from blobs.

11.3 Imperative Elements

Our language goes beyond declarative statements to simplify using quantifiers. This is illustrated when more than one *blob* in an image matches the criteria in an object definition. In this situation we have the option to specify how to select the correct one or to save the list for post-processing. We do this via the imperative <select> tag. We may choose either to keep the entire set or to select one object according to some criteria. Figure 11.3 illustrates the example where we want to choose the largest orange blob with fewer than 500 pixels.

We can choose to select by any of the attributes of the blob or by comparing to some other named entity. For example, it is possible to select the closest red blob to the object identified already to be the yellow goal. In the four legged league this could be useful in identifying the red goal keeper. In the absence of any select statement the default action is dependent on the type of object. If

Far Orange Ball XOD	Far Orange Ball XOD
<object>	
<id>FAR_BALL</id>	entity(A, far_ball) ⇔
<blob>	
<colour>ORANGE</colour>	colour(A, orange) ∧
<pixels op=lt>500</pixels>	less_than(pixels(A), 500) ∧
</blob>	
<select>	
<area op=gt></area>	$\forall x$ greater_than(area(A), area(x)) ∧
</select>	colour(x, orange) ∧ less_than(pixels(x), 500)
</object>	

Figure 11.3: XOD for an orange ball that is further away.

the object is a blob then the largest blob by area is selected. If the object is a point or an edge then the set is kept and the object remains a collection.

There are other imperative statements in the language used to modify or create objects rather than define or select them. These are the <create>, <copy>, and <set> tags. Full descriptions of these, and all of the statements in XOD can be found in Section 11.6. Before we proceed to the implementation of XOD, we will first examine several illustrative applications.

11.4 XOD Illustrations

11.4.1 Black and White Ball

One of the RoboCup four-legged league challenges in 2003 required teams to locate a black and white ball and then score a goal with it. The challenge was not performed successfully by any team in the competition. The XOD language allows us to specify a definition for the ball that requires only very minor alterations in the ball definition (see Figure 11.4) and the system will identify balls of non-uniform colour easily; in this case, a black and white ball. There is only a minor performance penalty in locating the black and white ball rather than the orange one. This is due to the system needing to calculate a bounding rectangle for the edge-processing operations based on relationships between black and white blobs rather than a single orange one. The edge processing operation will be performed in the same amount of time for either ball provided they appear to be the same size in the image. Compared to a hard-coded system[1] our system performs well. Figure 11.5 shows a comparison of the computation time

[1]As implemented by Griffith University for the RoboCup 2003 competition.

needed to locate various sized orange balls in an image. The extra computational time for the XOD system is largely due to the dynamic memory management associated with processing dynamic sets of objects[2]. The execution times shown in this table include the entire processing pipeline outlined in Chapter 8.

11.4.2 Flag Instead of Beacon

One proposal to move RoboCup more towards real soccer is to replace the navigation beacons with corner flags. The pictures in Figure 11.6 illustrate that our XOD system is capable of easily adapting to this change. Turning the beacons on the side requires only that the <above> relationship in the beacon definition be replaced with a <left_of> relationship.

11.4.3 Identifying an Air-Hockey Puck

We believe that our language and the techniques presented in this chapter are applicable to wider domains than RoboCup. To illustrate this point we describe our use of XOD to detect a cylindrically shaped hockey puck on an air hockey table. The hockey puck is white, as are the paddles which also resemble cylinders. There is white writing on the table and the reflection of the overhead lights is also white which complicates the visual environment. The table itself is blue and the table edge is a green colour that looks fluorescent under the partially black light.

Our rules specify that the puck blob must be touching the blue of the table and under the green table edge. We also specify that it must be at least 1.5 times as wide as it is high, as from the perspective of above the table this will be true no matter where the puck is on the table. Finally we rule out blobs that are too small to be pucks seen from the camera height. Figure 11.7 illustrates AIBO vision workshop detecting a hockey puck in this relatively complex visual environment. We illustrate some of our processing by outlining the detected table edge lines and the edge pixels of the hockey puck.

[2]The Griffith University team policy in the code-base of 2003 was to avoid dynamic memory management altogether for the sake of performance. Most code, including the vision system, ran entirely on statically allocated memory.

Orange Ball XOD	Black and White Ball XOD
<object> <id>BALL_BLOB</id> <blob> <colour>ORANGE</colour> </blob> </object>	<object> <id>BALL_BLOB</id> <contains> <blob> <colour>WHITE</colour> </blob> <proportional error=0.2> <touching proximity=1.5> <blob> <colour>BLACK</colour> </blob> <blob> <colour>BLACK</colour> </blob> </touching> </proportional> </contains> </object>
<object> <id>BALL</id> <edge> <source>BALL_BLOB</source> <colour>GREEN,ORANGE</colour> <colour>WHITE,ORANGE</colour> </edge> <vectorise>circle</vectorise> </object>	<object> <id>BALL</id> <edge> <source>BALL_BLOB</source> <colour>GREEN,WHITE</colour> <colour>GREEN,BLACK</colour> </edge> <vectorise>circle</vectorise> </object>

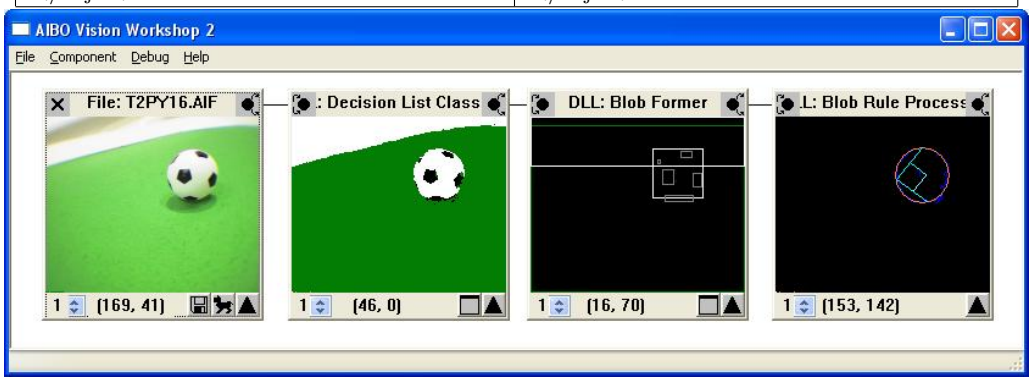


Figure 11.4: A comparison between the XOD for a black and white ball, and the XOD for an orange ball. Notice that no code in the vision system was changed in order to identify a drastically different ball. The alterations to XOD are very simple.

Ball Size	XOD Orange Ball (ms)	XOD Black and White Ball (ms)	Hard Coded Orange Ball (ms)
Small (100 pixels)	8ms	8ms	6ms
Medium (7,000 pixels)	10ms	11ms	7ms
Large (10,000 pixels)	14ms	18ms	9ms

Figure 11.5: The execution of XOD is fast because it is translated into C++ code. The times in this table are measured from Aibo Vision Workshop 2 (see Chapter 12), not the AIBO.

Figure 11.6: The XOD to rotate a beacon by 90 degrees and recognise a flag requires only one tag be modified. The <above> relationship must be changed to <left_of>.

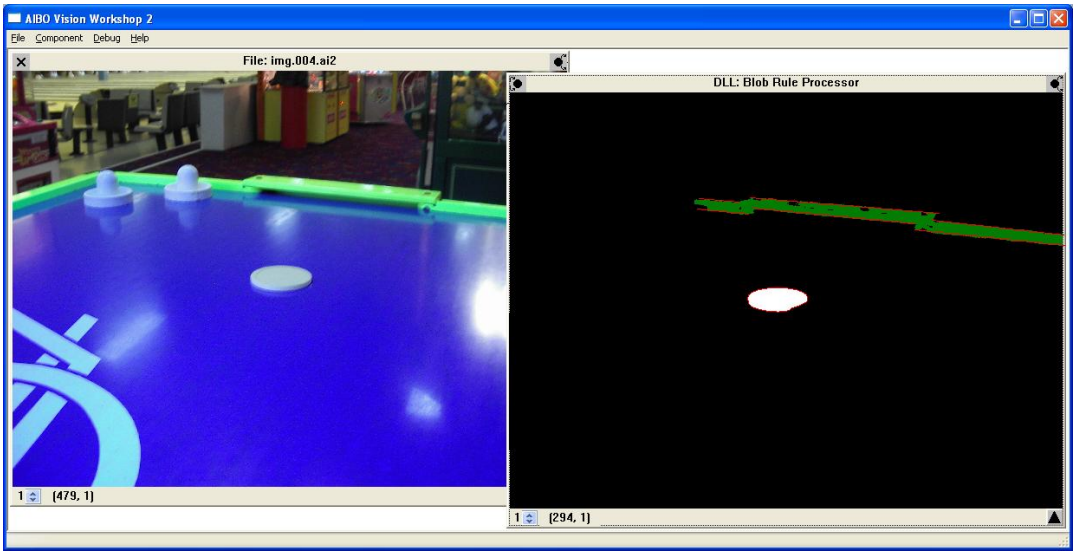


Figure 11.7: XOD is very generic. It is simple to write an XOD for any object that is likely to be encountered. Here we demonstrate our XOD for an air-hockey puck. Notice that XOD is versatile enough to write a description that correctly distinguishes the puck from the similarly shaped paddles.

11.5 Implementation

The first step in our implementation is to perform a sparse classification of the source image (refer to Chapter 4). Pixels of the same colour class are then clustered into groups. From these groups, blobs can be formed as required using the process in Chapter 8, or edges can be found using the process in Chapter 5. The system knows the properties of each object it is required to find, so it is specific as to which blobs will be formed and which edges will be identified. For example, if the system is never required to locate anything that is PINK then it will never form blobs or detect edges from these pixels.

Once the appropriate blobs and edges have been processed, a universal working set is formed of each. Each XOD is then converted to C++ code that performs the following three-step meta-procedure:

1. Build subsets of the universal set according to the properties specified in the <blob> and/or <edge> section(s) of the definition.

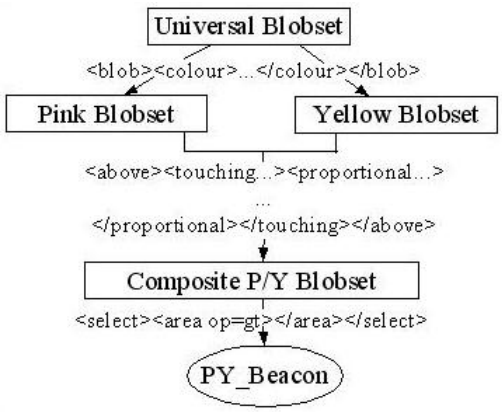


Figure 11.8: A flowchart implementation of the XOD in Figure 11.1. Blob sets are formed by colour and then searched for blobs that fulfill the relationship predicates. Of all the blobs that fill these relationships, one is selected by the properties specified in the <select> tag.

2. Apply predicates forming sets of composite blobs and edges which represent intermediary and candidate objects.

3. Apply the operators according to the <select> statements.

Figure 11.8 illustrates the meta-code generated from the XOD example in Figure 11.1 (searching for a pink on yellow beacon). Firstly subsets containing all the pink and yellow blobs respectively are collated from the universal blob set. If there were other property restrictions defined on these blobs then they would also be checked at this stage. The second step is to compare every blob in list 1 against every blob in list 2 to see if the predicates defined by the relationship tags evaluate to true. If they do, then a new blob is created that encapsulates both of these blobs and it is added to the universal blob set as well as a new set containing those blobs generated in this stage. Properties are generated from the two source blobs and the new blob is given a colour unique to this XOD. Finally the largest blob (by area) is selected from the set of composite blobs and named PY_BEACON. (The XOD in Figure 11.8 has no <select> statement so the default action applies — select largest by area).

11.5.1 Optimisation

For efficiency, our implementation does not actually manipulate sets of objects but rather indices to objects in an array. All objects (including composite objects created by our procedure) are stored in a single, immutable (with the exception of adding new objects) array.

We also perform several other important optimisations. Calculating the relationship between objects can be expensive. Determining the *on_top_of* predicate between two blobs, for example, requires a pixel-by-pixel analysis of the two blobs. We do not wish to force our system to calculate these relationships more than once if they are needed by multiple queries but we also want to avoid calculating relationships between blobs that are never queried. Single images can contain many blobs so it is unfeasible even to keep an exhaustive list of blob relationships which have been evaluated. Our solution is to store calculated relationships in a sparse two dimensional array implemented as a map of maps. An absent query using the two blob indices as keys indicates that a relationship has not been evaluated. Computationally expensive predicates (such as <above>, <touching>) are evaluated and stored only on first use. Some bits are set at this time to indicate that the computationally expensive predicates have not yet been calculated. When they are required, the predicates are calculated individually. The predicates mostly evaluate to true or false so a bit-field is an optimal implementation both in terms of space and time efficiency. Thus we minimise the space required to 16 bits for each pair of objects[3].

One of the most expensive techniques applied to most vision pipelines is that of edge detection. In our system, edges are optimally evaluated only as required. Late edge detection is always used, and the XOD system knows when to use partial late edge detection. When a <blob> tag is encountered, the system will check to see if there is an included <vectorise> before the edges of the blob will be discovered. The properties of the <vectorise> tag determine the choice between partial and complete late edge detection.

The result of edge detection is a set of connected pixels. Many applications[4] require a vectorisation process before this information is useful. In the XOD

[3]This includes the space required to store some added information required to evaluate some of the properties on the relationships. The <touching> relationship, for example, allows a tolerance so we need to store how close the blobs are rather than the fact that they touch.

[4]Though not all. The German team in the four-legged league in 2003 presented a paper on a Monte-Carlo localisation technique that required a set of edge pixels[109].

Black and White Ball XOD

```
<object>
    <id>BALL_BLOB</id>
    <contains>
        <blob>
            <colour>WHITE</colour>
        </blob>
        <proportional error=0.2>
            <touching proximity=1.5>
                <blob>
                    <colour>BLACK</colour>
                </blob>
                <blob>
                    <colour>BLACK</colour>
                </blob>
            </touching>
        </proportional>
    </contains>
</object>
```

```
<object>
    <id>BALL</id>
    <edge>
        <source>BALL_BLOB</source>
        <colour>GREEN,WHITE</colour>
        <colour>GREEN,BLACK</colour>
    </edge>
    <vectorise>circle</vectorise>
</object>
```

Figure 11.9: A recap of the XOD for a black and white ball presented here for reference.

language, the shape of the object is given in the <vectorisation> tag so the task becomes one of determining *if* the required shape is present and represented by a certain point set and, if it is, to parameterise it. We have shown in the previous section how simple shapes can be determined much more quickly in our system than in many current image processing techniques, such as the Hough Transforms, permit. Therefore, we build on the object recognition techniques presented in Part II.

11.5.2 XOD Implementation Example

Here we illustrate the entire system by describing the processing pipeline that would be generated by the XOD in Figure 11.9. This XOD description is of a black and white soccer ball as illustrated above in Section 11.4.1 and Figure 11.4. We repeat it here for reference.

The first step in every pipeline is to perform sparse image segmentation as described in Chapter 4. Pixels are classified and simultaneously clustered according to their colour class. The clusters form the basis of *blobs* in our

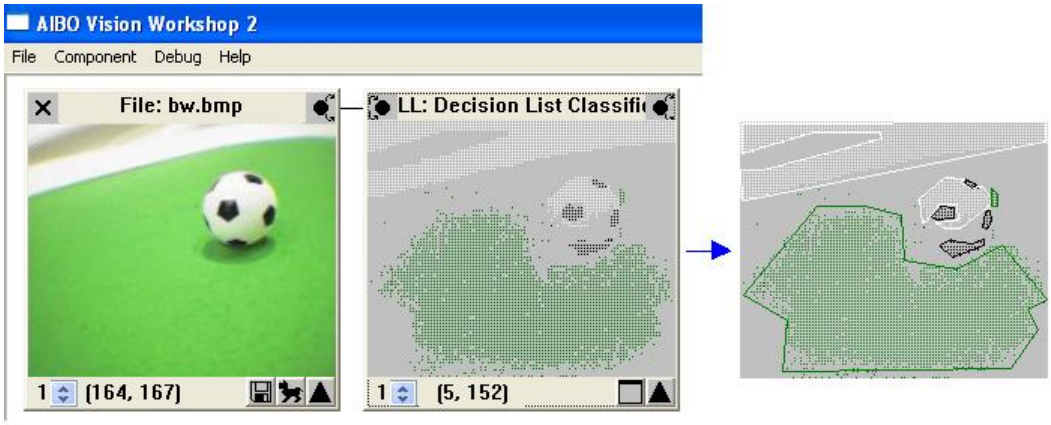


Figure 11.10: Blobs are formed in our system by clusters of pixels. We show an image here of the black and white clusters. The image has been modified so that black clusters are visible.

system. This step is illustrated in Figure 11.10 which shows a modified image so that the colour classes BLACK and WHITE can be easily seen. The lines drawn on the right image are to illustrate the clusters — no boundary detection of clusters is performed at this stage. Each cluster is labelled as a blob and placed in the universal blob set, sorted by colour and then size to optimise searching.

The object recognition system will know what colours are required because it was generated from the XOD. The XOD in Figure 11.9 describes a composite object that is made of two BLACK blobs contained in one WHITE blob (as per the <contains> tag) so the object recognition system will work with blobs of these colours. The innermost tags are <touching> and <proportional> and they require two black blobs so the universal blob set is searched for black blobs that fill the spatial requirements. Each two matching black blobs that are found will be compared against the universal set of white blobs to see if there is a blob that satisfies the <contains> tag. If there is, then a new blob will be created to represent the merge of these three blobs and put in a set labelled CANDIDATE_BALL_BLOBS. The lack of any <select> tag indicates that the largest of these blobs is to be chosen as the BALL_BLOB. The pseudo-code for this process is in Algorithm 11.1.

Algorithm 11.1 Pseudocode for the C++ implementation of the XOD in Figure 11.9.

Input: A universal set of *blobs* B, indexed by colour.

Output: A *blob* labelled BALL_BLOB (inserted into B) representing a black and white ball, if it exists.

```
1:  Let C be the empty set labelled CANDIDATE_BALL_BLOBS.
2:  for all BLACK blobs, b1 in B do
3:    for all BLACK blobs, b2 in B do
4:      if proportional_to(b1, b2, error) ∧ touching(b1, b2, proximity) then
5:        for all WHITE blobs, w in B do
6:          if contains(w, b1) ∧ contains(w, b2) then
7:            Create c representing w, b1 and b2.
8:            C = C ∪ {c}
9:          end if
10:       end for
11:     end if
12:   end for
13: end for
14: if C ≠ ∅ then
15:   Select BALL_BLOB, the largest blob from C.
16:   B = B ∪ {BALL_BLOB}.
17: end if
```

(a) (b) (c)

Figure 11.11: The ball is identified using partial edge detection. Seed points are found (a) and rays are cast to find points on the circumference of the circle (b). We then use the perpendicular bisectors method to parameterise the circle (c).

As the algorithm must compare every black blob to every other, and then each accepted pair to every white blob, it has cubic cost — there is no way of avoiding this. Clever indexing of the universal set, and carefully written criteria on the blobs can significantly optimise the runtime cost.

The final step in Figure 11.9 is to use this blob to detect a circle. The XOD informs us that we are interested in locating edges between white and green pixels, and between black and green pixels but not, for example, between black and white pixels. After the edges have been located we vectorise them as a circle.

The system knows that circles can be easily vectorised using partial edge detection and so edges are found using the method presented in Section 5.2.1 and examined against pixels in the original image to determine if they represent a boundary between the correct colours. Note that we are not required to specify colours in an <edge> tag. If they are not present then any edge will be permitted. Once sample points are found that we know lie on the edge of the circle, the circle is vectorised using the perpendicular bisectors method. Repeated application of perpendicular bisectors among different sample points allows us to determine whether we have actually found a circle with the correct characteristics. If we have not, the shape is discarded as noise. Figure 11.11 illustrates how samples are selected for partial late edge detection and the resulting circular vectorisation is obtained.

More complicated objects can be identified using the complete edge detection method presented in Chapter 5. For example, the uniform of an AIBO can not

AIBO XOD

```
<object>
    <id>AIBO</id>
    <source>AIBO_COMPOSITE_BLOB</source>
    <edge></edge>
    <vectorise>aibo_posture</vectorise>
</object>
```

Figure 11.12: We may use XOD to describe the posture of an AIBO. The system knows that it must apply the algorithms from Chapter 9 to obtain this information.

be recognised using partial edge detection. However, the XOD knows which code module to call for each type of vectorisation. The posture vector of an AIBO described in Chapter 9 can be expressed very simply in XOD as shown in Figure 11.12 (assuming that AIBO_COMPOSITE_BLOB is a composite blob representing a correctly identified cluster of AIBO uniform patches).

11.5.3 Assumed Objects

Assumed objects are those that are not described by XOD but are nevertheless available to it (and therefore assumed) in every system. We have only one such assumed object in our system, that being the horizon in the image. Not only can XOD use this object to test if blobs are above or below the horizon in the image (for example), but the XOD system itself uses this entity when determining some relationships. For example, the relationship *above* is not determined relative to the top of the image, but relative to the horizon. The horizon is represented as a straight-line vectorised edge with one endpoint on one side of the image, and the other on the other side. Therefore there must be code that runs on the robot that calculates these endpoints through whatever means are available on the particular robot. On the AIBO we use the same system as presented by the German team in their 2003 team report [108] — a combination of accelerometer readings and joint positions.

11.6 XOD Language Specification

A complete specification of the XOD language will appear in a publication subsequent to this thesis. We list here, for reference, the most commonly used tags and options. In this table, an *entity* refers either to a *blob*, to an *object* or to an

General Tags	**Contained by**	**Description**
blob	object/relationship	Specifies a cluster of pixels
edge	object/relationship	Specifies a list of pixels on border of some blob
object		Declares a named XOD entity
Relationship Tags	**Contained by**	**Description**
above	entity/relationship	Specifies that the first entity is above the second
behind	entity/relationship	Specifies that the first entity is behind (according to Z-order) the second
below	entity/relationship	Specifies that the first entity is below the second
contains	entity/relationship	Specifies that the first entity contains the second
in_front	entity/relationship	Specifies that the first entity is in front (according to Z-order) of the second
left_of	entity/relationship	Specifies that the first entity is left of the second
not	entity/relationship	Negates the contained relationship
proportional	entity/relationship	Specifies that the first entity is proportional in size to the second
right_of	entity/relationship	Specifies that the first entity is right of the second
touching	entity/relationship	Specifies that the first entity touches the second
Imperative Tags	**Contained by**	**Description**
select	entity	In the case of multiple entities matching a single XOD, describes which one to select
vectorise	edge	Describes how to vectorise an edge (e.g., as a line, circle, etc.)
Other Tags	**Contained by**	**Description**
area	blob	Restricts the blobs that match this XOD by area
	select	Specifies conditions on selection by area
bounding_rect	blob	Restricts the blobs that match this XOD by bounding box
	select	Specifies conditions on selection by the bounding box
colour	edge	Restricts the edges that match this XOD by colour
	blob	Restricts the blobs that match this XOD by colour
	select	Specifies conditions on selection by colour
id	object	Uniquely names an object or a set of objects
pixels	blob	Restricts the blobs that match this XOD by the number of pixels
	select	Specifies conditions on selection by the number of pixels
shape	select	Specifies conditions on selection by shape when select is working on a set of vectorised edges
source	edge	Specifies the source entity (blob/object) to edge detect
Assumed Objects	Object Type	**Description**
HORIZON	edge	A vectorised line representing the horizon in the image

Table 11.1: An XOD reference. The complete specification is forthcoming.

edge.

Part IV

Development and Debugging Facilities

Chapter 12

AIBO Vision Workshop 2

12.1 Basic Concepts and Related Work

In embedded systems, such as autonomous robots, development of vision processing tasks is severely hindered, not only by the limitation on the available processing power inherent in such a system but also by the usually meager facilities available for testing, debugging, quality evaluation and profiling. The task of development for an embedded device is often very challenging[5, 89, 120] and several attempts have already been made to provide generic support tools for the process (for example Thrun's embedded development system[124]). However, we add to these general difficulties the requirements for developing a real-time vision processing system, which compounds the problem significantly. For example, it is often very difficult to use generic programming techniques, such as in Thrun's work, when programming vision systems, because of the vastly different image formats delivered by different hardware[1], the sheer quantity of data and the limitations in processing power already mentioned. Such limitations place restrictions on the usual mechanisms for the dynamic binding required for generic programming. To process an image of size 176 by 144 pixels of 3 bytes per pixel requires over 25 thousand virtual function calls if the class that represents a colour, or a pixel, is generically programmed. If there were, say, 25 image frames per second we would be performing 625 thousand unnecessary virtual function translations per second on a low-powered device. Add to all of this the requirement of real-time response to visual stimuli (featured in many robotics based

[1]Examples of formats include RGB, YUV and HSI. For more information refer to colour spaces in the Glossary on Page 181.

tasks), as well as the necessity for rapid development to respond to changes in the visual environment, and development becomes impossible without the simultaneous development of utilities that assist in the process.

There are many examples of tools, some of them quite complex, written for the sole purpose of aiding the development of some other piece of software. In a broad sense, every IDE[2] and debugging tool fits into this category. Specific tools such as our first example, a simulation environment to simplify the development of robot navigation algorithms [87], are written frequently to aid in development and testing of complex problems. This particular tool was developed to simulate large environmental terrains for the purpose of comparing different navigation strategies and allows the user to "plug in" different navigation algorithms. There are many other examples of such tools: a virtual reality (VR) toolkit for distributed development of VR user interfaces [114], an IDE for scientific computing and data visualisation [6] and a tool for simulation and analysis of scheduling algorithms [112] to name but a few. All of these tools are similar in that they aid in developing solutions for a particular problem domain in a generic manner. Of course, not all support tools take the form of complete applications in themselves. Statically linked code libraries that support development features are very common such as *Ptolemy* [17], which is a simulation library for heterogeneous and embedded systems. Tools may even take the form of raw source code that is compiled with the target code to perform common tasks such as reliable estimation of execution time on embedded devices [48].

We see from these examples that if a particular type of problem becomes a frequent task then companies and individuals alike will go to great lengths to produce reliable, flexible and reusable toolkits to aid in development of solutions for it. Some of the toolkits cited above classify as major software engineering efforts in their own right. A toolkit may still be developed for sufficiently complex problems that are not very common, but it is likely that the toolkit will be highly domain-specific and largely not reusable. For this reason it is unlikely to be widely distributed.

[2]Refer to the Glossary on IDEs on Page 183.

12.2 AIBO Vision Workshop 2

In this chapter we present our tool for vision system development — AIBO Vision Workshop 2 (AVW2). The inspiration for AVW2 comes from a program called Filter-Graph distributed with the (free) Microsoft Direct-Show SDK[3]. To illustrate the concept behind AVW2, let us consider briefly the philosophy behind Direct-Show. The Direct-Show architecture divides the task of displaying a movie into a pipeline architecture as defined by Shaw *et al.* [115]. For example, a file loader component may open the file containing the data and then pass it to a normalising filter for sound which passes the data to a renderer to display it on screen. If the incoming data arrives from a network rather than a file then the only change that needs to be made is to replace the file reading component with a network retrieval component. Similarly, if we want to save the movie to a file rather than display it then we simply replace the renderer component with a file writer. By allowing a "plug in" architecture with a standard set of data formats it is possible to obtain an extremely flexible environment for movie processing. All input components collect the data, convert it to the internal format and pass it on to "connected" processing components. Finally one or more output components receives processed data and renders it in some way. If a user wished to process a home-made movie and write their name in the corner of every frame, for example, then this is almost a trivial task by simply writing a processing component that works with the common data formats.

AVW2 works in a similar manner. Images are obtained by one of the input filters and converted from whatever format they were in to AVW2's standard image format. The standard image format chosen is commonly used by many hardware camera devices (YUV2). The images are then passed to one or more processing components which are called *filters*. Each filter, in turn, receives the cumulated incoming data, performs some processing task and then passes the original data as well as any new or modified data to any connected filters. Along the way each filter displays any debugging information either in a visual manner (on the screen) or, if more appropriate, to AVW2's text output console.

Figure 12.1 illustrates this process. Image (a) is a screenshot of AVW2 with a file-loading input filter and three custom-built processing filters while Image (b) shows how data are passed between the different filters internally. The load file

[3]The SDK can be accessed at http://msdn.microsoft.com.

filter passes the AVW2 format converted source data to the first processing filter which, in this case, is a colour classifier performing the common task of image segmentation. The colour classifier converts the data to a colour segmented image (which it displays for debugging purposes) and passes both the source image and the colour segmented data to the next filter in the chain. In this case the next filter is a bounding box detection algorithm (or blob former) which locates bounding boxes of objects within the image. Finally all of these data are passed to a circle detection filter which identifies the ball. These filters correspond to the processing steps that we examined in the XOD description of black and white balls in Chapter 11.

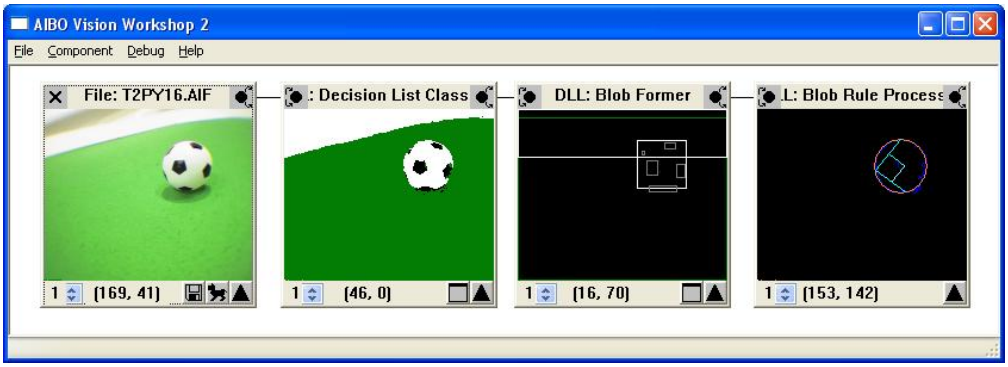
12.2.1 AVW2 Customisation

In order to allow users to create their own processing components AVW2 must do two things. Firstly, the filters must be user-written as the processing that each filter does is specific to the application. Secondly, AVW2 must be able to properly handle user-defined data types. For example, a user should be able to create a filter that processes an image and produces some intermediate data of form A. The data must be passed up the filter chain even though AVW2 knows nothing of its structure. Furthermore, if a filter that requires data of type A is added to the filter chain then it is necessary for AVW2 to check that some prior filter in the chain can supply data of that type. AVW2 does this by allowing filters to access data by a string key. Each component in the chain assigns a key to each of its outputs and filters further up the chain can request this data by its key. The keys are typed so AVW2 can check that each filter's data requirements are met when the filter is added to the chain (see Figure 12.2).

Filters may be written by users in one of two ways: as a dynamically-linked library (DLL)[4] or as a script interpreted by AVW2 at runtime. Figure 12.3 explains the filter architecture. Users create general purpose image processing filters by following a standard DLL format which acts as a class factory for a class inherited from a generic DLL filter class. Alternatively, scripted image processing behaviour (in C++) may be used in place of compiled filters. In this case the ScriptFilter simply reads a C++ source file and executes it rather than the user creating an entire compiled filter. In either of these ways then, code can

[4]Refer to the Glossary on DLLs on Page 182.

(a)



(b)

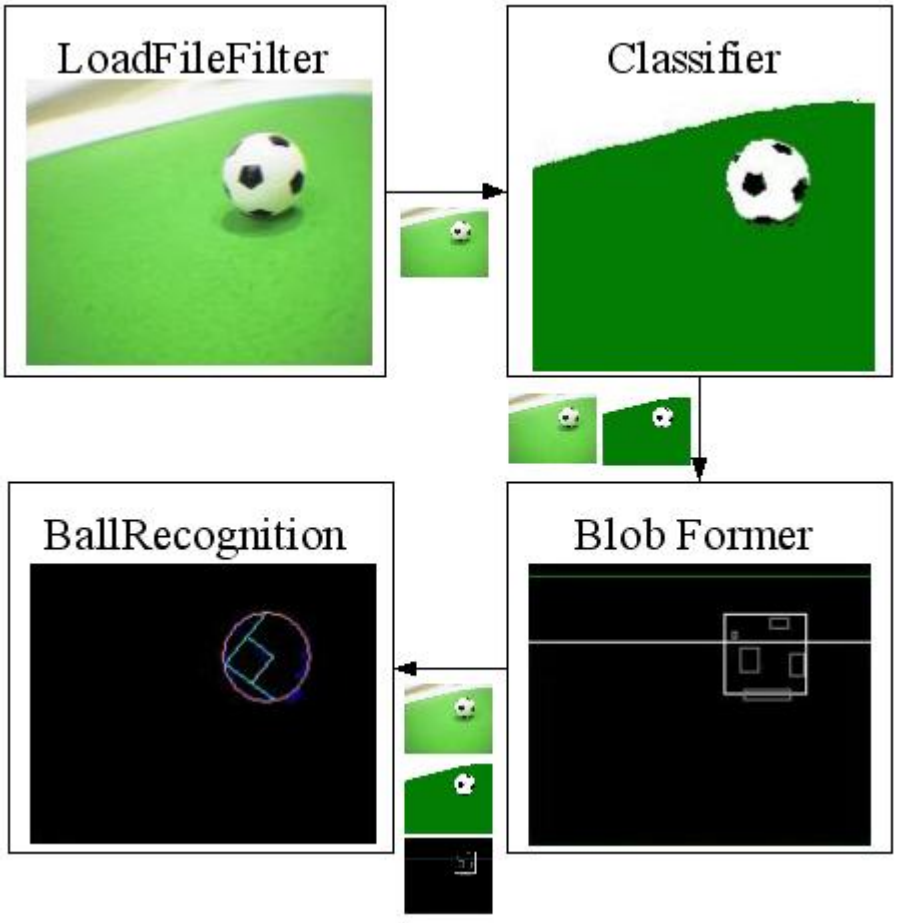


Figure 12.1: AVW2 works similarly to a pipeline architecture. An input filter loads the image and passes the data through a series of other filters until the object recognition task is complete.

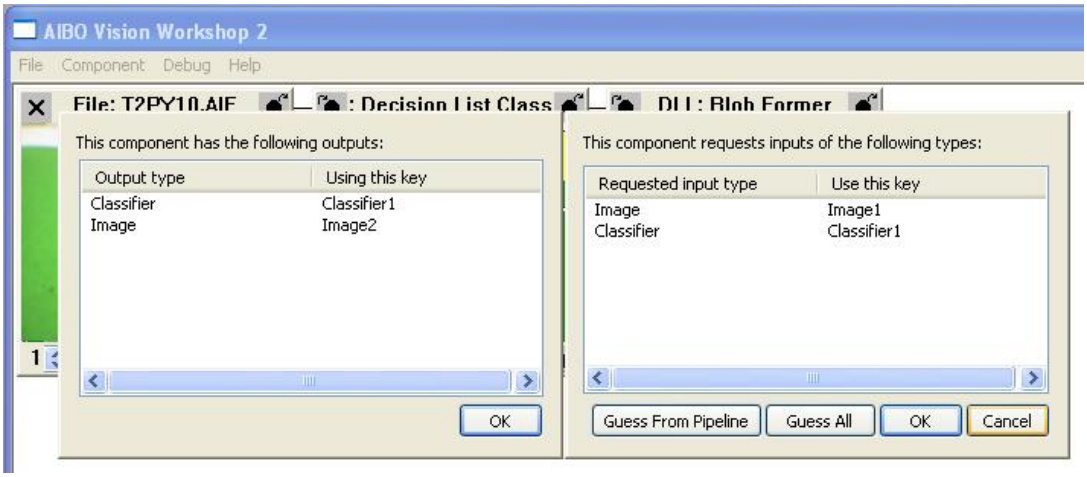


Figure 12.2: AVW2 is a generic tool for vision system development. It supports generic data types and type cheking through a system of keys and values.

be moved (ported) directly from the embedded device and either compiled into an AVW2 DLL or used in an AVW2 script.

By doing this we gain several important advantages. Firstly, code can be debugged in a much more sophisticated debugging environment than is likely to be present on the embedded device. Secondly, debugging information can either be drawn superimposed on the source image or replace the source image entirely. This allows for visual representation of the debugging data which is critical for developing image processing algorithms. If the entire functionality of the target code is divided into several filters then debugging information can be displayed for each step. Thirdly, the image processing pipeline itself is constructed at runtime and filters are loaded and connected dynamically. This makes it easy to see the effect of using different algorithms to perform the same task. For example, a nearest neighbour colour classification algorithm can be replaced by a decision tree colour classification algorithm with only two mouse clicks. The performance of each (both in runtime efficiency and quality of result) in the context of the entire processing pipeline can then be easily evaluated.

Of course, in order to develop a filter for AVW2 one must support AVW2's native image format for the source image data. As already mentioned, we often wish to avoid the overhead of generic programming on a real-time embedded device[5]. We do not, therefore, wish to impose AVW2 support code into the code

[5]For development on an AIBO this is not an issue because the internal data format is

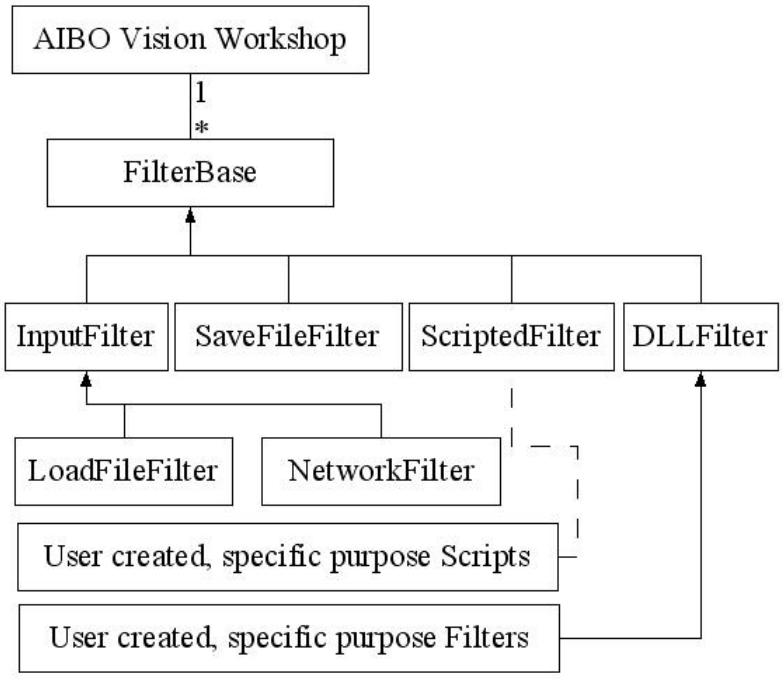


Figure 12.3: The filter architecture of AVW2. Filters provide the generality and extensibility required to make AVW2 a generic tool.

base of the embedded device. The way then to overcome this problem is to write an AVW2 specific wrapper for each filter that can convert the data from AVW2's format into the native format of the device. The wrapper is only used when the code is run from within AVW2 and is discarded when run on the embedded device (see Figure 12.4). Therefore the rigid real-time performance requirement is relaxed for the wrapper ccode.

Another important requirement for customisation of AVW2 is that users can configure the data-flow *between* their filter components as easily at runtime from within AVW2 as they can at compile time on the embedded system. AVW (the original, not AVW2) had a strictly pipeline data-flow architecture [115]. That is, each filter output could connect to one, and only one, filter input. As well as this, only the data output from the filter directly preceding could be accessed by any particular filter. While this provided users with a simple mental-model, it severely restricted what could be done. If some component needed, for example,

the same (AVW was originally designed to support development on an AIBO), but this will obviously not be the general case.

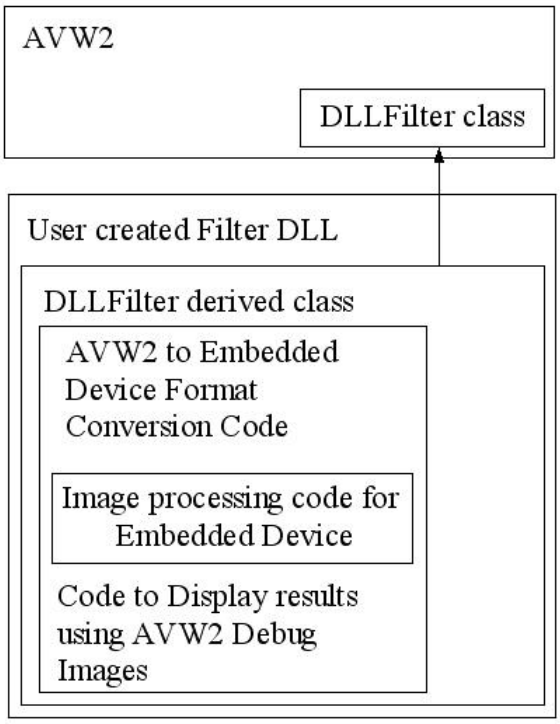
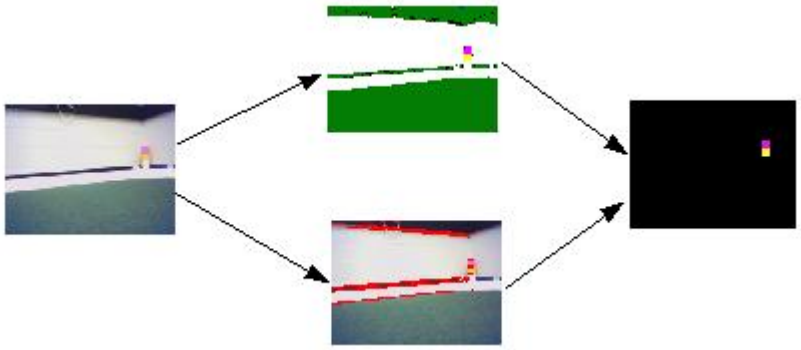
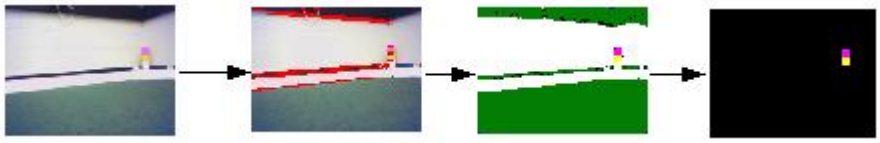


Figure 12.4: Each filter must wrap the code intended for the embedded device by marshalling the data required.



(a)



(b)

Figure 12.5: Several small variations to the pipeline architecture make AVW2 much more powerful. For example, every piece of data from any filter is available as input to any other filter further up the pipeline. This alleviates the problem of requiring multiple inputs on filters (a). Instead the same thing can be achieved more simply (b).

to use data from two separate sources with the purpose of providing data fusion, then there was no way this could occur.

The new architecture of AVW2 is much more flexible while at the same time preserving some of the intuition behind the pipeline idea. The first difference is that *all* of the data from every ancestor in the chain is preserved and accessible through the key mechanism described above. This completely solves the problem of requiring multiple inputs on filters. Figure 12.5 illustrates why. The data-flow in (a) requires that the final component, a beacon detection routine, uses both the data from the colour segmentation routine (top) and that from the edge detection routine (bottom). If every prior piece of data in the filter chain is available to each filter then these two filters do not need to run concurrently and a sequential pipeline will suffice (b). Filters are, however, allowed multiple outputs and this essentially creates a branch in the processing chain. This is most useful when

we want to compare two different filters that perform the same function on the same source data. The screenshot in figure 12.6 shows this feature being used to compare two different image segmentations. The final data-flow architecture of AVW2 is similar in many ways to the oscilloscope architecture defined by Shaw *et al.* [115]. It definitely resembles a pipeline architecture but many differences emerge because of the flexibility needed to allow the filters to interact in the same way they would on the embedded device. Thus filters may be connected and interact in a more complex manner than a pipeline would strictly allow.

12.3 Features of AVW2

12.3.1 AVW2 as a Testing, Debugging and Validation Tool

One of the most powerful means of testing and debugging vision-processing code is to use immersive techniques to display the data as it is actually seen by the device, in a frame-by-frame manner, and watch the results of the code as it is executed. AVW2 permits each filter to display the results of its computation in a visual representation (refer to the screenshot in Figure 12.1). If the incoming images are streamed over a network connection directly from the device then not only is it easy to see the result of the entire vision pipeline, but if some part of the pipeline is malfunctioning then it is far easier to identify the faulty component. Images can be saved from an incoming stream in real time and played back at a later date for a more comprehensive, step-by-step debugging. In fact, it is possible even to run part of the processing chain in the embedded device, send the results over an Ethernet (often wireless) back to AVW2 and pick up the rest of the processing chain remotely. All that is required is a custom input component that receives results from the LAN rather than images. This functionality adds much flexibility to AVW2, especially when it is impossible to stream images from the device over a LAN at full speed (as is the case with AIBOs). For example, we often embed the image classification component on the AIBO and send segmented data back to the rest of the image-processing components in AVW2. The segmented data is much smaller than a raw image so we are able to receive many more frames per second this way and thus overcome some of the limitations of the wireless network itself.

Figure 12.6: Permitting multiple outputs from filters increases the flexibility of AVW2 for purposes of comparison. In this image we are comparing the results of two different colour calibration classifications.

The image processing code resides in a separate compilation unit from AVW2 (in a DLL) so it is quite easy to attach a debugger such as gdb or one that comes with an integrated development environment to the DLL to obtain all the features of a modern debug environment. Stack traces, line-by-line and breakpoint debugging, variable inspection and alteration, register and disassembly information and memory leak detection utilities can all be utilised on the code which will eventually run on the embedded device. Most embedded devices support very few, if any, of these debugging facilities, so code development will be faster and testing and debugging will be easier if AVW2 is used. Logic errors will also

be simpler to detect due to the immersive environment provided at each step of the filter chain.

Although visual immersion is usually the tool of choice for debugging and code validation, sometimes visual environments lack basic text-based I/O. While it is extremely difficult to debug vision systems using only text-based I/O, it is a very valuable supplementary tool. AVW2 provides this facility by a UI component that displays and logs any text-based I/O from each filter as it executes. To do this it provides a set of platform-independent macros for passing messages such as warnings from either a C++ script or an AVW Filter DLL to AVW2. The code to do this does not need to be changed when it is moved from either a script to a filter DLL or from an AVW component to the embedded device. Text output macros are interpreted differently in a script to a DLL and are simply ignored (or printed to standard output) in code on the target platform. Future versions of AVW will redirect standard input and output to AVW2's console, allowing stream based I/O.

12.3.2 AVW2 as a Rapid Development Tool

One of the features of AVW2 is that C++ code can be interpreted rather than compiled. AVW2 is a fully-featured C++ scripting environment supported by Agilent Technologies' embedded C/C++ interpreter CINT[6]. Although scripted code runs at considerable performance penalty, it is very useful when a fast development cycle is necessary or when minor alterations to existing code must be made quickly.

AVW2 exposes an interface to CINT which the script can then use. This interface allows the script to set the parameters of the filter (such as the type of data the filter requires and produces) as well as to obtain the source data, set the result data and draw the debug image for the filter. A script then runs in exactly the same way as a compiled DLL — the main script file will simply include the target code for the embedded system, marshalling the data as necessary. We have included in Appendix B a script that illustrates how easy it is to produce a simple filter for AVW2. This simple script uses a class that is intended to run directly on the target device to complete the task of image segmentation by colour. The other support code simply informs AVW2 of the current script's

[6]http://root.cern.ch/root/Cint.html. This software is free to distribute.

data requirements. Incoming data is retrieved using the key mechanism discussed previously and outgoing data is passed back to AVW2 the same way.

After the script is complete and changes are unlikely to occur it is possible to have CINT and AVW2 compile it using *#pragma compile* at the top of the source code[7]. This produces a DLL that can be used instead of the script. However, this is not equivalent to writing a filter DLL (described above) as it must still communicate with AVW2 through CINT and the exposed scripting interface. It is still a useful compile, though, as it does significantly increase the speed of execution.

These and many other advantages justify the use of AVW2 as a rapid development tool for embedded vision processing systems. The CINT compiler exposes a fully featured, command line interface, debug environment on scripts. It is possible to execute a script line by line, inspect and alter variables and trace execution through the code, all from within AVW2. This requires no third party utilities at all making AVW2 the only necessary tool to develop vision-processing code on a lightweight system.

12.3.3 AVW2 as a Profiler and Performance Monitor

Another feature of AVW2 is that it exposes a simple interface for profiling and performance monitoring of code. AVW2 profiles every filter on every execution without user intervention, but this is often not an accurate reflection of the time actually spent by the vision-processing code. As mentioned above (and illustrated in Figure 12.4) there is a small overhead in DLL filters and a somewhat larger overhead in scripted pieces of code where the code must marshal parameters from AVW2 formats to the format expected by the target device. While this overhead is acceptable in AVW2 execution, it is undesirable in profile statistics in which we would like to reflect only the execution time of our code targeted to the embedded device. For this reason it is possible through both the exposed script interface and the DLL filter base class to tell AVW2 when to start the profiler and when to stop it. This is useful to compare execution times of different algorithms before making an implementation decision. As mentioned above, AVW2 exposes its own console-like command interface which is used to display and log profiling code.

[7]This requires a third party command line compiler (such as g++) set up correctly on the system.

12.4 Discussion

The development of embedded vision-system code is usually a very difficult task, but is nevertheless becoming increasingly important. Autonomous robots, in particular, often use vision as their primary sensory input and more and more robots of such nature are being developed. Still, however, the tools available are usually highly specific tools for particular tasks developed by individuals or groups and not redistributed. Using this kind of development model, many hours of effort are required simply to develop the tools needed before actual vision system development can take place. With the increasing emergence of embedded devices that require vision-processing technology, we are seeing many such customised and extremely specific tools developed. We can provide many such examples of these kinds of toolkits that are useful for RoboCup, in particular, but are not really useful in broader applications. The vision toolkit by Carnegie Mellon University for RoboCup [16] is an excellent example of a class library that would be very useful in creating a filter for AVW2 but does not attempt to provide generic support for vision solutions on embedded devices.

AVW2 is a tool that significantly reduces the effort spent on support tools for vision systems programming, by providing a generic tool that aids in most, if not all, of the common tasks associated with such problems. The user must only write the application-specific code and a small wrapper to marshal data between that code and AVW2. After this AVW2 improves debugging, testing and validation systems, provides an immersive development environment and allows for code profiling and runtime evaluation. AVW2's flexibility and support for fast software development cycles lies both in its ability to accept scripted, rather than compiled, source code and to allow any fraction of the vision-processing pipeline to run on the embedded device, communicating with the rest of the pipeline within AVW2 over LAN. AVW2's power as a general tool lies in its ability to accurately reflect the vision-processing architecture on the target device by its filter components coupled with the keyed data-types. We have ourselves used and developed AVW2 as a support tool for embedded vision system development on our own code for the RoboCup competition over the previous three years.

Part V

Putting it all Together

Chapter 13

Conclusions and Future Work

We have observed that there are several critical areas when considering machine vision as the primary sensory input for mobile autonomous robotics. Firstly, much of the literature in machine vision focuses on still image processing in controlled environments and therefore many of the algorithms and techniques that have been developed are robust and useful, but computationally expensive. They also have poor adaptability to unpredictable conditions. We therefore addressed these issues in Parts I and II of this thesis by introducing several improvements to key low-level vision algorithms. We argued that by improving the efficiency and adaptability of the low-level algorithms we would obtain a remarkable improvement in many high-level vision systems that are composed of them.

We also noted that most vision systems are developed for use in a specific task and visual environment and are therefore quite difficult to adapt to changing operational contexts. We addressed this issue in Part III of this thesis by introducing our descriptive language for object recognition, XOD.

Finally we have observed the practical difficulties inherent in building and maintaining vision systems in embedded devices (such as robots) due to the nature both of vision system programming, and embedded devices. We have noted the lack of good support tools that are available and so, in Part IV of this thesis, we have proposed our own tool, AVW2, for the purpose of real-time, embedded vision system development.

Although improvements in any one of these areas are valuable, vision will only become useful as the primary sensory input for mobile robotics when all four areas are developed concurrently. For example, what use is it having fast

low-level algorithms that cannot be used on the robot because they cannot adapt to dynamic lighting conditions? Or what use are the most robust and adaptable algorithms when they operate too slowly for use in a real-time processing environment? How will vision system development be maintained for a wide variety of domain contexts if the entire system must be re-coded from scratch every time a new object must be recognised? Or what use are any algorithms at all if they cannot be quickly developed, debugged, tested and maintained by the programmers working on them? In the context of robotic vision, each of these areas relies heavily on the others.

In this thesis we have presented significant contributions to each of these areas in order to develop a robotic vision system for the AIBO that is fast, robust, tolerant to changing conditions and easily adaptable. There are many areas that still need to be addressed and many techniques that we have not examined. We will discuss these in the next section. Here we describe the final vision system pipeline that is a result of this thesis. We consider this pipeline a substantial improvement in nearly every way over the basic pipelines that we examined in Section 1.2.

13.1 Our Advanced Vision Pipeline

There are two basic inputs that the vision system developer must provide to our vision system. These are the sparse colour classification (Chapter 4) and an XOD for each object that the system is required to recognise (Chapter 11). The system does not interpret the XOD at runtime[1] so the XOD is translated into automatically generated C++ and compiled into the runtime of the robot. This means that the vision system developer is (usually) not required to write any code. There are some exceptions to this that we will come to. Figure 13.1 describes our pipeline.

Once the XOD has been provided our pipeline can be implemented. The basic means of implementation of our vision system pipeline was described in Section 11.5.2. The colour class of pixels is first determined by the sparse colour classification and each group of similarly coloured pixels is clustered forming *blobs*. The XOD describes both how the blobs must relate to each other and

[1]Though it is certainly possible to interpret XOD at runtime, and indeed we have an AVW2 component that does this, pre-compiled code is faster.

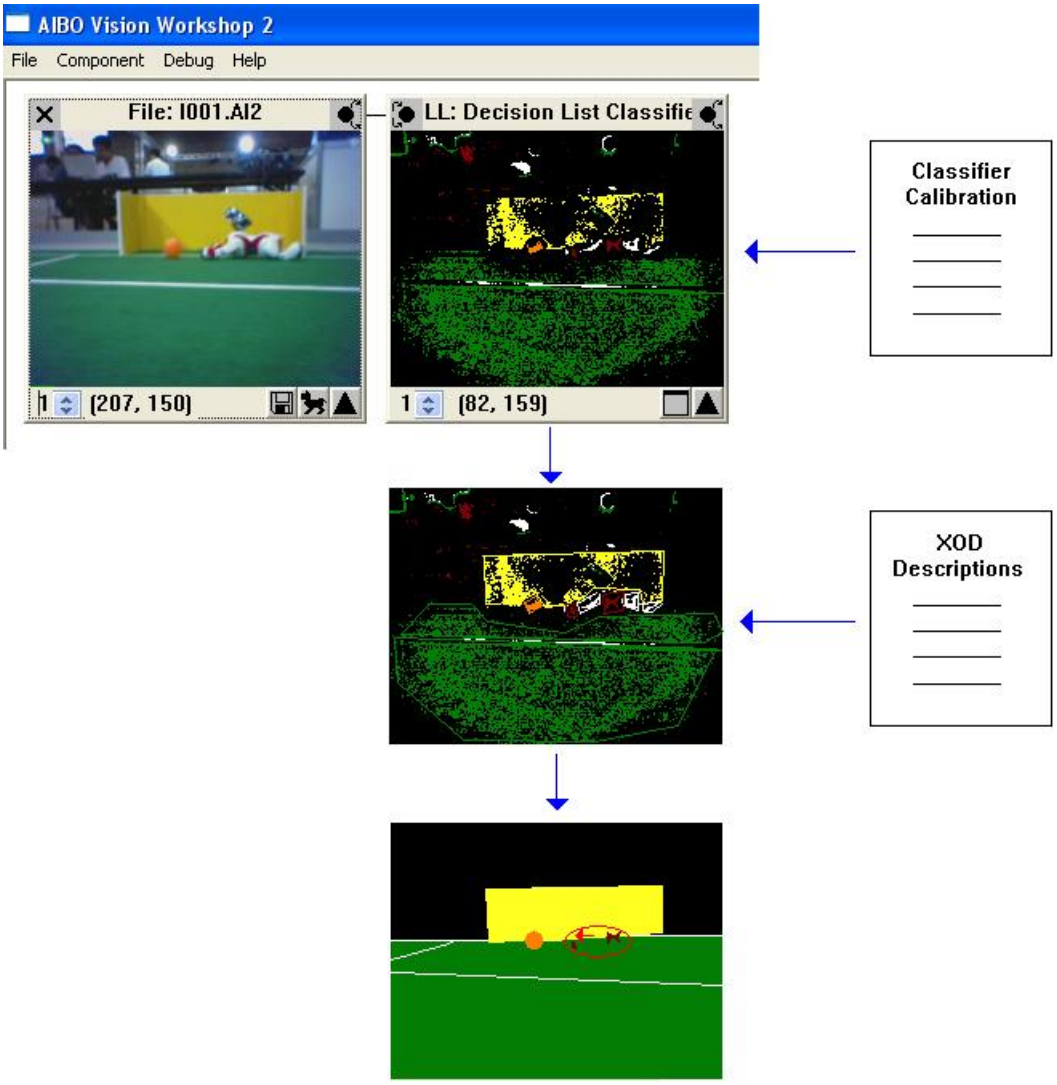


Figure 13.1: Our advanced vision pipeline utilises a fast, illumination-tolerant object recognition system along with a generic object description language. This provides the greatest possible robustness and flexibility to the system. The user of the system need only supply the sparse classifier calibration and an XOD description for each object in the environment.

Stop Sign XOD

```
<object>
    <id>STOP_SIGN_BLOB</id>
    <blob>
        <colour>RED</colour>
    </blob>
</object>
```

```
<object>
    <id>STOP_SIGN</id>
        <source>STOP_SIGN_BLOB</source>
    <edge>
    <not><colour>RED,WHITE</colour></not>
    </edge>
    <vectorise>hexagon</vectorise>
</object>
```

Figure 13.2: Some XOD could require a programmer to write an extension. For example, to detect hexagons (as in this code) would require a custom vectorisation module.

their shape. The vision processing system locates candidate blobs within the image that matches each description. The XOD also specifies how to select the correct blob for each object.

Usually we will wish to find the edges of the blobs, not merely a randomly shaped cluster of pixels inside them. For example, balls should be round, goals and beacons rectangular and we certainly require the outline of the AIBO's uniform in order to apply the posture detection technique in Chapter 9. Therefore, very often the XOD will specify that objects be selected by shape. The system already knows how to find many vectorisations efficiently, including straight lines (Chapter 6), circles and squares (Chapter 5) and even the posture vector of the AIBOs (Chapter 9). However, it is plain to see that determining specific shapes that are unknown to the system will require new code to be written. If we were required to locate hexagons (for example, stop signs), we would need to write a code module that could detect them: there is no way to specify how to detect a hexagon using XOD. However, once the module was written, the XOD for a stop sign would be something like Figure 13.2.

For the RoboCup domain, the XOD is fairly large. There are four beacons, two goals, field lines, the ball and various robots that the system must know how to find. The XODs that we have shown in this thesis (both in this chapter and in Chapter 11) have been simplified from what works in a real system. We did this so we could illustrate the concepts more clearly. It is not sufficient, for example, to find any pink, rectangular blob that is on top of a yellow, rectangular blob

and call it a beacon. This would mean that certain combinations of clothes in the crowd would be identified as beacons. Beacons should also be checked by their relationship to the horizon and field — both in distance and size. There are many such checks that are required. The ball must be on the field and of an appropriate size especially compared to the goals, as it is easy to confuse a few misclassified orange pixels in a yellow goal with a ball that is very far away. Goals should also be checked against the horizon to eliminate the problem of people dressed in blue standing on the edge of the field. All of these checks tend to extend the length of the XOD. Nevertheless, the concept remains the same and final runtime performance is not greatly impacted.

13.1.1 Summary

Machine vision for mobile, autonomous robotics is both a difficult and active research field. As we noted in the introduction, there are many available systems with different approaches and advantages that address this topic. We selected the Bruce *et al.* pipeline for comparison not because it represented the state of the art, but because it or similar pipelines are popular and are used by many people in the field of robotics. Such pipelines are especially popular in robotic applications where the main research effort is not in vision, and the development team simply require vision to "work" so that they can test something else. For these reasons it has emerged as something of a standard within the community.

We saw the performance data for our raw pipeline in Section 8.3. In Table 13.1 we examine our systems performance once XOD is also incorporated. We do not see a marked increase in runtime cost because the XOD is translated into C++ and compiled into the system. However there is a small performance penalty for using XOD because of the dynamic memory allocation required to maintain lists of objects. We compare our system against the Bruce *et al.* pipeline.

The data in Table 13.1 was gathered from over 1000 images in our database, all typical scenes from RoboCup. The objects present in the environment include balls (sometimes more than one per image), beacons, goals and opponent robots. We do not have an implementation of the Bruce *et al.* pipeline that includes the dog recognition component that is present in our system, so the data for that pipeline does not reflect this processing, but ours does.

We see from these results that the one thing the Bruce *et al.* pipeline still

	Min frame (ms)	Max frame (ms)	Avg frame (ms)
The Bruce *et al.* pipeline	25.97	26.81	26.37
Our pipeline (no XOD)	25.32	29.07	27.09
Our pipeline (with XOD)	26.48	34.51	29.91

Table 13.1: Our pipeline implements a robust and flexible object recognition system that is fast enough to use in the context of real-time vision system processing on mobile autonomous robotics.

	Our pipeline	Bruce *et al.*
Operation in constant lighting conditions	✓	✓
Operation in variable lighting conditions	✓	×
Real-time object recognition	✓	✓
Predictable speed per frame	×	✓
Versatility to changing operational domains	✓	×
Support for complex object recognition (e.g. AIBOs)	✓	×
Object analysis using edges	✓	×
Object recognition in blurry images	✓	×

Table 13.2: A summary of the features of our system.

has in its favour is that the execution time is very predictable. This is because there is nothing in the system that changes depending on the information within each frame. Our pipeline must use different amounts of dynamically allocated memory if there are more blobs in a particular frame. In addition our system will spend more time border following and edge detecting if the objects that require these routines are large within the images.

Table 13.2 summarises the features of our improved object recognition system as opposed to the features available to the pipeline of Bruce *et al.*

13.2 Future Work

The techniques that we have presented in this thesis represent a significant advance in vision processing for mobile autonomous robotics. Nevertheless the field is in its infancy and there are certainly many further avenues to explore. We will examine each of the areas addressed in this thesis in turn.

13.2.1 Computational Complexity of Image-Analysis Algorithms

We have examined several common low-level image processing algorithms and improved on their runtime performance. They were an image segmentation algorithm, an edge detection algorithm and a straight-line vectorisation algorithm. Nevertheless there are many low-level image processing concepts that we have not yet addressed.

For example, our system is incapable at this time of using textures to discriminate between objects, although this is an important and well studied area [106]. Texture analysis techniques generally are exceptionally computationally expensive (even more so than edge detection) but there is little doubt that textures are as important to human vision as edges [70]. At some stage, efficient texture analysis algorithms must be developed and employed in machine vision systems for mobile, autonomous robots. There is a large gap in the literature as to exactly how this may be accomplished. At the time of writing the author is unaware of any real-time robotic vision systems that employ texture mapping techniques as part of their processing pipeline — they are simply too slow. Some simple texture mapping has been used, for example on the AIBO by Sony's AIBO Mind software. This software examines areas within each image for the black/white pattern that identifies the AIBO's charging station. In a way this could be considered primitive texture mapping but in reality it is doing little more than colour thresholding.

Another area that we anticipate will be crucial in future robotic vision systems is that of optical flow [63]. This technique is concerned by information that is obtained by analysing the difference *between* two subsequent incoming frames. Our system examines the information only on a per-frame basis and therefore any information that can be gleaned by a differential analysis of two consecutive frames will be lost. At the time of writing optical flow algorithms are only in early stages of development and even though they are somewhat slow and inaccurate now, we expect to see more robotic vision systems incorporating these kinds of algorithms in the near future.

A great deal of information, in particular depth and perspective, can also be gained from stereoscopic vision [94]. These systems are typically very processor-intensive. Indeed, stereoscopic vision systems take *greater* than twice the pro-

cessing time of a monoscopic system unless dedicated hardware is used for each camera. We expect that the development of faster and cheaper processors will somewhat alleviate this problem, but it remains to be seen whether the benefits associated with stereoscopic vision will be worth the added computational cost.

13.2.2 Vision for Dynamic and Unpredictable Conditions

A survey of recent literature will show that the most intensive research in robotic vision at the current time is in adaptability to changing lighting conditions. We have addressed this in this thesis by utilising a sparse classification algorithm in combination with fast edge detection for object recognition. However, our algorithm is not a perfect solution to the problem. While our algorithm is successful over a very wide range of illumination conditions, if the lighting conditions change significantly then our algorithm will still fail. For example, the same calibration can be used in our system for indoor, outdoor, direct light or shade but a system that is calibrated for daylight will not adapt to evening conditions. A true solution to the problem of changing lighting conditions therefore remains undiscovered. We have examined some of the current work in this area in Chapter 7.

13.2.3 Reliance on Domain-Specific Knowledge in Object Recognition

Certainly the ideal robot would have the ability to encounter unrecognised objects in its environment and learn their properties and utility. It would then store this information so that the next time it encountered the same object, or similar objects, it would recognise them. Children certainly show the capacity to do this in the early stages of development and so we would like a robot to mimic this behaviour. This would drastically reduce the development time for robots because there would be no need to codify information concerning every possible object in the environment. Nevertheless, this goal is still a very long way away.

In this thesis we have presented an implementation-independent language that can be used to store information about the objects that we expect the robot to encounter. This is extremely useful for abstracting the details of the environmental information from the vision system code, but our system still re-

quires someone to write object descriptions. The next logical step would be to apply machine learning techniques to the XOD language and have the system automatically generate descriptions of objects that it was shown. This step would reduce the labour involved with generating object descriptions. Perhaps, after this is achieved, the somewhat ambitious task of automatically recognising previously un-encountered objects in an arbitrary environment could be addressed. We have noted in the literature review in Chapter 10 some of the current work that is heading in this direction.

Part VI

Appendix

Appendix A

C++ Implementation of our Border Following Algorithm

We attach, for reference, the C++ implementation of our edge detection algorithm. The C++ class "EdgeDetector" implements our early edge detection, while "RunningEdgeDetector" implements partial late edge detection and is used in conjunction with the border following algorithm in Section 5.2.2 to implement complete late edge detection.

```
/* Mi-Pal 2005
** EdgeDetector.h
** Author: Nathan Lovell
*********************************************************
** This file is copyright to the authors.
** It formed part of the Mi-Pal 2005 entry in
** the Robocup Legged league
*************************************************************
** This code is released under GPL V2.0 (see license.txt)
** You should have received a copy of the GNU General Public
** Licensealong with this program; if not, write to the Free
** Software Foundation, Inc., 59 Temple Place, Suite 330,
** Boston, MA  02111-1307  USA
** Note: No warantee of any kind is associated with this code.
*************************************************************
*/

#ifndef EDGES_H
#define EDGES_H

#include "AVWImage.h"

class EdgeDetector
{
```

```
public:
. EdgeDetector();
. ~EdgeDetector();
. void NewImage(AVWImageBase *pSrc, int window);
. void MarkEdges(AVWClassifiedImage *pDest, int sensitivity);
.
private:
. int *m_diffsX;
. int **m_diffsY;
. int m_nx, m_ny;
. int m_nWindow;
. int m_nSensitivity;
. AVWImageBase *m_pSrc;
};

class RunningEdgeDetector
{
public:
. RunningEdgeDetector();
. ~RunningEdgeDetector();
. void NewRun(AVWColour clr);
. bool NextPixel(AVWColour clr);
.
private:
. AVWColour m_startClr;
. AVWColour m_prevClr;
};

#endif
```

```
/* Mi-Pal 2005
** EdgeDetector.cpp
** Author: Nathan Lovell
************************************************************
** This file is copyright to the authors.
** It formed part of the Mi-Pal 2005 entry in
** the Robocup Legged league
************************************************************
** This code is released under GPL V2.0 (see license.txt)
** You should have received a copy of the GNU General
** Public License along with this program; if not,
** write to the Free Software Foundation, Inc.,
** 59 Temple Place, Suite 330, Boston,
** MA  02111-1307  USA
** Note: No warantee of any kind is associated with this code.
************************************************************
*/

#include "EdgeDetector.h"
#include "ClassifiedColours.h"

#define WSZ (2 * m_nWindow + 1)

EdgeDetector::EdgeDetector()
{
. m_nWindow = 0;
. m_nx = 0;
. m_ny = 0;
. m_pSrc = 0;
. m_diffsX = 0;
. m_diffsY = 0;
. m_nSensitivity = 0;
}
```

```
EdgeDetector::~EdgeDetector()
{
. if (m_diffsX != 0)
. delete[] m_diffsX;
. if (m_diffsY != 0)
. {
.   for (int i = 0; i < m_nx; i++)
.   {
.     delete[] m_diffsY[i];
.   }
.   delete[] m_diffsY;
. }
}

void EdgeDetector::NewImage(AVWImageBase *pSrc, int window)
{
. if ((m_nx != pSrc->GetSizeX()) || (m_ny != pSrc->GetSizeY()))
. {
.   if (m_diffsX != 0)
.     delete[] m_diffsX;
.   m_diffsX = 0;
.   if (m_diffsY != 0)
.   {
.     for (int i = 0; i < m_nx; i++)
.     {
.       delete[] m_diffsY[i];
.     }
.     delete[] m_diffsY;
.     m_diffsY = 0;
.   }
. }

. m_nWindow = window;
. m_nx = pSrc->GetSizeX();
. m_ny = pSrc->GetSizeY();
```

```
. m_pSrc = pSrc;

. if (m_diffsX == 0)
.  m_diffsX = new int[WSZ];
. if (m_diffsY == 0)
. {
.   m_diffsY = new int*[m_nx];
.   for (int i = 0; i < m_nx; i++)
.   {
.     m_diffsY[i] = new int[WSZ];
.   }
. }
}

void EdgeDetector::MarkEdges(AVWClassifiedImage *pDest,
.                            int sensitivity)
{
. if ((m_diffsX == 0) || (m_nWindow == 0))
.   return;
. m_nSensitivity = sensitivity;

. int ptrX = 0, ptrY = 0;
. for (int y = 0; y < m_ny - 5; y++)
. {
.   for (int x = 0; x < m_nx - 5; x++)
.   {
.     AVWColour clr1 = m_pSrc->GetPixel(x, y);
.     AVWColour clr2 = m_pSrc->GetPixel(x + 4, y);
.     AVWColour clr3 = m_pSrc->GetPixel(x, y + 4);
.     m_diffsX[ptrX] = (int)(clr2 - clr1);
.     m_diffsY[x][ptrY] = (int)(clr3 - clr1);
.
.     if ((x >= WSZ) && (y >= m_nWindow) &&
.         (y < m_ny - m_nWindow - 1))
.     {
```

```
.       int a = 0;
.       int currentX = (ptrX - m_nWindow) < 0
.         ? (WSZ - m_nWindow + ptrX) : (ptrX - m_nWindow);
.       for (int i = 0; i < WSZ; i++)
.       {
.         if (i != currentX)
.           a += m_diffsX[i];
.       }
.       a /= (WSZ - 1);
.
.       if (m_diffsX[currentX] > a + m_nSensitivity)
.       {
.         pDest->SetPixel2(x - m_nWindow, y,
.                          COLOUR::ID_BOUNDRY);
.       }
.     }
.
.     if ((y >= WSZ) &&
.         (x >= m_nWindow) &&
.         (x < m_nx - m_nWindow - 1) &&
.         (pDest->GetPixel2(x, y - m_nWindow) !=
.           COLOUR::ID_BOUNDRY))
.     {
.       int b = 0;
.       int currentY = (ptrY - m_nWindow) < 0
.         ? (WSZ - m_nWindow + ptrY) : (ptrY - m_nWindow);
.       for (int i = 0; i < WSZ; i++)
.       {
.         if (i != currentY)
.           b += m_diffsY[x][i];
.       }
.       b /= (WSZ - 1);
.
.       if (m_diffsY[x][currentY] > b + m_nSensitivity)
.       {
```

```
.         pDest->SetPixel2(x, y - m_nWindow, COLOUR::ID_BOUNDRY);
.       }
.     }
.
.     ptrX++;
.     if (ptrX == WSZ)
.       ptrX = 0;
.   }
.
.   ptrY++;
.   if (ptrY == WSZ)
.     ptrY = 0;
. }
.
. for (int y = 0; y < m_ny - 1; y++)
. {
.   for (int x = 0; x < m_nx - 1; x++)
.   {
.     if ((x == m_nWindow - 1) || (x == m_nx - m_nWindow - 1))
.     {
.       if ((y >= m_nWindow - 1) && (y <= m_ny - m_nWindow - 1))
.       {
.         pDest->SetPixel2(x, y, COLOUR::ID_BOUNDRY);
.         continue;
.       }
.     }
.     if ((y == m_nWindow - 1) || (y == m_ny - m_nWindow - 1))
.     {
.       if ((x >= m_nWindow - 1) && (x <= m_nx - m_nWindow - 1))
.       {
.         pDest->SetPixel2(x, y, COLOUR::ID_BOUNDRY);
.         continue;
.       }
.     }
.   }
```

```
. }
}

#define Y_THRESH 15
#define UV_THRESH 15

RunningEdgeDetector::RunningEdgeDetector()
{
}

RunningEdgeDetector::~RunningEdgeDetector()
{
}

void RunningEdgeDetector::NewRun(AVWColour clr)
{
. m_startClr = clr;
. m_prevClr = clr;
}

bool RunningEdgeDetector::NextPixel(AVWColour clr)
{
. int diffY = ABS(m_prevClr.GetDataA() - clr.GetDataA());
. if (diffY > Y_THRESH)
.   return true;
.
. int diffU = ABS(m_startClr.GetDataB() - clr.GetDataB());
. int diffV = ABS(m_startClr.GetDataC() - clr.GetDataC());
.
. if ((diffU * diffU + diffV * diffV) > (UV_THRESH * UV_THRESH))
.   return true;
.
. return false;
}
```

Appendix B

C++ script for AVW2

We attach, for reference, a C++ script for AVW2 that implements a colour classification filter by running the code intended for the embedded device.

```
1 // Uncomment to compile instead of interpret
2 //#pragma compile
3 // Embedded code for colour classifier
4 #include "DecisionListClassifier.cpp"

5 void main()
6 {
7    // Set up the scripted filter
8    SetNameForCurrentFilter("Classifier");
9    SetInputTypesForCurrentFilter("Image");
10   SetOutputTypesForCurrentFilter("Classifier|Image");
11   Whiteboard *pBoard =
          GetWhiteboardForCurrentFilter();
12   AVWImage* pInput =
          pBoard->GetInputForCurrentFilter("Image");
13   AVWImage* pOutput =
          pBoard->GetOutputImageForCurrentFilter("Image");

14   // Create my colour classifier using the code
     // to run on the embedded device
15   DecisionListClassifier dlc;
16   dlc.InitFromListFile("clscal.dlc");

17   // Classify my image using the code to run on the
     // embedded device
18   unsigned char *pClassifiedData =
           dlc.Classify(pInput, pOutput);

19   // Set the output data for the next filter
```

```
20   pBoard->SetOutputDataForCurrentFilter("Classifier",
           pClassifiedData);
21 }
```

Glossary of Terms

Big-O Notation We use the classical notation of growth in functions for lower (Ω), upper (O) and exact (Θ) worst case complexity for algorithms [2]. A function $f(n)$ is $O(g(n))$ if $\exists n_0$ and constant $c > 0$ such that $f(n) < cg(n), \forall n > n_0$ where n is the size of the input. A function $f(n)$ is $\Omega(g(n))$ if $\exists n_0$ and $c > 0$ such that $f(n) > cg(n), \forall n > n_0$. A function is $\Theta(g(n))$ if $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$.

C/C++ C and C++ are both programming languages. C is one of the most popular programming languages in the world, particularly for low-level tasks such as operating systems and when performance is required. C++ is a development of C that allows object-oriented programming. The AIBOs are programmed using C++.

CCD (Charge coupled device) A matrix of small sensing devices capable of detecting the intensity of light. The common name for this device is a digital camera.

Colour Space A colour space is a means of discretizing and assigning a value to each colour within a range of perceived colours. There are three component colours of light (red, green and blue) and from these three wavelengths it is possible to create any other colour. This means that colour spaces must be represented as three dimensions. The simplest possible colour space is the RGB colour space. Each dimension in the space represents the amount of either red, green or blue that has been used to mix the colour. RGB is

common because it is easy to visualise and understand — it is a cube in the colour space.

Our research uses the YUV colour space, where again, each pixel is represented by three components. The Y component is the intensity of the light — if you view only this component then you get a black, white and grey image. The U and V are called chromatism. U represents the balance between green/red (which are opposite in this colour space) while V is the balance between blue/yellow. The YUV space can best be thought of as an inverted square pyramid where the apex sits on the $U = 0, V = 0$ plane and the pyramid extends up symmetrically around the Y axis. When you compare this to the RGB colour space it is easy to see why there is no linear transform between the two colour spaces. For a more complete description of colour spaces see [126].

Although we use the YUV colour space, there is nothing in our research that is intrinsic to this particular space. Our algorithms and methods could be applied without modification to any colour space.

Cross-Compiled Code In a normal development environment, code is written on the same machine on which it will eventually be run. The compilation process takes a human-readable source set (such as a Java or C++ program) and translates it into a set of instructions that the machine will be able to understand directly. Sometimes it is impossible to develop code on the target device — there may be no keyboard, screen or file system for example. In this case code must be developed on a PC and the compiler will translate it into instructions that the target machine will understand directly. This is called cross-compilation.

DLL (Dynamically Linked Library) A module of compiled code that is not a complete program in itself. Instead it is linked (dynamically *linked* library) to an executable piece of code at runtime. Functions in the module can be called from the main executable.

Fps (Frames per second) The number of picture (frames) delivered from the camera to the software each second.

Frame One of a repeating pattern of inoming data. We use this word most often to refer to an incoming image though it can also refer to incoming

sensor data.

HCI (Human-Computer Interaction) The area of research that deals with how humans use computers. This most often refers the graphical user interface of an application but other interfaces are also included such as virtual reality and speach processing.

IDE (Integrated Development Environment) A suite of tools that support code development for a particular platform. An IDE may contain, for example, a text editor, compiler and debugger.

JIT (Just-in-time) Debugging A debugging facility that enables a debugger to be attached to a program just before it crashes, *just in time*. The developer can then examine the internals of the program to discover the cause of the crash.

Ladar (Laser-Radar) A relatively new technology that uses a laser-based, radar-like device to build a 3D map of the environment that includes texture information.

Mobile Autonomous Robot A robot that is capable of moving around its environment and making its own decisions on the actions it will perform, based on input that is received from on-board sensors. In particular this type of robot precludes any direct human intervention in its execution.

OOP (Object-Oriented Programming) A methodology for programming. OOP encapsulates the state of the program in a set of objects. State data is hidden within each object and only accessible through method calls to the object.

PC (Personal Computer) We use this term in contrast to the type of computer that is found on a mobile autonomous robot.

Rasterisation The process by which visual information is converted to a matrix of pixels. Rasterisation is a deterministic process — the same visual information will yield the same pixel matrix every time.

SLAM (Simultanous Localisation and Mapping) A popular area of research in robotics. A robot wanders around its environment while simultaneously discovering landmarks and building a map.

XML An extensible markup language for arbitrary data transfer. XML is used in increasing numbers of applications and framworks including many document formats and communication protocols.

Bibliography

[1] I. Abdou and W. Pratt. Quantitative design and evaluation of enhancement/thresholding edge detectors. *Proceedings of the IEEE*, 67(5):753–763, 1979, ISSN: 0018-9219.

[2] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, U.S.A., 1974, ISBN: 0-2010-0029-6.

[3] H. Altunbasak and H. Trussell. Colorimetric restoration of digital images. *IEEE Transactions on Image Processing*, 10(3):393–402, 2001, ISSN: 1057-7149.

[4] F. Anzani, D. Bosisio, M. Matteucci, and D. Sorrenti. On-line colour calibration in non-stationary environments. In *Proceedings of RoboCup 2005 — Robot Soccer World Cup IX, Osaka, Japan*. Springer-Verlag, 2005, *To appear*.

[5] G. Araujo, D. Devadas, K. Keutzer, S. Liao, S. Malik, A. Sudarsanam, S. Tjiang, and A. Wang. *Challenges in Code Generation for Embedded Processors*, pages 49–64. Kluwer Academic Publishers, Germany, 1995, ISBN: 0-7923-9577-8.

[6] D. Argiro, S. Kubica, M. Young, and S. Jorgensen. *KHOROS: An Integrated Development Environment for Scientific Computing and Visualization*. Available from: http://www.khoral.com.

[7] C. Bajaj. Proving geometric algorithm non-solvability: An application of factoring polynomials. *Journal of Symbolic Computation*, 2(1):99–102, 1986, ISSN: 0747-7171.

[8] D. Ballard. *Generalizing the Hough Transform to Detect Arbitrary Shapes*, pages 714–725. Morgan Kaufmann, U.S.A., 1987, ISBN: 0-9346-1333-8.

[9] K. Barnard, V. Cardei, and B. Funt. A comparison of computational color constancy algorithms — part I: Methodology and experiments with synthesized data. *IEEE Transactions on Image Processing*, 11(9):972–984, 2002, ISSN: 1057-7149.

[10] K. Barnard, V. Cardei, and B. Funt. A comparison of computational color constancy algorithms — part II: Experaments with image data. *IEEE Transactions on Image Processing*, 11(9):385–996, 2002, ISSN: 1057-7149.

[11] B. Bartlett, V. Estivill-Castro, and S. Seymon. Dogs or robots — why do we see them as robotic pets rather than canine machines? In *Proceedings of the 5th Australasian User Interface Conference*, pages 7–14. Australian Computer Society, 2004, ISBN: 1-9206-8210-4.

[12] B. Bartlett, V. Estivill-Castro, S. Seymon, and A. Tourky. Robots for pre-orientation and interaction of toddlers and preschoolers who are blind. In *Proceedings of the 2003 Australasian Conference on Robotics and Automation*, CD-Rom Proceedings, 2003, ISBN: 0-9587-5835-2.

[13] G. Baxes. *Digital Image Processing: Principles and Applications*. John Wiley and Sons, Australia, 1994, ISBN: 0-4710-0949-0.

[14] R. Bergevin and M. Levine. Generic object recognition: Building and matching course descriptions from line drawings. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(1):19–36, 1993, ISSN: 0162-8828.

[15] S. Bhandankar, Y. Zhang, and W. Potter. An edge detection technique using genetic algorithm based optimization. *Pattern Recognition*, 27(9):1159–1180, 1994, ISSN: 0031-3203.

[16] J. Bruce, T. Balch, and M. Veloso. Fast and inexpensive color segmentation for interactive robots. In *Proceedings of the International Conference on Intelligent Robots and Systems*, pages 2061–2066. IEEE Computer Society Press, 2000, ISBN: 0-7803-6348-5.

[17] J. Buck, S. Ha, E. Lee, and D. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *International Journal in Computer Simulation*, 4(2):155–182, 1994, ISSN: 1055-8470.

[18] S. Buluswar and A. Draper. Color models for outdoor machine vision. *Computer Vision and Image Understanding*, 85(2):71–99, 2002, ISSN: 1077-3142.

[19] T. Caelli and D. Reye. On the classification of image regions by colour, texture and shape. *Pattern Recognition*, 26(4):461–470, 1993, ISSN: 0031-3203.

[20] J. Cai, A. Goshtasby, and C. Yu. Detecting human faces in color images. *Image and Vision Computing*, 18(1):63–75, 2000, ISSN: 0262-8856.

[21] D. Cameron and N. Barnes. Knowledge-based autonomous dynamic colour calibration. In *Proceedings of RoboCup 2003 — Robot Soccer World Cup VII, Padua, Italy*, pages 226–237. Springer-Verlag, 2003, ISBN: 3-5402-2443-2.

[22] J. Canny. A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 8(6):679–698, 1986, ISSN: 0162-8828.

[23] M. Cazorla and F. Escolano. Two bayesian methods for junction classification. *IEEE Transactions on Image Processing*, 12(3):317–327, 2003, ISSN: 1057-7149.

[24] W. Cendrowska. PRISM: An algorithm for inducing modular rules. *International Journal of Man-Machine Studies*, 27(4):349–370, 1987, ISSN: 0020-7373.

[25] S. Chalup, R. Middleton, R. King, L. Li, T. Moore, C. Murch, and M. Quinlan. The NUbots' team description for 2004. In *Proceedings of RoboCup 2004 — Robot Soccer World Cup VIII, Lisbon, Portugal*, CD-Rom Proceedings. Springer-Verlag, 2004, ISBN: 3-5402-5046-8.

[26] J. Chen, E. Chung, R. Edwards, N. Wong, E. Mak, R. Sheh, M. Kim, A. Tang, N. Sutanto, B. Hengst, C. Sammut, and W. Uther. A description of the rUNSWift 2003 legged robot soccer team. In *Proceedings of RoboCup 2003 — Robot Soccer World Cup VII, Padua, Italy*, CD-Rom Proceedings. Springer-Verlag, 2003, ISBN: 3-5402-2443-2.

[27] S. Chen, M. Siu, T. Vogelgesang, T. Yik, B. Hengst, S. Bao Pham, and C. Sammut. The unsw robocup 2001 sony legged robot league team. In *Proceedings of RoboCup 2001 — Robot Soccer World Cup V, Seattle, USA*, page 39. Springer-Verlag, 2001, ISBN: 3-5404-3912-9.

[28] F. Chin, J. Snoeyink, and C. Wang. Finding the medial axis of a simple polygon in linear time. In *Proceedings of the 6th International Symposium on Algorithms and Computation*, pages 382–391. Springer-Verlag, 1995, ISBN: 3-5406-0573-8.

[29] K. Cho and S. Dunn. Learning shape classes. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(9):882–888, 1994, ISSN: 0162-8828.

[30] H. Choi, S. Choi, and H. Moon. Mathematical theory of medial axis transform. *Pacific Journal of Mathematics*, 181(1):57–88, 1997, ISSN: 0030-8730.

[31] I. Dahm, S. Deutsch, M. Hebbel, and A. Osterhues. Robust color classification for robot soccer. In *Proceedings of RoboCup 2003 — Robot Soccer World Cup VII, Padua, Italy*, pages 64–75. Springer-Verlag, 2003, ISBN: 3-5402-2443-2.

[32] N. Danielsson. Axiomatic discrete geometry. Master's thesis, Imperial College of Science, Technology and Medicine, University of London, U.K., 2002.

[33] I. Debled-Rennesson and J. Réveillès. A linear algorithm for segmentation of digital curves. *International Journal of Pattern Recognition and Artificial Intelligence*, 9(4):635–662, 1995, ISSN: 0218-0014.

[34] C. D'Elia, G. Poggi, and G. Scarpa. A tree-structured markov random field model for bayesian image segmentation. *IEEE Transactions on Image Processing*, 12(10):1259–1273, 2003, ISSN: 1057-7149.

[35] G. DeSouza and A. Kak. Vision for mobile robot navigation: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(22):237–267, 2002, ISSN: 0162-8828.

[36] L. Dorst and A. Smeulders. Discrete representation of straight lines. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 6(4):450–463, 1984, ISSN: 0162-8828.

[37] B. Draper. Learning control strategies for object recognition. In K. Ikeuchi and M. Veloso, editors, *Symbolic Visual Learning*. Oxford University Press, U.K., 1996, ISBN: 0-1950-9870-6.

[38] B. Draper, U. Ahlrichs, and D. Paulus. Adapting object recognition across domains: A demonstration. *Lecture Notes in Computer Science*, 2095:256, 2001, ISSN: 0302-9743.

[39] R. Duda and P. Hart. *Pattern Classification and Scene Analysis*. Wiley, U.S.A., 1973, ISBN: 0-4712-2361-1.

[40] R. Duda, P. Hart, and D. Stork. *Pattern Classification*. Wiley, U.S.A., 2001, ISBN: 0-4710-5669-3.

[41] R. Dufour, E. Miller, and N. Galatsanos. Template matching based object recognition with unknown geometric parameters. *IEEE Transactions on Image Processing*, 11(12):1385–1396, 2002, ISSN: 1057-7149.

[42] M. Elstrom, P. Smith, and M. Abidi. Stereo-based registration of ladar and color imagery. In *Proceedings of Intelligent Robots and Computer Vision XVII: Algorithms, Techniques, and Active Vision*, pages 343–354. Spie Web Publications, 1998, ISBN: 0-8194-2983-X.

[43] N. Eua-Anant and L. Udpa. Boundary detection using simulation of particle motion in a vector image field. *IEEE Transactions on Image Processing*, 8(11):1560–1571, 1999, ISSN: 1057-7149.

[44] T. Fong, C. Thorpe, and C. Baur. Collaboration, dialogue, and human-robot interaction. In *Robotics Research: Proceedings of the The 10th International Symposium*, pages 255–270. Springer-Verlag, 2003, ISBN: 3-5400-0550-1.

[45] A. Forrest. Interactive interpolation and approximation by Beźier polynomials. *Computer Aided Design*, 22(9):527–537, 1990, ISSN: 0010-4485.

[46] D. Forsyth. A novel algorithm for color constancy. *International Journal of Computer Vision*, 5:5–36, 1990, ISSN: 0920-5691.

[47] G. Genello, J. Cheung, S. Billis, and Y. Saito. Graeco-Latin squares design for line detection in the presence of correlated noise. *IEEE Transactions on Image Processing*, 9(4):609–622, 2000, ISSN: 1057-7149.

[48] P. Giusto, G. Martin, and E. Harcourt. Reliable estimation of execution time of embedded software. In *Proceedings of the Design, Automation and Test in Europe Conference*, pages 580–589. IEEE Computer Society Press, 2001, ISBN: 0-7695-0993-2.

[49] C. Goenner, M. Rous, and K. Kraiss. Real-time adaptive colour segmentation for the robocup middle size league. In *Proceedings of RoboCup 2004 — Robot Soccer World Cup VIII, Lisbon, Portugal*, pages 402–409. Springer-Verlag, 2004, ISBN: 3-5402-5046-8.

[50] R. Gonzalez and R. Woods. *Digital Image Processing*. Prentice Hall, U.S.A., 1992, ISBN: 0-2011-8075-8.

[51] K. Gunnarsson, F. Wiesel, and R. Rojas. The color and the shape: Automatic on-line color calibration for autonomous robots. In *Proceedings of RoboCup 2005 — Robot Soccer World Cup IX, Osaka, Japan*. Springer-Verlag, 2005, *To appear*.

[52] J. Gutmann and C. Schlegel. Amos: comparison of scan matching approaches for self localization in indoor environments. In *Proceedings of the 1st Euromicro Workshop on Advanced Mobile Robotics*, pages 61–71. IEEE Computer Society Press, 1996, ISBN: 0-8186-7695-7.

[53] J. Gutmann, T. Weigel, and B. Nebel. A fast, accurate and robust method for self-localization in polygonal environments using laser range finders. *Advanced Robotics*, 14(8):651–667, 2001, ISSN: 0169-1864.

[54] A. Halme, K. Koskinen, V-P. Aarnio, S. Salmi, I. Leppnen, and S. Ylnen. Workpartner — future interactive service robot. In *Proceedings of the Millenium of Artificial Intelligence Conference, 9th Finnish Conference on Artificial Intelligence*, CD-Rom Proceedings. Finnish Artificial Intelligence Society, 2000, ISBN: 9-5122-5128-0.

[55] R. Haralick and L. Shapiro. *Computer and Robot Vision*. Addison-Wesley, U.S.A., 1992, ISBN: 0-2015-6943-4.

[56] J. Hartigan. *Clustering Algorithms*. Wiley, U.S.A., 1975, ISBN: 0-4713-5645-X.

[57] M. Heath, S. Sarkar, T. Sanocki, and K. Bowyer. Robust visual method for assessing the relative performance of edge-detection algorithms. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19(12):1338–1359, 1997, ISSN: 0162-8828.

[58] G. Heidemann and H. Ritter. A neural 3-D object recognition architecture using optimized gabor filters. In *Proceedings of the 13th International Conference on Pattern Recognition*, pages 70–74. IEEE Computer Society Press, 1996, ISBN: 8-8186-7472-5.

[59] L. Hermes and J. Buhmann. A minimum entropy approach to adaptive image polygonization. *IEEE Transactions on Image Processing*, 12(10):1243–1258, 2003, ISSN: 1057-7149.

[60] J Hershberger and J. Snoeyink. Speeding up the Douglas-Peucker line simplification algorithm. In *Proceedings of the 5th International Symposium on Spatial Data Handling*, pages 134–143, 1992, ISBN: 0-9633-5320-9.

[61] J. Hershberger and J. Snoeyink. An $O(n \log n)$ implementation of the Douglas-Peucker algorithm for line simplification. In *Proceedings of the 10th Annual Symposium on Computational Geometry*, pages 383–384. ACM Press, 1994, ISBN: 0-89791-648-4.

[62] W. Higgins and C. Hsu. Edge detection using 2D local structure information. *Pattern Recognition*, 27(2):277–294, 1994, ISSN: 0031-3203.

[63] B. Horn and B. Schunck. *Determining Optical Flow*, pages 389–407. Jones and Bartlett Publishers, U.S.A., 1992, ISBN: 0-8672-0452-4.

[64] J. Hornegger and H. Niemann. Statistical learning, localization, and identification of objects. In *Proceedings of the 5th International conference on computer vision*, pages 914–919. IEEE Computer Society Press, 1995, ISBN: 0-8186-7042-8.

[65] P. Hough. Methods and means for recognising complex patterns, 1962. Patent (U.S.A.): 3 069 654.

[66] H. Hsin. Texture segmentation using modulated wavelet transform. *IEEE Transactions on Image Processing*, 9(7):1299–1302, 2000, ISSN: 1057-7149.

[67] G. Iannizzotto and L. Vita. Fast and accurate edge-based segmentation with no contour smoothing in 2-D real images. *IEEE Transactions on Image Processing*, 9(7):1232–1238, 2000, ISSN: 1057-7149.

[68] J. Illingworth and J. Kittler. A survey of the Hough transform. *Computer Vision, Graphics and Image Processing*, 44(1):87–116, 1988, ISSN: 0734-189X.

[69] D. Ioammou, W. Huda, and A. Laine. Circle recognition through a 2 d hough transform and radius histogramming. *Image and Vision Computing*, 17:15–26, 1999, ISSN: 0262-8856.

[70] R. Jain, R. Kasturi, and B. Schunck. *Machine Vision*. Mcgraw-Hill Inc., U.S.A., 1995, ISBN: 0-0703-2018-7.

[71] M. Jüngel, J. Hoffmann, and M. Lötzsch. A real-time auto-adjusting vision system for robotic soccer. In *Proceedings of RoboCup 2003 — Robot Soccer World Cup VII, Padua, Italy*, pages 214–225. Springer-Verlag, 2003, ISBN: 3-5402-2443-2.

[72] A. Kak and N. DeSouza. Robotic vision: What happened to the visions of yesterday? In *Proceedings of the 16th International Conference on Pattern Recognition*, pages 839–847. IEEE Computer Society Press, 2002, ISBN: 0-7695-1695-X.

[73] C. Kotropoulos, A. Tefas, and I. Pitas. Frontal face authentication using morphological elastic graph matching. *IEEE Transactions on Image Processing*, 9(4):555–560, 2000, ISSN: 1057-7149.

[74] B. Lahme and R. Miranda. Karhunen-Loève decomposition in the presence of symmetry — part I. *IEEE Transactions on Image Processing*, 8(9):1183–1190, 1999, ISSN: 1057-7149.

[75] C. Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design*. Prentice Hall, U.S.A., 1998, ISBN: 0-1374-8880-7.

[76] R. Larsen. 3-D contextual bayesian classifiers. *IEEE Transactions on Image Processing*, 9(3):518–524, 2000, ISSN: 1057-7149.

[77] R. Larsen and M. Marx. *An Introduction to Mathematical Statistics and its Applications*. Prentice Hall, U.S.A., 2003, ISBN: 0-1392-2303-7.

[78] Y. LeCun, F. Huang, and L. Bottou. Learning methods for generic object recognition with invariance to pose and lighting. In *Proceedings of the 2004 International Conference on Computer Vision and Pattern Recognition*, pages 97–104. IEEE Computer Society Press, 2004, ISBN: 0-7695-2158-4.

[79] D. Lee. Medial axis transformation of a planar shape. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 4(4):363–369, 1982, ISSN: 0162-8828.

[80] R. Lenz. Estimation of illumination characteristics. *IEEE Transactions on Image Processing*, 10(7):1031–1038, 2001, ISSN: 1057-7149.

[81] M. Lindenbaum and A.M. Bruckstein. On recursive, $O(n)$ partitioning of a digitized curve into digital straight segments. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(9):949–953, 1993, ISSN: 0162-8828.

[82] M. Lotzsch, J. Bach, H. Burkhard, and M. Jungel. Designing agent behaviour with the extensible agent behaviour specification language xabsl. In *Proceedings of RoboCup 2004 — Robot Soccer World Cup VIII, Lisbon, Portugal*, pages 114–124. Springer-Verlag, 2004, ISBN: 3-5402-5046-8.

[83] V. Hlavac M. Sonka and R. Boyle. *Image Processing, Analysis, and Machine Vision*. PWS Publishing, U.K., 1998, ISBN: 0-5349-5393-X.

[84] W. Ma and B. Manjunath. EdgeFlow: A technique for boundary detection and image segmentation. *IEEE Transactions on Image Processing*, 9(8):1375–1388, 2000, ISSN: 1057-7149.

[85] C. Madden and R. Mahony. An ordered list approach to real-time line detection based on the Hough transform. In *Proceedings of the 2003 Australasian Conference on Robotics and Automation*, CD-Rom Proceedings, 2003, ISBN: 0-9587-5835-2.

[86] S. Manay and A. Yezzi. Anti-geometric diffusion for adaptive thresholding and fast segmentation. *IEEE Transactions on Image Processing*, 12(11):1310–1323, 2003, ISSN: 1057-7149.

[87] J. Markusek and A. Vitko. Unified simulation environment for learning navigation of a robot operating in unknown terrain. In *Proceedings of the 4th International Conference on Climbing and Walking Robots*, pages 435–441. Professional Engineering Publishing, 2001, ISBN: 1-86058-365-2.

[88] A. Marshall and R. Martin. *Computer Vision, Models and Inspection.* Barnes and Noble, U.S.A., 1992, ISBN: 9-8102-0772-7.

[89] M. Mataric and D. Cliff. Challenges in evolving controllers for physical robots. Technical Report CS-95-184, Brandeis University and University of Sussex, U.K., 1995.

[90] G. Mayer, H. Utz, and G. Kraetzschmar. Towards autonomous vision self calibration. In *Proceedings of the International Conference on Intelligent Robots and Systems*, pages 214–219. IEEE Computer Society Press, 2002, ISBN: 0-7803-7545-9.

[91] G. Mayer, H. Utz, and G. Kraetzschmar. Playing robot soccer under natural light. In *Proceedings of RoboCup 2003 — Robot Soccer World Cup VII, Padua, Italy*, pages 238–249. Springer-Verlag, 2003, ISBN: 3-5402-2443-2.

[92] G. Medioni and A. Francois. 3-D structures for generic object recognition. In *Proceedings of the 15th International Conference on Pattern Recognition*, pages 30–37. IEEE Computer Society Press, 2000, ISBN: 0-7695-0750-6.

[93] M. Meng and A. Kak. Mobile robot navigation using neural networks and nonmetrical environment models. *IEEE Control Systems*, 13(6):30–39, 1993, ISSN: 0272-1708.

[94] H. Moravec. Robot spatial perception by stereoscopic vision and 3d evidence grids. Technical Report CMU-RI-TR-96-34, Carnegie Mellon University, U.S.A., 1996.

[95] S. Nayar, H. Murase, and S. Nene. Parametric appearance representation. In S. Nayar, T. Poggio, and P. Nayar, editors, *Early Visual Learning*. Oxford University Press, U.K., 1996, ISBN: 0-1950-9522-7.

[96] Y. Ogihara, Y. Shibata, H. Najima, K. Kii, K. Oda, and T. Ohashi. Asura: The kyushu united team 2005 in the four legged robot league. In *Proceedings of RoboCup 2005 — Robot Soccer World Cup IX, Osaka, Japan*, CD-Rom Proceedings. Springer-Verlag, 2005, *To appear*.

[97] R. Ogniewicz and O. Kübler. Hierarchic Voronoi skeletons. *Pattern Recognition*, 28(3):343–359, 1995, ISSN: 0031-3203.

[98] J. O'Rourke. *Computational Geometry in C*. Cambridge University Press, U.K., 1998, ISBN: 0-5216-4010-5.

[99] T. Pajdla and J. Matas. Feature-based object detection and recognition ii: Weak hypotheses and boosting for generic object detection and recognition. In *Proceedings of the 8th European Conference on Computer Vision*, pages 71–84. Springer-Verlag, 2004, ISBN: 3-540-23989-8.

[100] G. Paschos. Perceptually uniform color spaces for color texture analysis: An empirical evaluation. *IEEE Transactions on Image Processing*, 10(6):932–937, 2001, ISSN: 1057-7149.

[101] L. Pau. *Computer Vision for Electronics Manufacturing*. Plenum Press, U.S.A., 1990, ISBN: 0-3064-3182-3.

[102] P. Perona and J. Malik. Scale-space and edge detection using anisotropic diffusion. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(7):629–639, 1990, ISSN: 0162-8828.

[103] C. Privitera and L. Stark. Human-vision-based selection of image processing algorithms for planetary exploration. *IEEE Transactions on Image Processing*, 12(8):917–923, 2003, ISSN: 1057-7149.

[104] M. Quinlan, S. Chalup, and R. Middleton. Application of SVMs for colour classification and collision detection with aibo robots. In *Proceedings of the Advances in Neural Information Processing Systems Conference*, pages 635–642. MIT Press, 2003, ISBN: 0-262-20152-6.

[105] M. Quinlan, S. Chalup, and R. Middleton. Techniques for improving vision and locomotion on the sony aibo robot. In *Proceedings of the 2003 Australasian Conference on Robotics and Automation*, CD-Rom Proceedings, 2003, ISBN: 0-9587-5835-2.

[106] T. Reed and J. Hans du Buf. A review of recent texture segmentation and feature extraction techniques. *CVGIP: Image Understanding archive*, 57(3):359–372, 1993, ISSN: 1049-9660.

[107] T. Röfer, R. Brunn, S. Czarnetzki, M. Dassler, M. Hebbel, M. Jungel, T. Kerkhof, W. Nistico, T. Oberlies, C. Rohde, M. Spranger, and C. Zarges. GermanTeam 2005. In *Proceedings of RoboCup 2005 — Robot Soccer World Cup IX, Osaka, Japan*. Springer-Verlag, 2005, *To appear*.

[108] T. Röfer, I. Dahm, U. Duffert, J. Hoffmann, M. Jungel abd M. Kallnik, M. Lotzsch, M. Risler, M. Stelzer, and J. Ziegler. Germanteam 2003. In *Proceedings of RoboCup 2003 — Robot Soccer World Cup VII, Padua, Italy*, CD-Rom Proceedings. Springer-Verlag, 2003, ISBN: 3-5402-2443-2.

[109] T. Röfer and M. Jungel. Vision-based fast and reactive Monte-Carlo localization. In *Proceedings of the 2003 IEEE International Conference on Robotics and Automation*, pages 856–861. IEEE Computer Society Press, 2003, ISBN: 0-7803-7736-2.

[110] A. Roy and J. Stell. A qualitative account of discrete space. In *Proceedings of the 2nd International Conference on Proceedings of Geographic Information Science*, pages 276–290. Springer-Verlag, 2002, ISBN: 3-540-44253-7.

[111] G. Schuster and A. Katsaggelos. An optimal polygonal boundary encoding scheme in the rate distortion sense. *IEEE Transactions on Image Processing*, 7(1):13–26, 1998, ISSN: 1057-7149.

[112] F. Sensini, G. Buttazzo, and P. Ancilotti. Ghost: A tool for simulation and analysis of real-time scheduling algorithms. In *Proceedings of the 2nd IEEE*

Real-Time Education Worksop, pages 110–123. IEEE Computer Society Press, 1997, ISBN: 0-8186-8256-6.

[113] J. Shaik and K. Iftekharuddin. Automated tracking and classification of infrared images. In *Proceedings of the 2003 International Joint Conference on Neural Networks*, pages 1201–1206. IEEE Computer Society Press, 2003, ISBN: 0-7803-7899-7.

[114] C. Shaw, M. Green, J. Liang, and Y. Sun. Decoupled simulation in virtual reality with the MR toolkit. *ACM Transactions on Information Systems*, 11(3):287–317, 1993, ISSN: 1046-8188.

[115] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, U.S.A., 1996, ISBN: 0-1318-2957-2.

[116] B. Shepherd. An appraisal of a decision tree approach to image classification. In *Proceedings of the 8th International Joint Conference on Artificial Intelligence*, pages 496–501. Morgan Kaufmann, 1983, ISBN: 0-8657-6064-0.

[117] S. Shimizu, T. Nagahashi, and H. Fujiyoshi. Robust and accurate detection of object orientation and id without color segmentation. In *Proceedings of RoboCup 2005 — Robot Soccer World Cup IX, Osaka, Japan*. Springer-Verlag, 2005, *To appear*.

[118] D. Sim and R. Park. Two-dimensional object alignment based on the robust oriented hausdorff similarity measure. *IEEE Transactions on Image Processing*, 10(3):475–489, 2001, ISSN: 1057-7149.

[119] I. Sobel. *Camera Models and Machine Perception*. PhD thesis, Stanford University, U.S.A., 1970.

[120] N. Sochen, R. Kimmel, and R. Malladi. A general framework for low level vision. *IEEE Transactions on Image Processing*, 7(3):310–318, 1998, ISSN: 1057-7149.

[121] V. Srinivasan. Edge detection using neural networks. *Pattern Recognition*, 27(12):1653–1662, 1994, ISSN: 0031-3203.

[122] C. Su and B. Wu. A low memory zerotree coding for arbitrarily shaped objects. *IEEE Transactions on Image Processing*, 12(3):271–282, 2003, ISSN: 1057-7149.

[123] P. Suetens, P. Fua, and A. Hanson. Computational strategies for object recognition. *ACM Computer Survey*, 24(1):5–62, 1992, ISSN: 0360-0300.

[124] S. Thrun. A framework for programming embedded systems: initial design and results. Technical Report CMU-CS-98-142, Carnegie Mellon University, U.S.A., 1998.

[125] O. Tobias and R. Seara. Image segmentation by histogram thresholding using fuzzy sets. *IEEE Transactions on Image Processing*, 11(12):1457–1465, 2002, ISSN: 1057-7149.

[126] D. Travis. *Effective Color Displays. Theory and Practice*. Academic Press, U.S.A., 1991, ISBN: 0-1269-7690-2.

[127] A. Tremeau and P. Colantoni. Regions adjacency graph applied to color image segmentation. *IEEE Transactions on Image Processing*, 9(4):735–744, 2000, ISSN: 1057-7149.

[128] H. van Assen, M. Egmont-Peterson, and J. Reiber. Accurate object localization in gray level images using the center of gravity measure: Accuracy vs precision. *IEEE Transactions on Image Processing*, 11(12):1379–1384, 2002, ISSN: 1057-7149.

[129] C. Veenman, M. Reinders, and E. Backer. A cellular coevolutionary algorithm for image segmentation. *IEEE Transactions on Image Processing*, 12(3):304–316, 2003, ISSN: 1057-7149.

[130] M. Veloso, S. Chernova, C. McMillen, P. Rybski, J. Fasola, F. vonHundelshausen, A. Trevor, S. Hauert, and R. Espinoza. Cmdash05: Team description paper. In *Proceedings of RoboCup 2005 — Robot Soccer World Cup IX, Osaka, Japan*, CD-Rom Proceedings. Springer-Verlag, 2005, *To appear*.

[131] M. Veloso, W. Uther, M. Fujita, M. Asada, and H. Kitano. Playing soccer with legged robots. In *Proceedings of the International Conference on*

Intelligent Robots and Systems, pages 437–442. IEEE Computer Society Press, 1998, ISBN: 0-7803-4465-0.

[132] A. Verikas, K. Malmquist, and L. Bergman. Colour image segmentation by modular neural network. *Pattern Recognition Letters Archive*, 18(2):173–185, 1997, ISSN: 0167-8655.

[133] S. Wan. Symmetric region growing. *IEEE Transactions on Image Processing*, 12(9):1007–1015, 2003, ISSN: 1057-7149.

[134] H. Wang, G. Schuster, A. Katsaggelos, and T. Pappas. An efficient rate-distortion optimal shape coding approach utilizing a skeleton-based decomposition. *IEEE Transactions on Image Processing*, 12(10):1181–1193, 2003, ISSN: 1057-7149.

[135] R. Wilcox. *Introduction to Robust Estimation and Hypothesis Testing.* Academic Press, U.S.A., 1997, ISBN: 0-1275-1542-9.

[136] D. Wilking and T. Röfer. Realtime object recognition using decision tree learning. In *Proceedings of RoboCup 2004 — Robot Soccer World Cup VIII, Lisbon, Portugal*, pages 556–563. Springer-Verlag, 2004, ISBN: 3-5402-5046-8.

[137] P. Winston and B. Horn. *Artificial Intelligence.* Addison-Wesley, U.S.A., 1992, ISBN: 0-2015-3377-4.

[138] I. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations.* Morgan Kaufmann, U.S.A., 1999, ISBN: 1-5586-0552-5.

[139] I. Witten and E. Frank. *Data Mining — Practical Machine Learning Tools and Technologies with Java Implementations.* Morgan Kaufmann, U.S.A., 2000, ISBN: 1-5586-0552-5.

[140] M. Worboys. *Geographical Information Systems: A Computing Perspective.* Taylor & Francis, U.K., 1995, ISBN: 0-7484-0064-8/0-7484-0065-6.

[141] L. Wu. On the chain code of a line. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 4(3):347–353, 1982, ISSN: 0162-8828.

[142] C. Zhang and F. Cohen. 3-D face structure extraction and recognition from images using 3-D morphing and distance mapping. *IEEE Transactions on Image Processing*, 11(11):1249–1259, 2002, ISSN: 1057-7149.

[143] M. Zhang and J. Fulcher. Face recognition using artificial neural network group-based adaptive tolerance (GAT) trees. *IEEE Transactions on Neural Networks*, 7(3):555–567, 1996, ISSN: 1045-9227.