

# Three Dimensional Human Face Acquisition for Recognition

---

*A dissertation submitted to the University of Cambridge as a requirement for the degree of  
Doctor of Philosophy*

Adam D. Tibbalds, Trinity College

March 1998

---



SIGNAL PROCESSING AND COMMUNICATIONS LABORATORY  
Department of Engineering  
University of Cambridge

To those who should still be here today...

## Declaration

---

The research described in this dissertation was carried out by the author between October 1993 and September 1997. Except as indicated in the text, the contents are original and not the result of work done in collaboration. No part of this dissertation has been submitted to any other University. The length of this dissertation does not exceed 34 000 words and contains 74 figures.

## Acknowledgements

---

I would like to acknowledge the contribution of my supervisor Dr Nick Kingsbury to this research project. It was he who first suggested the idea, arranged the sponsorship for it and has been the source of much guidance to it throughout its course.

My very grateful thanks must also go to Dr Anil Kokaram for his consistent help throughout the past few years. He was always willing to discuss ideas and help with problems that I was having and without him this project could have taken substantially longer. Dr Joan Lasenby played a very similar role in helping me throughout the project, particularly whilst commenting: 'I haven't really thought about it, but isn't something like this?' - and of course, it was. Many other members of the lab have also been of great help in a variety of ways including the proof-readers Simon Barker and Anil (again).

The smooth running of the lab network, in spite of many upheavals (two moves and numerous operating system changes), was achieved through the efforts of many people, including Anil Kokaram, Pete Wilson, Alastair Cain and myself (when I was inadvertently ensnared!)

Finally I would like to express my heartfelt thanks to my parents who encouraged my education at every possible opportunity and who did more than anyone else to set in motion the events that led both myself and my brother to Cambridge.

This research was funded by a grant from the Engineering and Physical Sciences Research Council and by a CASE award from Cambridge Neurodynamics Limited.

---

## Summary

---

Machine identification and recognition of human faces is a rapidly growing research area in both the academic and commercial world.

Most of the research to date has concentrated on the use of two dimensional information, acquired from video cameras or photographs. The use of a three dimensional system is hoped to remove many of the problems affecting the two dimensional systems such as disruption caused by changes in the face's orientation or changes in the ambient lighting. A three dimensional system will obviously not be influenced by orientation changes and the lighting is irrelevant, as it is the shape not the shading of the face that is important.

For this system to be of practical use it is important that the process of acquiring the necessary information to generate the three dimensional surface model should not require any complex or expensive equipment and should not impose any undue constraints on the human subject, such as a requirement to remain still for an extended period of time.

The described system uses a projected structured light, stereo vision arrangement to acquire the surface shape. Traditionally these systems have either been limited to simple surfaces or have required multiple views for more complex objects to prevent ambiguity problems in the stereo matching phase. This dissertation describes a novel method, loosely based on graph theory, to resolve this problem and thus requires only a single stereo image.

The individual stages in the extraction of the surface are discussed and demonstrated: (i) the capture of the necessary stereo view (ii) the recovery of the features in the camera image (iii) the stereo matching of these features to the projected pattern using the new graph based technique, and (iv) the creation of the surface model.

This dissertation concludes with an evaluation of the success of the system and suggests possible areas of further work including the steps needed to produce a successful recognition or identification system.

**Keywords**

---

The following keywords may be useful for indexing purposes:

Face recognition, face identification, stereo vision, structured light stereo vision, stereo correspondence, three dimensional surface acquisition.

# Contents

---

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.0.1	Identification . . . . .	2
1.0.2	Recognition . . . . .	2
1.0.3	Implementation . . . . .	3
1.1	Current Face Recognition/Identification Systems . . . . .	3
1.1.1	2D Systems . . . . .	3
1.1.2	3D Systems . . . . .	5
1.1.3	Bibliography . . . . .	7
1.2	This Project . . . . .	8
1.3	Synopsis of Dissertation . . . . .	9
<b>2</b>	<b>Stereo Vision Techniques</b>	<b>11</b>
2.1	Introduction . . . . .	11
2.2	Stereo Vision Issues . . . . .	11
2.2.1	Correspondence . . . . .	11
2.2.2	Calibration . . . . .	14
2.2.3	Disparity . . . . .	14
<b>3</b>	<b>Surface Extraction</b>	<b>17</b>
3.1	Basic System Type . . . . .	17
3.2	Projected Feature Choice . . . . .	17
3.2.1	Feature Shape . . . . .	17
3.2.2	Feature Density . . . . .	18
3.2.3	Multiple Images . . . . .	19
3.2.4	Feature Ambiguity Resolution . . . . .	20
3.3	Problem Statement . . . . .	20

---

3.4	System Summary . . . . .	20
3.4.1	The Physical Equipment Setup . . . . .	20
3.4.2	Image Processing . . . . .	22
<b>4</b>	<b>Image Filtering and Edge Extraction</b>	<b>24</b>
4.1	Image Edge Detection . . . . .	24
4.2	Filtering . . . . .	25
4.2.1	Filter Implementation . . . . .	25
4.3	Zero Crossing Extraction from filtered images . . . . .	29
4.3.1	Zero Crossing Line Location . . . . .	29
4.3.2	Data Objects . . . . .	31
4.3.3	Zero Crossing Line Selection . . . . .	33
4.4	Zero Crossing Line Set Combination . . . . .	34
4.4.1	Zero Crossing Line Set Searching . . . . .	36
4.5	Conclusions . . . . .	40
<b>5</b>	<b>Establishing Stereo Correspondence</b>	<b>42</b>
5.1	The Adjacency Graph . . . . .	43
5.2	Graph Implementation . . . . .	44
5.2.1	The Node Data Object . . . . .	46
5.2.2	The Node Comparison Operators . . . . .	47
5.2.3	Graph Creation . . . . .	50
5.2.4	Node Insertion . . . . .	50
5.2.5	Node Removal . . . . .	53
5.2.6	Graph Display . . . . .	55
5.2.7	Graph Shape . . . . .	55
5.2.8	Graph Node Shuffling . . . . .	57
5.2.9	Graph Collapsing . . . . .	59
5.2.10	Graph Processing Output . . . . .	66
5.2.11	Algorithm Implementation and Debugging . . . . .	66
5.3	Conclusions . . . . .	67
<b>6</b>	<b>Surface Generation</b>	<b>70</b>
6.1	Surface Creation . . . . .	70
6.1.1	The Surface Data Object . . . . .	72

---

6.1.2	Surface Filling . . . . .	72
6.1.3	Erroneous Surface Regions . . . . .	74
6.1.4	Surface Curvature . . . . .	75
6.2	Surface Display . . . . .	77
6.2.1	Testing . . . . .	78
6.3	Surface Combination . . . . .	81
6.4	Conclusions . . . . .	82
<b>7</b>	<b>Conclusions</b>	<b>84</b>
7.1	Introduction . . . . .	84
7.1.1	Results . . . . .	85
7.1.2	Program Execution Time . . . . .	90
7.2	Classification . . . . .	91
7.3	Further Work . . . . .	93
<b>A</b>	<b>Glossary</b>	<b>96</b>
<b>B</b>	<b>Software Routines</b>	<b>97</b>
B.1	Introduction . . . . .	97
B.2	The C++ program structure . . . . .	98
B.2.1	<code>filtdef</code> - Filter definitions . . . . .	99
B.2.2	<code>imagedef</code> - Image definitions . . . . .	100
B.2.3	<code>linedefs</code> - Line definitions . . . . .	100
B.2.4	<code>leveldef</code> - Level (line set) definitions . . . . .	101
B.2.5	<code>dlinkdef</code> - Double Linked List definitions . . . . .	103
B.2.6	<code>glinedef</code> - A line grouping definition . . . . .	105
B.2.7	<code>mlinkdef</code> - The Multiply Linked List . . . . .	106
B.2.8	<code>graphdef</code> - the graph manipulation object . . . . .	114
B.2.9	<code>surfdefs</code> - surface generation and display . . . . .	115
B.2.10	<code>facescan</code> - the main program . . . . .	117
B.2.11	Numerical Output . . . . .	118
B.2.12	Technical details . . . . .	119
	<b>Bibliography</b>	<b>121</b>

# Introduction

---

Identification or recognition <sup>1</sup> of human faces is seen as a potentially useful tool in a wide area of applications. It forms one part of the overall research area referred to as Biometric Systems, all of which attempt to reliably measure some part of the human subject, such as their eyes, face, hands or even voice and use this to uniquely identify that person. Typically these include both cooperative and non-cooperative situations such as building entry control and covert surveillance (such as airport security), respectively.

The suitability of a recognition or identification system is often measured in terms of the false-acceptance and false-rejection rates. These will commonly be varied according to the actual application and the demands of the operator. For example, High Street Banks tend to opt for a low false-rejection rate (which obviously implies a higher false-acceptance rate) on the grounds of minimising customer inconvenience, whereas military applications will tend to favour a very low false-acceptance rate at the expense of a higher false-rejection rate.

Acceptability is often quoted as the reason for preferring a more complicated and less reliable system, such as face recognition, as opposed to a simpler and more reliable system, such as fingerprint identification. Despite the extreme difficulty of the face recognition task, the ‘man in the street’ understands what it means, as it is a task that he can perform very easily (indeed, there is much ongoing physiological research in this area). Fingerprinting and iris recognition, for example, are two commercially available biometric identification systems, which tend to be viewed distrustfully by the general public as these non-understandable systems can seem to have an element of ‘Big Brother’ about them, a term somewhat presciently coined by George Orwell in ‘Nineteen Eighty-Four’ [35].

Many magazines and journals have had special editions dealing with the issues of biometric research, such as the IEEE Spectrum journal [31], with commercial interest

---

<sup>1</sup>Identification being confirmation of a claimed identity; recognition being confirmation without prior claim

being demonstrated in such professional magazines as *Advanced Imaging* [1]. The problem has been considered for many years prior to the more recent advances in this field, as exemplified by an article in the *Proceedings of the IEEE* in 1971 in which Goldstein, et al [17] performed a statistical study using human ‘jurors’ to analyse photographs of 256 test subjects. They concluded that about 6 features of the face are needed to isolate a subject within a population of 256 and that about 14 would be needed to isolate a subject within a population of 4 million.

In general, a person identification system may make use of more than one type of feature, exemplified by the combined face and voice identification schemes discussed by Brunelli, et al [6]

### 1.0.1 Identification

A typical identification scenario would be the use of a ‘hole in the wall’ cash machine. The subject inserts a card into the machine and thus claims the identity of that account holder. The subject enters a Personal Identification Number as a confirmation of this identity, although recent surveys suggest that often many other people also know this number. However, this step means that a fraudulent user has to do more than simply steal the card. It is proposed that even a simple face identification system would then minimise fraudulent use even further, as the subject has to look like the account holder. Thus three steps of identification have been achieved i) possession of a physical object - the card ii) possession of a piece of information - the PIN and iii) possession of a certain set of physical attributes. Such a system will obviously be more secure than a system relying only on the first two steps.

### 1.0.2 Recognition

A typical recognition scenario would be in Airport Security where human security personnel watch passengers from behind one-way glass in an attempt to recognise certain individuals. The performance of such personnel is limited by the length of time that a sufficiently high level of concentration can be maintained and by the number of individuals that they can watch for. Once again, even a simple face recognition system could be superior to such individuals as the level of performance is unaffected by time and the total number of subject individuals could be far higher.

### 1.0.3 Implementation

The implementation of recognition systems typically can be split into several stages. The initial step is to acquire the necessary information from the individuals who are to be identified or recognised. In the case of identification systems, this is likely to be easily done as the individual will be cooperating with this process. In the case of recognition systems, this information will have to be collected from as many alternative sources as are available.

This information will be stored in a database of biometric details. When the identification or recognition of an individual is required later on, the individual would be scanned and the physical data converted into the same form as that of the database. For identification, the appropriate record is extracted from the database and compared with the subject's data, whereas with recognition systems, the entire database has to be searched for any positive matches to the subject. It is likely, therefore, that an identification system would be more successful in practice as an identity is being confirmed rather than deduced.

## 1.1 CURRENT FACE RECOGNITION/IDENTIFICATION SYSTEMS

The current research is broadly divided into that which uses two dimensional information and that which uses three dimensional information. An excellent survey of both these areas was published in the Proceedings of the IEEE [12] and surveys on more specific types of face processing have also been published, such as the survey of connectionist models of face processing by Valentin, et al [45]

### 1.1.1 2D Systems

Most of the current research in face recognition is based on two dimensional information. This is probably entirely due to the ease with which such information can be acquired. For computer vision purposes, the most likely sources are video cameras, but information can also be gathered from such sources as surveillance video tapes or photographs.

There are a variety of different approaches to deal with the 2D information which can

broadly be divided into two strategies, template matching and feature matching. Many articles have been published comparing these two strategies such as Brunelli and Poggio [7] and Robertson and Craw [40].

### *Feature Matching*

Features are extracted from the image of the face and the size, shape and location of these features form the set of information that is used to search the library database of individuals. Such features may be things like eye shape or colour, or the location of the eyes relative to the corners of the mouth. These systems tend to only have a small amount of information stored about each individual, which makes them more efficient in terms of minimising database size and search time. Many examples of this type of system have been described in the literature such as the work on eigenfaces by Turk and Pentland [43], face profiles by Najman, Vaillant and Pernot [32] and inter-feature distances by Kamel, et al [24].

### *Template Matching*

The entire image of the face is compared with a series of templates based on the stored library information. There are many variations on this basic idea, such as only using small templates of the eyes and mouth, or using broad templates to group faces into similar types (interestingly enough, these types, although automatically selected by the system, tend to produce templates that match faces into racial and gender groups). A very simple scheme would be to use the library information to create a ‘matched filter’ for each person, each of which would be used to filter the subject image to check for a match. A significantly more complex scheme is described by Craw, et al [14], who have also done work on the automated indexing of police ‘mug shots’ using a polygonal mesh template. A similar ‘active shape’ model has been described by Lanitis, et al [25] [26] in which line templates and gray scale templates of the face are fitted to the supplied images and the parameters of these templates are used to reconstruct or identify the individual faces.

### *Problems*

The major problems that affect 2D face recognition/identification systems are mainly those of orientation and lighting. As most of the research is based on greyscale images (although more recently some work has been done on colour images) changes in the

lighting of the subject can result in large brightness, contrast and shading changes as seen in the scanned image. These changes cause significant problems in the development of these systems. The changing orientation of the head can also cause problems by changing the shape and dimensions of the face as seen in the scanned image. For example if, in a feature matching system, the eyes are measured as 2cm apart, then that measurement will only be accurate if the subject was facing straight at the camera. If the head was turned slightly to one side, then any measurement will be too small, thus supplying incorrect information to the identification/recognition process. This has inspired some research into geometrically invariant feature systems [24] without resorting to three dimensional systems.

### 1.1.2 3D Systems

The main advantages of using 3D information about the shape of the subject face, are to overcome the lighting and orientation problems that affect 2D systems. Assuming that it is possible to acquire such data, it will obviously be totally independent of the lighting on the face and the shape of the face will be unchanged by its orientation. In this way the shape of the subject face can be compared with the database faces using template or feature techniques, without the subject data being compromised by lighting or orientation changes.

#### *Scanning Systems*

The immediate disadvantage of such systems is the increased complexity of the equipment needed to acquire the information. The work independently done by Bruce [5], Gordon [20] [19] [18] and Chang, et al [11] requires the use of sophisticated (and expensive) laser scanning systems to acquire the necessary 3D data, although the results are correspondingly impressive. In the first two cases the surface data is transformed into curvature maps, whereas the work of Chang, et al uses the surface normals - all techniques requiring accurate, noise free data. Commercial scanning services are available from many companies such as Cyberware Inc [23] and 3D Scanners Ltd [28] who described one of their systems at an IEE Symposium on '3D Imaging and analysis of depth/range images' [10].

*Structured Light Systems*

An alternative method uses principles of stereo vision to acquire the necessary data. This is discussed at greater length in the next chapter. The advantages of this method are that it only need require cameras and some kind of projection system. In many cases this work has concentrated on the analysis of simple objects rather than complex ones such as human faces. Examples of this are the use of structured light patterns to analyse mechanical parts such as the active projection systems described by Nurre, et al [34] and McDonald, et al [30]. Work on tracking simple moving surfaces with a structured light projection system has also been done by Lindsey and Blake [27] in which two camera views of a changing surface with a projected pattern are decoded into a changing surface model. Another example from Van Gool, et al [37] uses a projected grid pattern and has been applied to human faces as part of the ACTS project “VANGUARD”.

*Stereo Vision Systems*

Systems that attempt to extract full 3D information from two (or more) views of an object do exist but are virtually all confined to objects which will generate a sufficient number of image features to allow for conclusive stereo matching. In the case of trying to establish the shape of a reasonably smooth object, such as a human face, these systems would not be able to generate an accurate surface shape. Two such systems are described by Ayache, et al [2] and Pollard, et al [36]. A two camera technique applied to human faces has been described by Urquhart [44] although his technique requires smooth surfaces to be ‘roughed up’ by the projection of a textured pattern onto the face.

*Other Techniques*

Other novel techniques for determining three dimensional shape include a range of systems based on ‘reverse rendering’ whereby assumptions and deductions about the lighting on and the material properties of an object are used to ‘reverse render’ the intensity image to determine its shape. These techniques are commonly referred to as ‘shape from shading’ systems, exemplified by Nakamura [33], Zheng [46] and Hougen, et al [22], and determine surface shape from the grey level contours in the supplied image.

*Medical Systems*

Particularly in the fields of plastic and reconstructive surgery, there is much interest in systems that would be able to record the shape of various parts of a patient during a process of treatment or surgery. Examples of these use a variety of the techniques described above but all benefit from the ability to scan the patient under highly controlled conditions, such as being able to clamp the head still for a scan of prolonged duration or dealing with anaesthetised individuals.

Systems using structured light or laser scanning for the study of human faces have been described by Gregory, et al [21], Bhatia, et al [3] and Bush, et al [8]. The system described by Gregory used a projected grid pattern and multiple CCD camera imaging, Bhatia describes a system using six pairs of pattern projectors and digital cameras and the system described by Bush used a Cyberware scanner.

A full body colour scanning system was described by Rioux [39] which uses red, green and blue light sources together with a motion platform to allow detailed scanning from fingertip size to full body.

*Problems*

The problems facing these systems in a practical situation are fairly clear. Mechanical scanning systems are complex, expensive and require a significant time to operate compared to the electronic scan of a modern CCD camera. Structured light systems (as described in the next chapter) require a projection system and can have difficulty with resolving the shape of the pattern in the camera image. Stereo vision systems (as described in the next chapter) are generally unsuited to smooth objects when a reasonably accurate surface shape is required, due to the lack of image features for matching.

**1.1.3 Bibliography**

As there is much work, both past and present, on 2D face recognition/identification, there is an ample range of publications referring to it. This section has attempted to present the main research areas and refer to some typical publications in each area particularly papers that present summaries of other work[12]. It was not felt necessary to include any more references than this as the 2D field is not directly related to this research.

However, in 3D research the selection of suitable publications is not so clear. Much of the stereo view work is based on corner or edge detection techniques and is, therefore,

unsuitable for use on the relatively smooth features of the face. Most of the structured light work seems to be based on highly smooth objects to prevent ambiguities, or uses other techniques such as multiple camera views or a coarse pattern imaged at different positions over multiple frames to resolve such ambiguity problems. The extreme end of this research uses mechanical scanning systems or is used in such a specialised field (e.g. surgery) that other constraints can be applied to allow the successful acquisition of a detailed surface map. In summary, not many references have been found which appear to be discussing a similar problem to that addressed in this dissertation, hence the shorter than usual bibliography of material in this area. This is exemplified in the IEEE Proceedings survey paper[12] in which only 2 papers were cited (out of nearly 150) that directly addressed the issue of face recognition using three dimensional data - both were by Gordon[19][18].

## 1.2 THIS PROJECT

On the basis of the previous discussion of potential systems, it was decided to attempt a realistically practical system based on the use of three dimensional information. The use of three dimensional information would remove the effects of lighting and orientational changes but it was also felt important that the system should not require any complex or unusual equipment. As stated in the IEEE Proceedings Survey of Face Recognition [12] *'Owing to the cost of the equipment and the need for easy maintenance, intensity based systems are preferred in law enforcement applications. Although range information is richer than the 2D intensity array, we feel that cost considerations will make range image based techniques less attractive for field use.'* As these comments apply equally well to other applications, and the original intent of this project was to realise a practical system, the simplicity of the equipment used in this system is paramount.

This implies that such a system is likely to be based on some kind of stereo imaging system, as described above, as the use of two video cameras or a video camera and simple projector would not qualify as complex or expensive, even in comparison with a 2D system only using one video camera.

---

## 1.3 SYNOPSIS OF DISSERTATION

**Chapter 1** presents the background to the project and a review of the various techniques that others have pursued in the fields of two and three dimensional face recognition/identification and stereo vision using both structured light and multiple views.

**Chapter 2** presents a discussion of the factors influencing the design of a stereo vision system - in particular the pros and cons of a two camera system versus a structured light system. This is influenced by the issues of stereo feature correspondence, calibration of the system and data accuracy versus camera disparity.

**Chapter 3** discusses the choice of projected feature with respect to its nature and density, the use of multiple camera images and the issue of feature ambiguity. The chapter concludes with a brief description of the proposed system to extract three dimensional surface information from the single acquired camera image.

**Chapter 4** describes in detail the image processing and filtering techniques used to extract the projected features from the acquired camera image. Firstly, the use of difference of Gaussian filters in a heirarchical/multiscale scheme to generate zero contours along possible image feature locations and an efficient manner in which to perform the processing is presented. Secondly, the processing of the zero contours from each scale and their amalgamation into a set of image features is shown.

**Chapter 5** describes a new technique to establish the correspondence between the features seen in the image and the features that were actually projected by the structured light source. This process was developed to overcome the feature ambiguity problem and to establish the stereo correspondence. It uses an approach based on graph theory to iteratively sort the image features into groups, thus matching them with the projected feature that created them. It is the work described in this chapter that is the key to the success of this system.

**Chapter 6** presents the method for generating the 3D surface from the list of feature correspondences presented to it. The combination of surfaces from multiple camera views is discussed. The display of the generated surfaces is also presented.

**Chapter 7** contains several sets of example results and compares the approach of this system to other similar systems. The dissertation ends with some concluding remarks and suggestions for possible further work.

**Appendix A** summarises the use of certain terms throughout chapters 4 to 6.

**Appendix B** presents a summary of the software routines used to develop this system.

It may be useful to refer to this in conjunction with the main text to ascertain exactly how a particular process was performed. Where appropriate, references have been included in the main text to facilitate this. It also highlights the manner in which a highly complex task could be broken down into a number of key routines and data objects.

## 2.1 INTRODUCTION

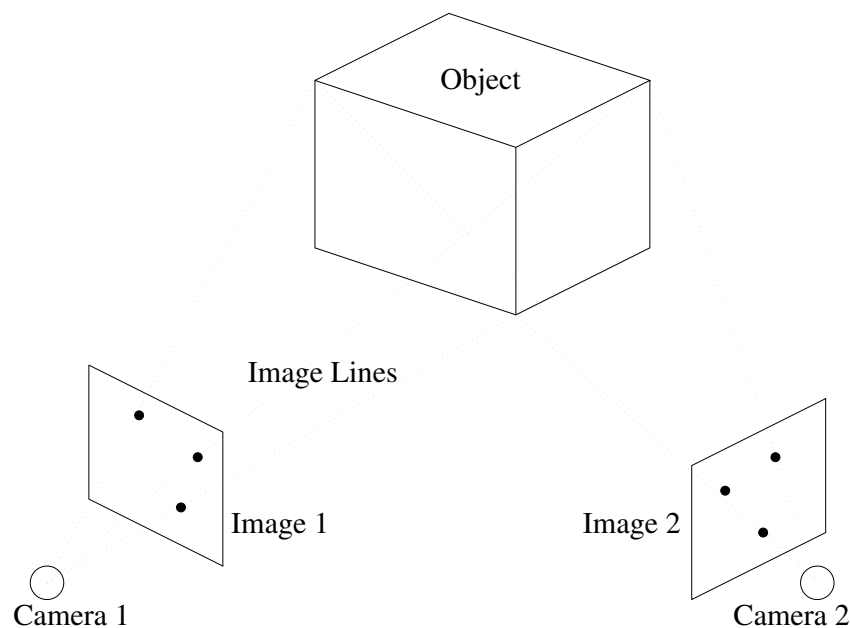
The term ‘Stereo Vision’, in the context of computer vision, is used to describe any situation in which two or more implied views are used to attempt to recover information about the three dimensional shape of the objects being viewed.

## 2.2 STEREO VISION ISSUES

There are three main issues in the field of stereo vision research each of which can be handled in a variety of ways. Firstly, the issue of correspondence which concerns the comparison of the images, secondly, the calibration of the camera system and thirdly, the physical separation of the cameras in the system.

### 2.2.1 Correspondence

The fundamental concern of a stereo vision system is to uniquely identify points in at least two of the available images that represent a single point in the three dimensional space. Given that the position of the cameras is known, each point in an image represents a single line in three dimensional space. Thus, two such points from different images that are believed to represent the same point in real space will generate two lines in real space that should intersect at that point. This is illustrated in figure 2.1 where the two images contain features corresponding to three of the corners of the object and thus the location of these features in the image can be used to determine the location of the corner features



**Figure 2.1:** *Two camera stereo system*

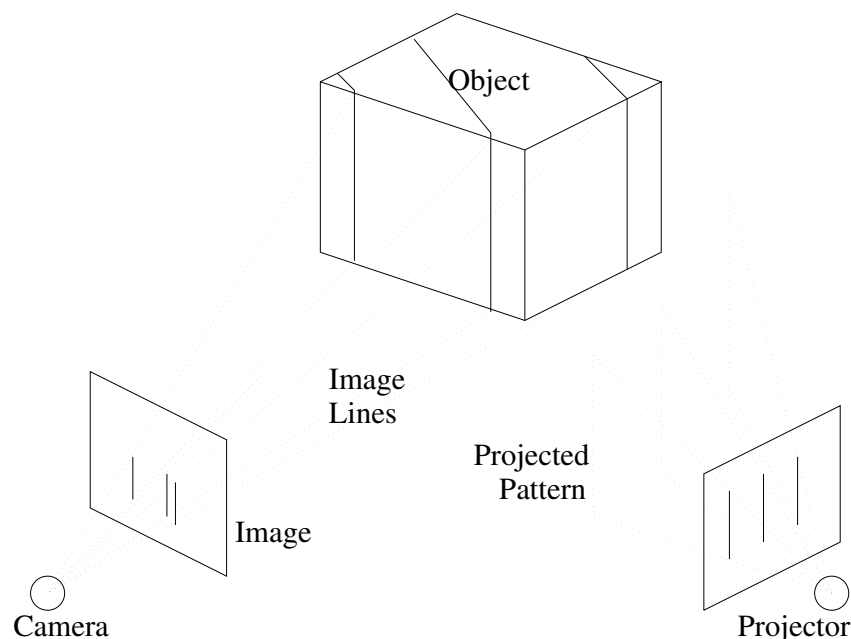
in real space.

### *Image Feature Matching*

Here the major problem is obviously determining the correspondence between feature points in the available images. By placing the cameras in the same horizontal plane and aligning their scan lines horizontally, it is possible to ensure that any potential corresponding point must lie in the same horizontal line in the two images. Some simple geometric considerations can give a range of positions at which a corresponding point might occur, but these are highly dependent on the positioning of the camera and can lead to very large ranges. Thus the success of feature matching often relies on the ability to compare the features from different images and determine if they could have been caused by the same three dimensional feature, for example, an object edge or corner as in figure 2.1

### *Projected Features*

In the case of a geometric object such as a cube, the edges and corners are sufficient to generate features that will create a wireframe three dimensional object but over the flat



**Figure 2.2:** *A projected feature system*

faces of the cube, there will be no image features at all. Thus if it was a slightly rounded cube, for example, it would be impossible to deduce this using image features.

In this instance, the concept of projecting features is useful. One of the cameras would be replaced with a projection device to create a structured light pattern over the object. Each point in the pattern would represent a line in real space. If this point could be linked with a point in the image, it would intersect with the image line at a point in real space on the surface of the object. Geometrically this situation is identical to that of the two camera system, but, in terms of correspondence the task is simplified as a known pattern is being searched for within the image. Obviously, the pattern is distorted by the object it is projected on to, but knowledge of the type of feature being searched for is a considerable asset when searching an image for corresponding features. This is illustrated in figure 2.2 where a projected pattern of vertical lines creates clear line features in the camera image from the object's flat surfaces.

This method then allows smooth surfaces to be dealt with as the projected pattern will create features on the surface. However, the pattern is likely to be sufficiently distorted by sharp edges or corners, that any correspondence in those areas is rendered impossible.

*Summary*

Two (or more) camera stereo matching leads to a potentially highly accurate description of the edges and corners of an object although matching the corresponding image features can be difficult. Lack of features on the object surface prevents the surface shape being deduced.

Projected features allows easier determination of the correspondence. It creates features on smooth surfaces of the object and thus allows their shape to be deduced. However, object edges and corners are likely to disrupt the pattern enough to prevent their position being determined.

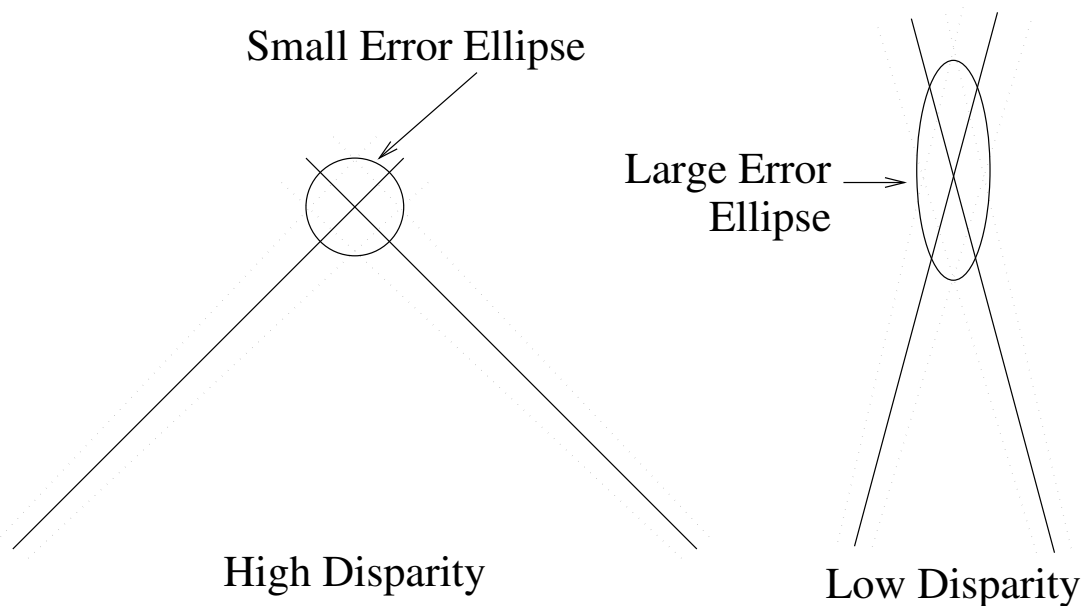
In conclusion, this would suggest that for predominantly smooth and potentially texture free surfaces such as human faces, a projected feature system would be the most suitable.

**2.2.2 Calibration**

The issue of camera calibration is a major research topic in its own right with most work being done in the area of uncalibrated camera systems. In a calibrated system, any point in the camera image can be immediately mapped to a projected line in real space. However, this involves knowing all the parameters and distortions of the lens system, the CCD system and the precise positioning and orientation of the camera with respect to the real world coordinate system. In practice it is often found easier to use an uncalibrated system in which approximate values for the camera parameters are used to get a view of a calibration object, such as a flat surface or a precisely machined cube. Knowledge of the actual dimensions of this calibration object then allow the relevant parameters to be accurately estimated.

**2.2.3 Disparity**

One of the major decisions to be made in the area of stereo vision is the disparity range to be used. If the cameras (or camera and projector) are widely spaced apart then determining the correspondence between image features becomes harder as the disparity in feature position in each image could be large, but because of this large disparity the estimate of the real world coordinates will be very accurate. If the cameras are close together then the corresponding image features will occur close to each other in each image and are thus easy to find. However, the real world coordinate estimate will then



**Figure 2.3:** *Depth estimation errors caused by disparity*

be much less accurate as illustrated in figure 2.3 where the dashed lines represent the uncertainty in real world projection of the image point.

The use of projected features allows the use of larger disparities (and hence more accurate position estimates) as it is easier to deduce correspondence when using a camera image containing known feature types.

The human visual system is a good example of a very low disparity system. The eyes are so close together that virtually all the features visible to one eye are visible in the other<sup>1</sup> so finding the necessary correspondence between the two images is easy. However, the distance information that is deduced is relatively poor although the relative distance information is still accurate. This means that if object A is 20m away and object B is 20.3m away, our estimate of the distance to the object would be poor in each case and the error would easily include the other object, but it would still be clear that object A was closer than B.

---

<sup>1</sup>Indeed it is often uncomfortable when this is not the case, for example, when using a camera or camcorder

### *Summary*

As the aim of this project is to ascertain the shape of the human face, it is essential that accurate positional information is obtained over the whole surface. Therefore it is clear that a wide disparity, projected feature system is likely to produce the best results.

## Surface Extraction

---

The aim of this project has been to develop the ‘front end’ of a practical human face recognition or identification system that will supply the necessary feature information to a standard classification algorithm.

### 3.1 BASIC SYSTEM TYPE

A three dimensional system was selected in order to remove the problems with lighting and orientation experienced by two dimensional intensity image based systems. In considering the acquisition of the three dimensional system it was felt necessary to ensure that the system would not rely on complex or expensive equipment. This is most easily achieved using some form of stereo vision system, as described in the previous chapter. It is also clear from the nature of this problem that a projected feature system would be likely to produce the best results as a human face will not present the clear image features necessary for image correspondence with a two camera system. A further benefit of the projected image system is that it will allow higher disparity placement of the camera and projector resulting in more accurate measurements of the surface points.

### 3.2 PROJECTED FEATURE CHOICE

#### 3.2.1 Feature Shape

Having established that projected features are to be used, there are a number of factors influencing the choice of the projected pattern.

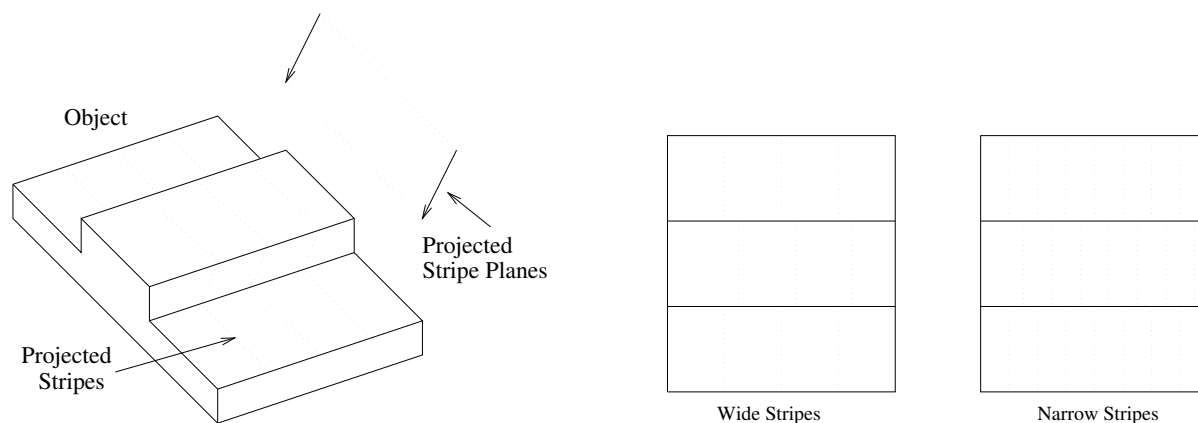
- Each feature must be clear in the camera image
- The position of the feature must be accurately obtainable
- The distortion of the pattern due to projection onto a non-flat surface must not render the pattern unusable in the camera image
- The correspondence between projected features and the camera image must be clear
- There must be enough features in the pattern to allow a comprehensive surface model to be obtained

The pattern selected was a set of vertical stripes as it is a pattern that best matches the above factors:

- Projected vertical stripes will appear as a set of predominantly vertical curves in the camera image
- The position of the edges of each stripe will be accurately obtainable
- Absolute correspondence between each projected edge and each curve in the image will not be possible, but *relative* correspondence will be. e.g. it will be possible to find a set of curves in the camera image that would have been created by adjacent stripes.

### 3.2.2 Feature Density

The only difficulty with the stripe pattern is the trade off between using enough features (e.g. narrow stripes) to obtain a comprehensive surface model and not using a pattern that will be disrupted by the non-flat surface. For example, discontinuities in the surface such as the sides or end of the nose will create breaks in the stripe pattern. In the regions around such breaks, errors may occur in determining the correspondence between the projected features and the image features which, in turn, will cause areas of the calculated surface to be invalid. Any useful system will have to be able to deal with this problem. This situation is illustrated in figure 3.1 where a simple 3D object containing two surface discontinuities is shown. If a narrow stripe spacing is used (as on the right) then there is an ambiguity problem. It is not clear, for the lines in the central section, as to which projected feature they correspond to relative to the lines in the upper and lower section of the object. The immediate solution is to lower the density of stripes (as on the



**Figure 3.1:** *A surface that will cause ambiguity with sufficiently narrow projected stripes*

left), causing a loss in the amount of information that can be obtained about the surface.

### 3.2.3 Multiple Images

From the citations in the first chapter where the use of structured light patterns to establish a surface shape was discussed, it is clear that a single projected pattern has only been used with relatively simple or smooth objects where there will be no potential ambiguities concerning the correspondence of features. In the situations where this is not the case, active systems have been used whereby, for example, widely spaced, non-ambiguous patterns are used but are moved with several images being taken. However, this means that the object must be guaranteed not to move while the images are taken and also involves the use of a projection system that can move in a fully repeatable manner. The laser scanning systems referred to use the simplest implementation of this where a single line is traversed across the object so that there can be no ambiguity problems at all as all the image features will have been caused by the single projected feature.

As the aim of this research has been to develop a solution which does not require complex equipment or impose undue constraints on the subject (for example, having to remain still for several seconds), the multiple image with a moving projection system combination is not deemed suitable.

### 3.2.4 Feature Ambiguity Resolution

In the case that the system will acquire information using only one camera image, considerable effort will be required to deal with the trade off between surface information density and image feature ambiguity. In order to use narrow enough stripes to obtain a sufficient density of surface information, a system to resolve the ambiguities in establishing the correspondences between image features and projected features will be required.

## 3.3 PROBLEM STATEMENT

This dissertation has so far described the purpose behind this research and the basic choice of techniques.

Given the choice of using projected feature stereo vision concepts, the problem is to obtain an accurate model of the surface of a human face using information from a single camera image. The face will be illuminated with an appropriate pattern of vertical stripes.

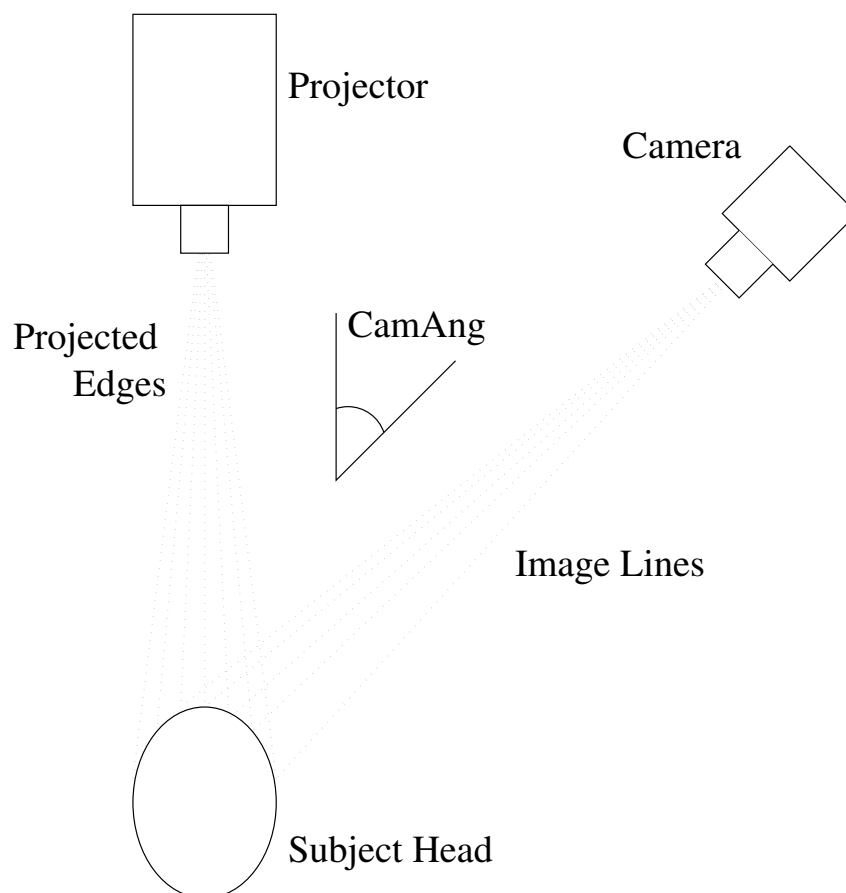
A system that is able to achieve the acquisition of a surface as complex as a human face, using only a single structured light image, will represent a considerable advance from systems such as those described in the citations of the first chapter.

## 3.4 SYSTEM SUMMARY

There are several stages in the process of extracting the 3D surface shape from the supplied image, which are summarised here and described in detail in the following chapters.

### 3.4.1 The Physical Equipment Setup

The equipment used was a CCD monochrome camera (Pulnix TM6 series) and a standard slide projector. A slide containing a suitable vertical stripe pattern was made using a dye sublimation printer onto clear film. A Matrox Meteor frame capture card was used in a Pentium PC to capture the necessary camera images. A plan view of the arrangement is shown in figure 3.2. Due to the shape of the human head and the desire to have a high disparity stereo system, it is clear that with a single camera, only a reduced area of the



**Figure 3.2:** *Plan view of the equipment setup*

face can be within the view of the camera and projector simultaneously. With the setup shown, this was the right half of the face. In order to obtain the full face surface two cameras, one on each side, would be needed, provided that the projector remains centrally positioned.

The head is positioned within the views of both the projector and the camera, roughly orientated straight towards the projector. A single image is captured from the camera for processing. As this is the only physical participation of the human in the process, it is fair to say that this does not involve any equipment or procedures that would be inappropriate for 'field use'. Indeed, there are many circumstances in which the human subject would be otherwise constrained into a volume of space ideal for viewing with this system. Examples include (i) Airport security - as people walk through the metal

detectors, their heads would be within a volume that could be covered by an infra red projector and camera system, (ii) Automated Cash Withdrawal Machines - as people stand in front of the machine to view the screen their heads are within a reasonably small volume, (iii) Any situation involving people walking through a doorway or similarly confined space.

Given the brightness of the slide projector used in the experiments, there would be considerably reduced subject discomfort if the imaging was performed in the infra-red, quite apart from the security benefits.

The camera image is contrast stretched so that the extremes of the grey scale fill the range available in the image storage format used, before being passed on for processing.

### 3.4.2 Image Processing

All the following processing was performed using purpose written C++ code, in some cases following prototyping using Matlab. Further information on the implementation of the algorithms is given in appendix B.

#### *Extraction of Image Features*

The first stage is to extract from the grey scale camera image, all the features that are likely to have been caused by the projected vertical edges. This has to be done in the presence of noise in the image and possible 'background clutter' caused by objects behind the subject head. It would be expected, however, that due to the arrangement of camera and projector, any objects in the background of the camera image will not have been illuminated by the projector and will, therefore, not contain image features created by the projected edges.

As the width of the stripes may vary considerably over the camera image, it is necessary to use a multi level filtering technique as any single filtering process would be unlikely to be able to extract the wide range of feature widths present.

#### *Feature Ambiguity Resolution*

For the stereo process to be used it is necessary to ascertain a correspondence between the features in the image and the projected features. In this case an absolute correspondence is not necessary, a relative correspondence will suffice.

---

As it is the edges of the projected stripes that create the image features, there will obviously be two types of feature - rising and falling edges (as viewed from left to right across the image). This will assist in the correspondence process as it will effectively halve the density of the features.

It is also likely that several image features may correspond with one single projected feature. Where a projected edge has been 'broken' by the presence of a surface discontinuity or a dark area of the surface, such as an eyebrow, several image features will be found. It is, therefore, necessary that the ambiguity resolution process can identify such image features and ensure that they are correctly linked with the appropriate projected feature.

#### *Surface Creation*

Once the image features have been linked with the appropriate projected features, the conversion to a three dimensional surface is straightforward. At this stage any image features that have been erroneously included may become clear as they will cause a surface curvature which is obviously invalid and can thus be removed.

The effectiveness of this stage of the system is increased by foreknowledge of the expected image contents. In this case long, predominantly vertical edges are being searched for as these are known to exist in the image.

The scheme used is a multi-level filtering approach proposed by David Marr in his book, 'Vision' [29]. This is based on filtering with two dimensional Laplacian of Gaussian filters, suggested to him by evidence from retinal cells in animals, followed by zero crossing detection in the resultant filtered images.

In the following sections and chapters a 'right' face<sup>1</sup> view has been used to illustrate typical results. As the techniques described have no directional selectivity (indeed where there is a possibility of such selectivity steps are taken to remove its effects) mirror image results can be assumed for an equivalent 'left' face view.

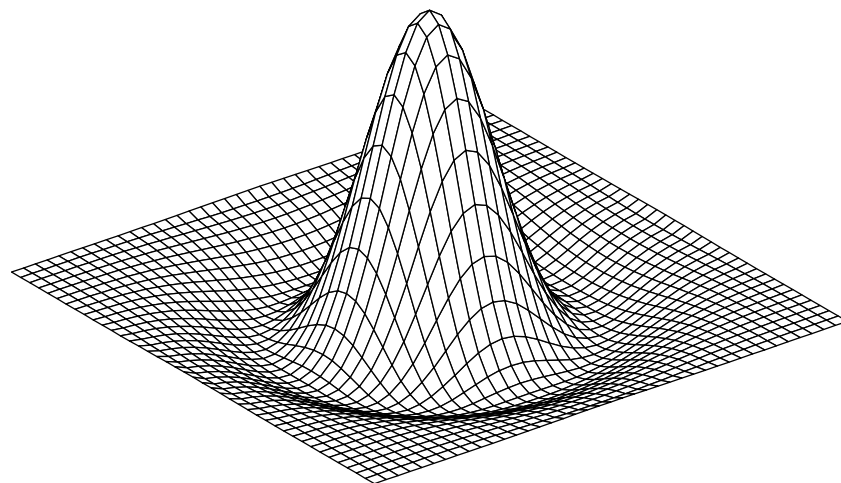
## 4.1 IMAGE EDGE DETECTION

There are a wide variety of methods for detecting edges in images. One of the most widely used methods was described by Canny [9] in which he describes a computational approach to one dimensional edge detection in the presence of noise using Gaussian filtering. He then develops this to a two dimensional approach. One disadvantage is that the edge detector is optimised for a particular shape of edge in a particular amount of image noise. In images containing edges of many differing scales and an unknown signal to noise ratio, a method is needed of combining the detected edges from several different scales of edge detector.

The approach suggested by Marr uses two dimensional Laplacian of Gaussian filters at

---

<sup>1</sup>As in medicine, this description is relative to the 'patient'



**Figure 4.1:** *A 2D Laplacian of Gaussian Filter*

a variety of different scales, and suggests a method by which the zero crossings detected at each scale can be combined together into a final set of detected edges. In this case, the combining of the zero contours from each scale is assisted by the fact that it is predominantly vertical edges which are being searched for.

The implementation of the algorithms described in the rest of this chapter is discussed in appendix B. Where appropriate, more specific references are given in the main text.

## 4.2 FILTERING

An example of the Laplacian of Gaussian filter is shown in figure 4.1. From its shape it is clear that it is a band pass filter and that it will optimally pass features of dimension similar to that of the standard deviation of the filter. The edges of such features will be at or near the zero contour in the filtered image. By the use of several such filters, a range of different width features can be extracted from the image.

### 4.2.1 Filter Implementation

Computationally there is a significant advantage to being able to use separable filtering techniques so that the filtering operation is reduced from an order  $N^2$  operation to an

order  $2N$  operation where the 2D filter is of size  $N \times N$ . This is not possible with the Laplacian of Gaussian as illustrated in figure 4.1. One option would be to filter the image separably with a 2D Gaussian filter and then take the Laplacian of the image. However, taking the Laplacian of the image is still a computationally intensive task. Another option (suggested by Marr) is to use an approximation of a Laplacian of a Gaussian formed by the subtraction of two Gaussians whose standard deviations have a ratio of approximately 1.6 (known as the Difference of Gaussian filter or DOG). By generating a series of images filtered with Gaussian filters where the standard deviation of each filter increases by a factor of 1.6, successive images can be subtracted to obtain a series of DOG filtered images whose standard deviations again vary by a factor of 1.6. Assuming that such a series of DOG filtered images is appropriate, this represents an efficient method of performing the necessary filtering. Further computational advantages can be realised by successively generating each Gaussian filtered image from the previously filtered image and by reducing the number of filter taps. The generation of a sequence of filtered images in this way is very similar to the concept of a non-integer subsampled wavelet decomposition as described by Rioul and Vetterli[38].

### *The Gaussian Filter*

The basic Gaussian filter in both one and two dimensions is given by:

$$g(\sigma, x) = \frac{1}{\sigma\sqrt{2\pi}} \exp \frac{-x^2}{2\sigma^2} \quad (4.1)$$

$$g(\sigma, x, y) = \frac{1}{\sigma\sqrt{2\pi}} \exp \frac{-(x^2 + y^2)}{2\sigma^2} \quad (4.2)$$

To generate a filtered image as though it had been filtered with  $g(\sigma, x, y)$  it is first filtered with  $g(\sigma, x)$  and then with  $g(\sigma, y)$ . An important issue is that of filter truncation as Gaussian filters are technically infinitely long. In this case the filter length was set such that the smallest coefficient was greater than -60dB from the peak coefficient:

$$20 \log_{10} \left( \frac{g(\sigma, x_l)}{g(\sigma, 0)} \right) = -60dB \quad (4.3)$$

$$x_l = \sigma \sqrt{\frac{60dB}{10} \ln 10} \quad (4.4)$$

$$length = 2 \text{floor}(x_l) + 1 \quad (4.5)$$

where *length* represents the number of taps in the symmetrical filter.

As a sequence of  $N$  Gaussian filtered images is required, each image can be derived from the previously filtered image as Gaussian filters convolve together to give Gaussian filters such that the required standard deviation for filtering image  $n$  to obtain image  $n+1$  is given by:

$$\sigma = \sqrt{\sigma_{n+1}^2 - \sigma_n^2} = \sigma_n \sqrt{1.6^2 - 1} \quad (4.6)$$

The number of filter taps can be reduced in this filter by ensuring that the resultant image peaks in the filter response caused by the subsampling are in areas where the filter response from the previous filter is more than 60dB down from the peak. The first of these subsampling peaks occurs at  $\frac{2\pi}{interval}$  where *interval* represents the tap interval (*interval* = 1 for no subsampling) and the fourier transform of  $g(\sigma, x)$  is given by  $G(\sigma, \omega)$ .

$$g(\sigma_n, x) = \frac{1}{\sigma_n \sqrt{2\pi}} \exp\left(\frac{-x^2}{2\sigma_n^2}\right) \quad (4.7)$$

$$G(\sigma_n, \omega) = \exp\left(\frac{-\omega^2 \sigma_n^2}{2}\right) \quad (4.8)$$

$$20 \log_{10} \left( \frac{G(\sigma_n, \omega_i)}{G(\sigma_n, 0)} \right) = -60dB \quad (4.9)$$

$$\omega_i = \frac{1}{\sigma_n} \sqrt{\frac{60dB \ln 10}{10}} \quad (4.10)$$

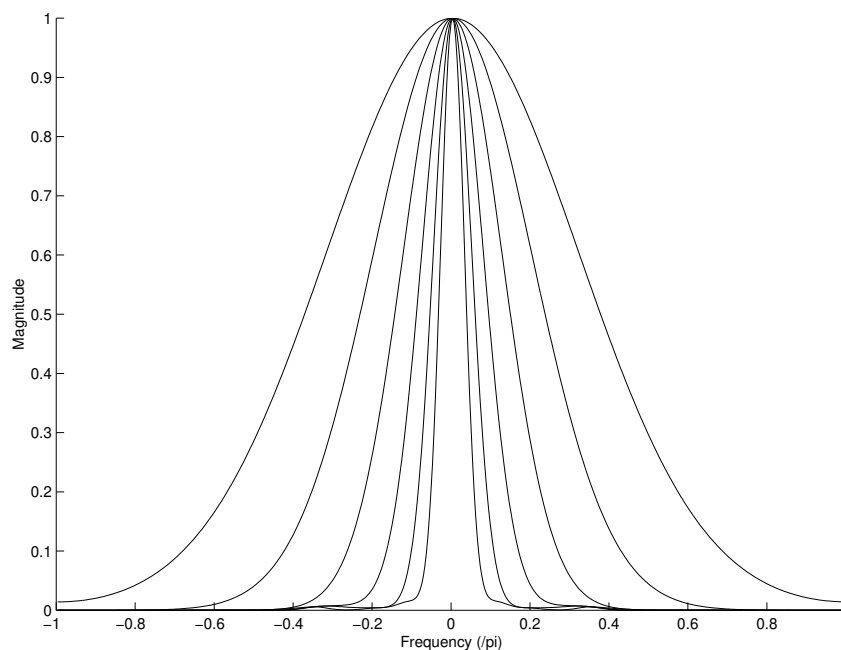
$$interval = floor \left( 2\pi \sigma_n \sqrt{\frac{10}{60dB \ln 10}} \right) \quad (4.11)$$

Table 4.1 at the end of the chapter shows the coefficients used at each scale to perform these filter operations. As these filters are symmetric about the middle tap, only half the coefficients are shown. Each of these filters is used successively on the previous filtered image to produce a sequence of  $N$  images, where each image,  $n$  from 2 to  $N$ , has been filtered with a Gaussian filter of standard deviation:

$$\sigma_n = \sigma_1 1.6^{n-1} \quad (4.12)$$

The frequency response of these filters is shown in figure 4.2. Note that despite the subsampling of the filters to produce an approximation to the large standard deviation Gaussians, the frequency responses are still very smooth.

The implementation of this type of filter is discussed further in section B.2.1.



**Figure 4.2:** *Frequency response of the six Gaussian filters*

---

### *The Difference of Gaussian Filter*

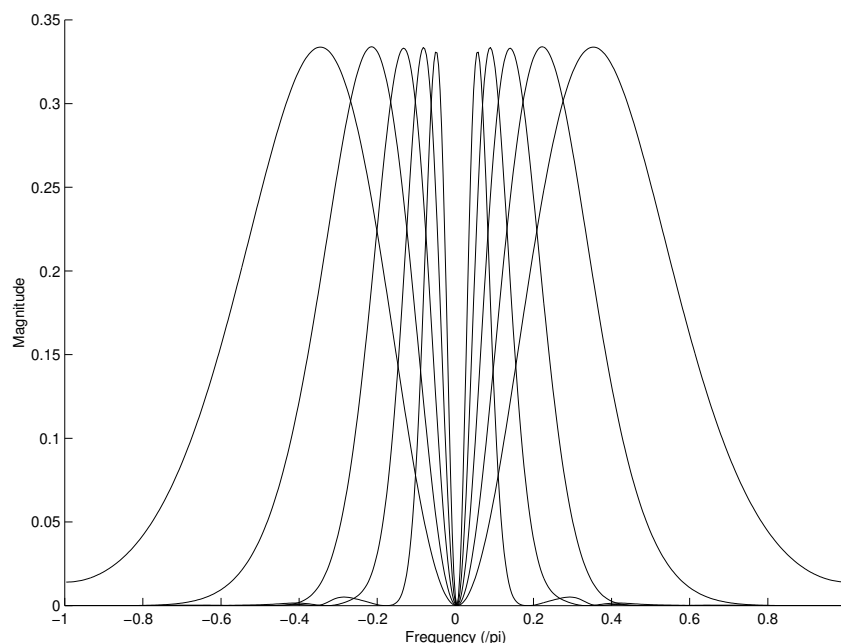
By subtracting image  $n + 1$  from image  $n$  a sequence of  $N - 1$  images is obtained, each of which have been filtered with a close approximation to a Laplacian of Gaussian filter:

$$\nabla^2 g(\sigma, x, y) = \frac{x^2 + y^2 - 2\sigma^2}{\sigma^5 \sqrt{2\pi}} \exp -\frac{x^2 + y^2}{2\sigma^2} \quad (4.13)$$

The frequency response of these filters is shown in figure 4.3

A sequence of five such images, with the original, is shown in figure 4.4. Note how the increasing filter scale picks out progressively wider stripes in the original image.

Sections B.2.1 and B.2.2 describe the data structures and routines used for this filtering approach.



**Figure 4.3:** *Frequency response of the five DOG filters*

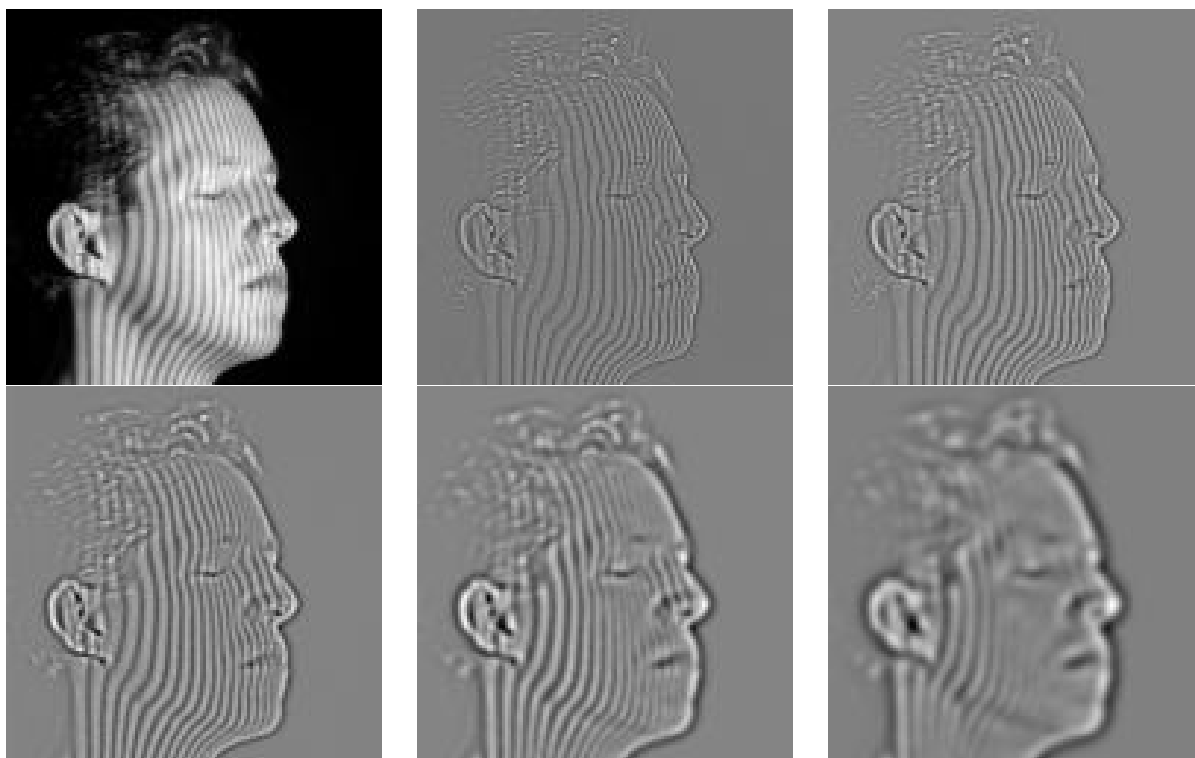
---

## 4.3 ZERO CROSSING EXTRACTION FROM FILTERED IMAGES

For each filtered image, the points at which the intensity values cross zero have to be found and these zero crossing points must be linked together into a set of lines or zero crossing contours.

### 4.3.1 Zero Crossing Line Location

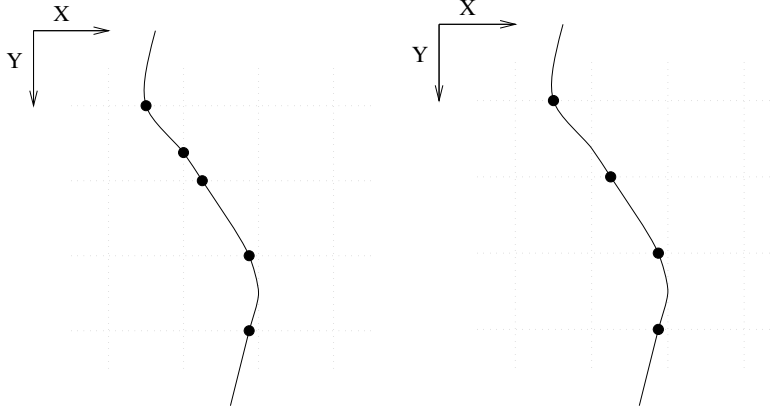
The standard method of locating such contours is to examine each rectangle in the sampling grid to see if the contour crosses into the rectangle. Figure 4.5 shows a group of nine such rectangles with a contour cutting through four of them. As the location of the contour can only be deduced from the values on the sampling grid, linear interpolation of these values would give the locations marked by dots. Note that each of these locations is characterised by one of the coordinates being an integer value and the other being an interpolated, floating point value. As the zero crossing contours in these filtered images are expected to be predominantly vertical, it is only necessary to record the contour lo-



**Figure 4.4:** *Original and 5 filtered images*

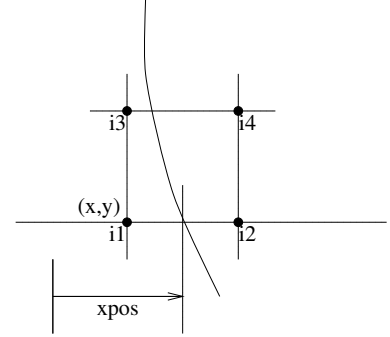
cations on the horizontal lines of the grid, thus giving rise to zero crossing locations with a floating point value for the x coordinate and an integer value for the y coordinate, as shown in figure 4.6.

To locate the zero crossing lines from the image data, each horizontal line from top to bottom in the filtered image is traversed to find zero crossings. Each zero crossing that is found is considered for inclusion in any already existing zero crossing line. As the process occurs from the top to the bottom of the image, the zero crossing lines grow downwards with new zero crossings being added to the bottom of the appropriate line as they are discovered. If it is not possible to add any discovered zero crossing to an already existing zero crossing line, then a new line is started with that zero crossing as the first (topmost) point in the line.



**Figure 4.5:** *Contour marking*

**Figure 4.6:** *Simplified contour marking*



**Figure 4.7:** *Contour slope*

### 4.3.2 Data Objects

Two types of data object are needed for the storage and manipulation of the zero crossings and the zero crossing lines.

#### *The Zero Crossing Data Object*

The position of the zero crossing is determined by linear interpolation along the horizontal line and the image intensity gradient is determined approximately with reference to the pixel values in the horizontal line above, as shown in figure 4.7:

$$xpos = x + \frac{i_1}{i_1 - i_2} \quad (4.14)$$

$$slope = \sqrt{\left(\frac{\partial I}{\partial x}\right)^2 + \left(\frac{\partial I}{\partial y}\right)^2} \quad (4.15)$$

$$\simeq \sqrt{(i_2 - i_1)^2 + \left(\frac{i_3 + i_4}{2} - \frac{i_1 + i_2}{2}\right)^2} \quad (4.16)$$

where  $I$  is the pixel intensity in the filtered image and  $i_n$  are the values of  $I$  at the specified locations. The image intensity gradient is stored only as a scalar value, called 'slope' in the zero crossing data object and the sign of this gradient is stored separately. In the filtered images both rising and falling edges are present (as viewed from left to right) and these will generate opposite polarity slopes.

Having located a zero crossing and determined its slope a zero crossing data object is created to store the information, provided that the magnitude of the slope is greater than a specified value.

The Zero Crossing Data Object

Object Type	Object Name	Comment
float	x	The x position of the zero crossing
integer	y	The y position of the zero crossing
float	slope	The slope of the zero crossing
integer	sgn	The polarity of the zero crossing

#### *The Zero Crossing Line Data Object*

For a zero crossing to be included in a zero crossing line data object, it must fit a set of criteria:

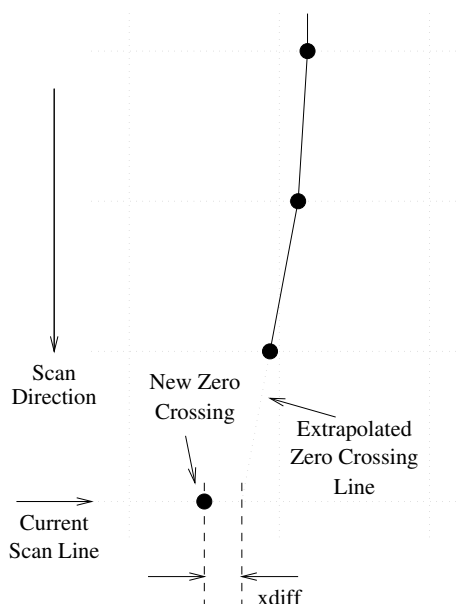
- it must be of the same polarity
- the y coordinate of the zero crossing must be 1 greater than the current bottom end of the line being considered
- the x coordinate of the zero crossing must be near to the extrapolated zero crossing line being considered as shown in figure 4.8. e.g. xdiff must be less than a certain value.

If a suitable zero crossing line is not found, then a new one is created with the zero crossing included as the first point in the new line.

The Zero Crossing Line Data Object

Object Type	Object Name	Comment
integer	sgn	The slope polarity
integer	topend	The start point of the line
integer	length	The length of the line
float	x_values[length]	The x location of the zero crossing
float	slope_values[length]	The slope of the zero crossing

The manner in which this zero crossing line data is selected and stored in the implementation of this system is discussed further in sections B.2.3, B.2.4 and B.2.5.



**Figure 4.8:** Adding a zero crossing to a zero crossing line

### 4.3.3 Zero Crossing Line Selection

When the entire filtered image has been traversed, the set of zero crossing line data objects has to be examined for zero crossing lines that are redundant or contain irrelevant information. Figure 4.9 shows all the zero crossing lines extracted from the filtered images 3 and 4 in figure 4.4. Note that there are many zero crossing lines in areas of the image that do not contain projected edges, such as the hair. These lines will not cause spurious output in the final combined set of lines, as the zero crossing line combinational process will automatically reject them, however, the large number of lines involved (several hundred in each set shown in figure 4.9) presents a large computational load. It is therefore sensible to find a simple method of rejecting as many of these unnecessary lines as possible.

The lines are rejected when they fail the following tests:

- The magnitude of the slopes averaged along the line must be greater than a specified value.
- The line length must be greater than a specified value.

After this final selection stage, a typical set of zero crossing lines would be like those shown in figure 4.10. These five sets of lines were generated from the filtered images



**Figure 4.9:** *Zero Crossing lines from filtered images 3 and 4 in figure 4.4*

shown in figure 4.4. These five sets of zero crossing lines are then combined together to create a single set of image edges.

These line sets are manipulated in the program implementation by routines discussed in sections B.2.4 and B.2.5.

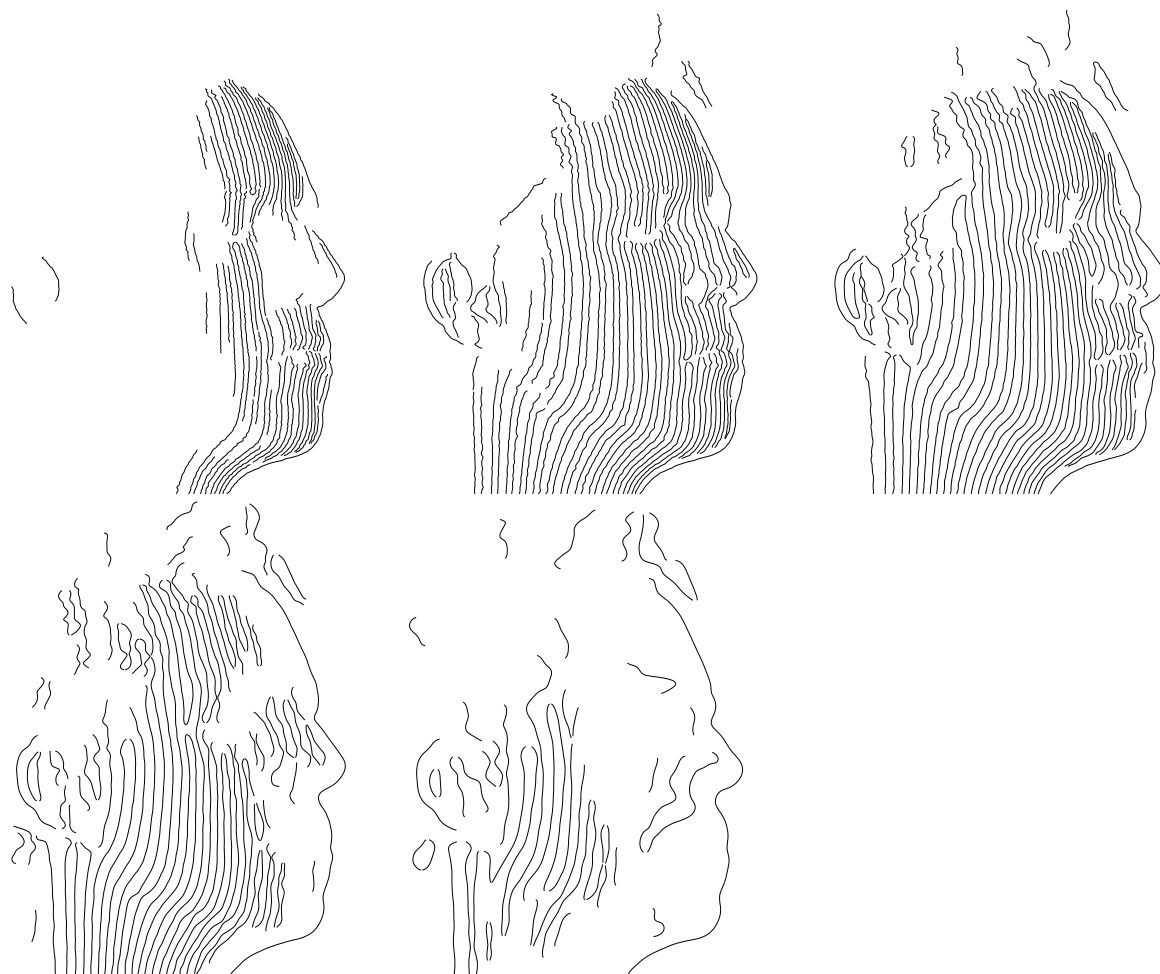
## 4.4 ZERO CROSSING LINE SET COMBINATION

Each set of zero crossing lines extracted from one of the filtered images contains edges of different scale. These sets of lines must be combined together to produce a single set of lines that represent the edges extracted from the image, known as Image Edges.

The principle behind this process is that a genuine image edge will appear in several of the line sets extracted from the filtered images, at a close physical location in each one. This can be crudely illustrated by overlaying the five sets of lines in figure 4.10 on top of each other, as shown in figure 4.11

Any lines from each of the line sets that do not have corresponding lines in at least one other line set are not considered to be lines that could represent an image edge.

This process must also successfully deal with image edges that change scale along their length. For example, many of the stripes in the camera image shown in figure 4.4 are



**Figure 4.10:** *The five sets of extracted zero crossings*

---

much wider in the centre part of the image than they are at the top or bottom. This implies that, for example, the middle section of such an image edge will appear in line sets 2,3 and 4 whereas the top and bottom sections of the image will appear in line sets 1,2 and 3. In this case, although there will be two lines in line set 1 that correspond to this image edge, they must both be included in the final image edge. This process is illustrated in the test grid pattern shown in figure 4.12. Here, line sets 1,2 and 3 all contain 20 zero crossing lines and line sets 4 and 5 contain 10 zero crossing lines. In this case, it would be expected to produce 10 image edges where each image edge was made up from two lines in each of line sets 1,2 and 3 and one line in each of line sets 4 and 5.



**Figure 4.11:** *The five line sets of figure 4.10, overlaid*

---

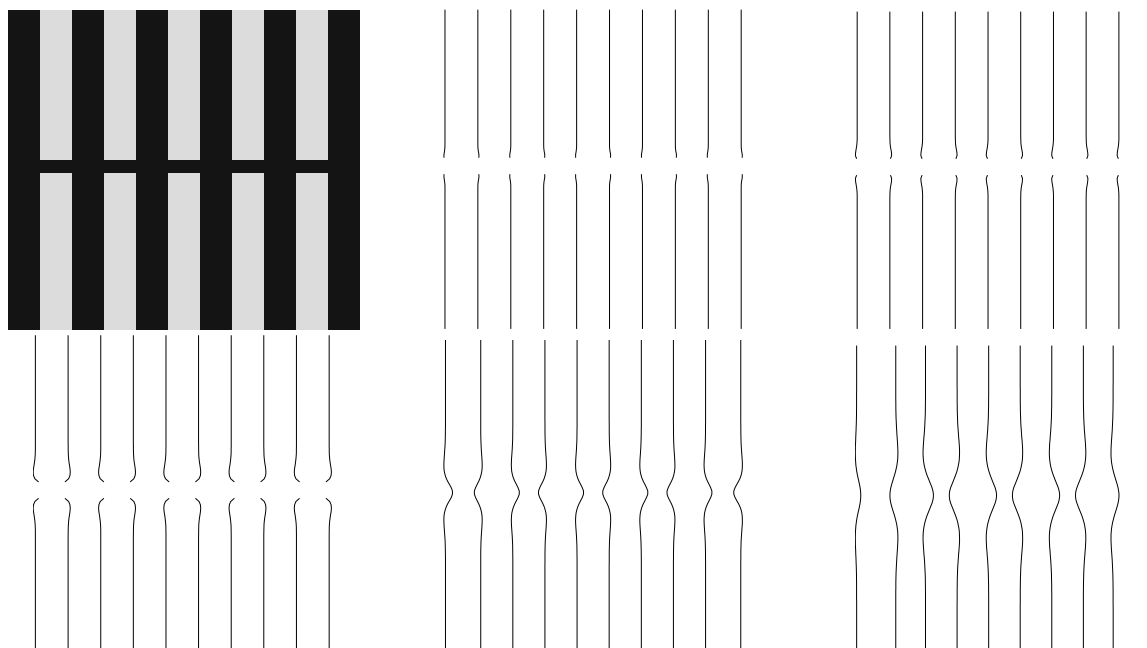
Sections B.2.3, B.2.4 and B.2.5 discuss further the implementation of the routines that perform the techniques described in this section.

#### 4.4.1 Zero Crossing Line Set Searching

The process of finding groups of zero crossing lines that can be collected together as a single image edge is implemented as a recursive search which will mark all the lines in the line sets that are deemed to form an image edge. Having marked these lines, they are removed from the sets and combined into a single image edge.

##### *The recursive search*

A function is defined whose purpose is to find and mark (or tag) any lines in line set  $N$  that are similar to a given line  $L$  and to then instigate a search in the adjacent line sets ( $N - 1$  and  $N + 1$ ) for lines similar to the tagged lines. Lines that have been marked will



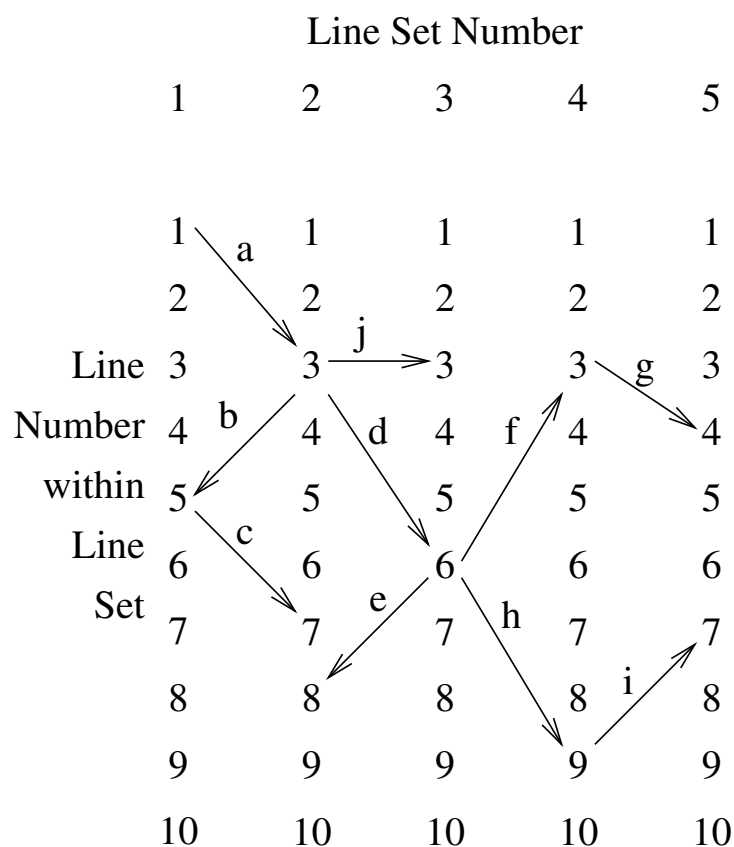
**Figure 4.12:** *The five line sets generated from the test grid*

be removed from the line sets after the search is complete and will either be ‘merged’ or discarded as appropriate.

```
function find_line(line set N,line L)
```

- 1) Mark line L
- 2) Let T be first unmarked line in line set N
- 3) If T is similar to L then find\_line(N+1,T) and find\_line(N-1,T)
- 4) Move T to next unmarked line in line set N
- 5) Goto step 3 until all lines in line set N have been tested

This function is initially called with the request to search line set 2 with reference to the first line in line set 1. As the order of the lines in the line sets is arbitrary (it is actually the order in which the start point zero crossings were discovered) it is not significant where the search starts as all lines will eventually be involved in the search. An example of how this could work for an image edge that has varying scale and thus creates zero crossing lines in all of the line sets, is shown in figure 4.13. The search starts with line 1 in line set 1, referred to as line(1,1). Line set 2 is then searched for similar lines and line(2,3) is found (step a). Line set 1 is then searched for lines similar to line(2,3) and line(1,5) is



**Figure 4.13:** A recursive search for lines forming an image edge

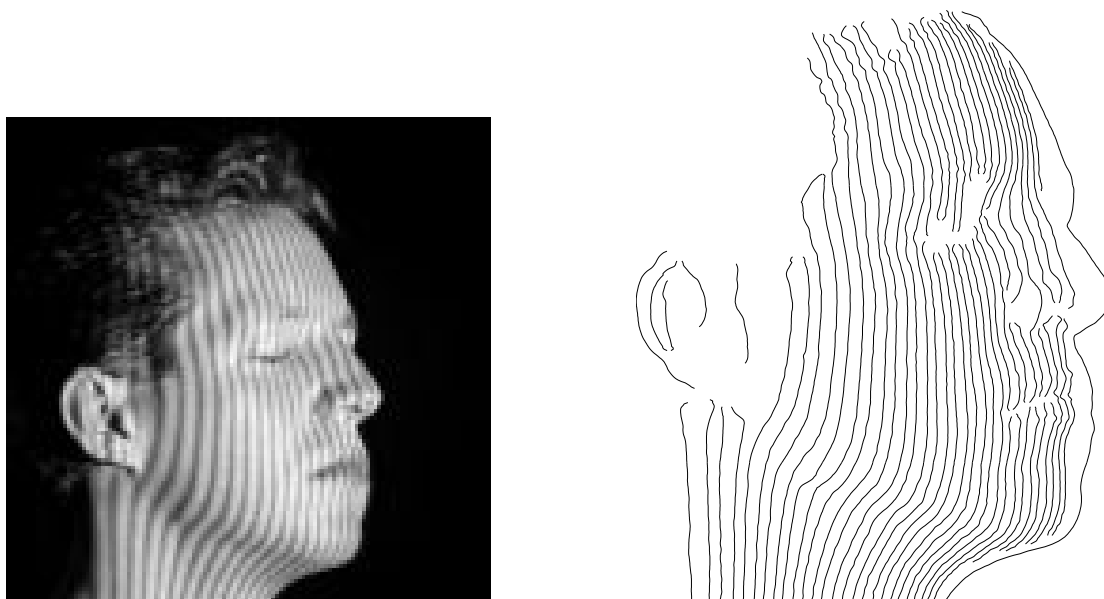
found (step b). This process continues until the recursion process terminates.

Then the lines that have been marked are all removed from the line sets and if more than two lines have been marked they are merged into one image edge. If only the search start line is present, or the search start line and one other, then that is deemed not significant enough to form an image edge, so those lines are simply deleted.

#### *Zero Crossing Line Merging*

The lines in the line sets which were created by the filters of lower standard deviation have greater positional precision and therefore play a greater part in determining the position of the image edge, as the wider the filter used on the image, the greater the uncertainty of the zero crossing position.

The process of merging consists of creating a new image edge data object whose



**Figure 4.14:** *The original image with the extracted image edges*

length will enclose all the lines within the marked set and at each vertical position down the image edge, the x\_values of the marked lines are averaged together to give the image edge x\_value. This average is weighted with the slopes of the marked lines at that point. Note that this will include the effect of weighting against filter standard deviation as the values of slope obtained from the images filtered with the widest filters will be lower.

The Image Edge Data Object

Object Type	Object Name	Comment
integer	sgn	The slope polarity
integer	topend	The start point of the line
integer	length	The length of the line
float	x_values[length]	The x location of the zero crossing

#### *Line Set Removal*

Each search is started with the first line in the first available line set and the search always concludes with the removal of the marked lines, regardless of whether they have been merged into an image edge or not. Eventually, various line sets may become empty, particularly the first one. If either the first or last line set becomes empty, then it can

---

be removed, and the set numbering is moved along as appropriate, so that if the first set is removed set 2 becomes set 1. The search then proceeds with the new set numbering. Thus, regardless of the number of image edges created, the lines sets will eventually become empty and this represents the conclusion of the image edge generation process.

## 4.5 CONCLUSIONS

The set of image edges that the zero crossing line combinational process generates is a definite list of long, predominantly vertical, rising and falling edges in the supplied image. Such a set of image edges is shown in figure 4.14 along with the original camera image that they were generated from.

The image edge features then have to be passed to a correspondence process that can match them with the projected edge that caused them. Note that at this stage, it is clear that there may be several image edges each representing a different part of the same projected edge. It is for the correspondence process to resolve this issue, as from an image edge extraction point of view, they are all separate image edges. As implemented, this system demonstrates a highly successful method of extracting the projected image features from the supplied camera image.

Level:	1	2	3	4	5	6
Tap:						
0	0.3991	0.3195	0.3993	0.4991	0.4679	0.5361
1	0.242	0.2319	0	0	0	0
2	0.05401	0.08865	0.242	0	0	0
3	0.004433	0.01785	0	0	0	0
4	0	0.001894	0.05386	0.2282	0	0
5	0	0	0	0	0	0
6	0	0	0.004404	0	0.2352	0
7	0	0	0	0	0	0
8	0	0	0.0001322	0.02182	0	0
9	0	0	0	0	0	0
10	0	0	0	0	0	0
11	0	0	0	0	0	0.2173
12	0	0	0	0.0004362	0.02988	0
13	0	0	0	0	0	0
14	0	0	0	0	0	0
15	0	0	0	0	0	0
16	0	0	0	0	0	0
17	0	0	0	0	0	0
18	0	0	0	0	0.0009594	0
19	0	0	0	0	0	0
20	0	0	0	0	0	0
21	0	0	0	0	0	0
22	0	0	0	0	0	0.01448
23	0	0	0	0	0	0
24	0	0	0	0	7.784e-06	0
25	0	0	0	0	0	0
26	0	0	0	0	0	0
27	0	0	0	0	0	0
28	0	0	0	0	0	0
29	0	0	0	0	0	0
30	0	0	0	0	0	0
31	0	0	0	0	0	0
32	0	0	0	0	0	0
33	0	0	0	0	0	0.0001584

**Table 4.1:** *The subsampled Gaussian filter coefficients*

The problem of determining which projected edge caused the detected image edges has traditionally limited the effectiveness of projected pattern stereoscopic vision systems, when using only a single camera image.

The problem is caused by discontinuities in the surface being scanned, or a local colouration of the surface that prevents the projected edges from being detected. In these circumstances, it may be that several of the detected image edges are actually all part of the same projected edge, although they are itemised as separate edges in the list, each representing a different section of the projected image edge. As described in chapter 2 there is a direct trade off between depth measurement accuracy and correspondence range. For a narrow angular disparity between the camera and projector, the breaks in features caused by surface discontinuities will be small but the depth estimation will be poor. For a large angular disparity between camera and projector the depth estimation will be good but the breaks in features caused by surface discontinuities may be large enough to prevent a correspondance being formed at all.

It is for this reason that many of the available commercial 3D object scanning systems use a single projected line of light which is swept across the object and multiple images of this projection are taken. In this case, in each frame of the image sequence there will be only one projected edge, thus all detected image edges must be part of this projected edge. Other systems that use multiple projected edges are designed for use with surfaces that are of uniform colouration and do not contain any discontinuities, such as systems designed to measure the strain of metal sheets.

In the particular example of human faces, as is visible in figure 4.14, there are likely to be problems over the eyes and eyebrows, the mouth and the nose. The eyebrows cause feature breaks due to the surface colouration obscuring the projected feature's visibility, whereas the nose, for example, causes feature breaks due to significant surface discontinuities. For successful stereo correspondance to be achieved it is necessary to establish exactly which image edges have been created by which projected edge. For

example, the lines above the eye have to be paired up with the corresponding lines below the eye in order that they can both be assumed to have been created by the same projected edge.

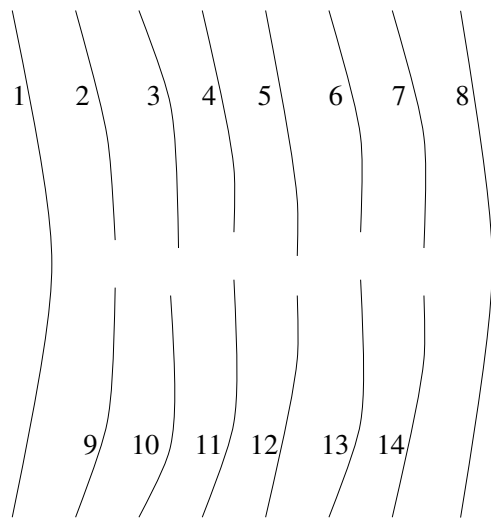
A novel technique has been developed to perform this task using a structure very similar to that of a graph as described by the field of mathematics known as graph theory. It is referred to as the Adjacency Graph as it records the manner in which image edges are physically positioned, in particular whether they are next to each other, or not.

## 5.1 THE ADJACENCY GRAPH

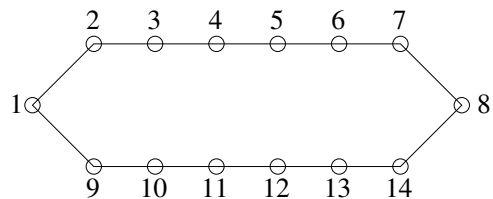
The purpose of the adjacency graph is to record the physical relationship of all the image edges to each other - in particular, those image edges that are adjacent to each other. By positively identifying that a pair of image edges A and B are both immediately to the left of image edge C, it is then possible that image edges A and B may be part of the same projected edge and a further test can determine whether this is the case.

A simple example of the kind of structure that was initially envisaged is shown in figures 5.1 and 5.2. The artificial image edge pattern in figure 5.1 represents eight projected edges of which six have been split into two image edges each. The graph structure shown in figure 5.2 represents each image edge as a node in the graph. Adjacency of one image edge to another is represented by the edges joining the nodes. Thus in this case image edges 3 and 4 are adjacent to each other, but 5 and 14 are not.

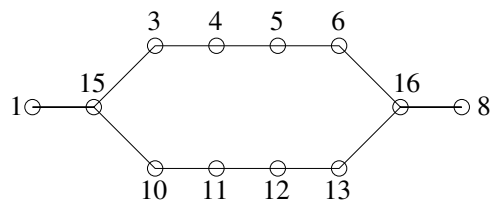
From this graph, it was hoped that a simple examination of the structure would reveal that, for example, as image edges 2 and 9 are both immediately to the right of image edge 1, they are suitable candidates to check to see if they could be part of the same projected edge. A simple check of correct edge polarity and proximity of the image edge endpoints was proposed for this purpose. If the image edges are deemed to be part of the same edge then nodes 2 and 9 are removed and replaced with node 15, representing the combined image edges. In a very similar manner, image edges 7 and 14 would be checked, removed and combined into node 16, as shown in figure 5.3. This process would then be repeated as nodes 3 and 10 would be immediately adjacent to node 15 and so on, until the entire graph becomes reduced to a simple line. If this occurs, then a very good estimation of stereo correspondence could be made by assuming that each node



**Figure 5.1:** Simple image edge set



**Figure 5.2:** Simple graph structure derived from figure 5.1



**Figure 5.3:** Simple graph structure after one iteration

represented a consecutive projected edge.

Obviously, in practice, the graphs would be expected to be significantly more complex and, in some cases, it would not necessarily be possible to collapse the graph to a single line.

The implementation of the algorithms described in the rest of this chapter is discussed further in appendix B. Where appropriate more references are given in the main text.

## 5.2 GRAPH IMPLEMENTATION

This section details how the graph is created and the operations that can be carried out on the graph in order to attempt to collapse it to a single line where each node represents the correspondence with a projected feature.

Remember that the eventual purpose of the graph is to allow a correspondence to be made between edges found in the image and edges projected onto the face. Thus we know that these edges should alternate in polarity across the graph (as they do across the face). However, in many of the situations in which it might seem obvious to use this alternation

of polarity as extra information to assist in the graph manipulation, this should not be done. Almost every type of insertion or removal operation that could be performed on the graph would violate this rule as would the presence of image edges that are not created by projected edges or the absence of image edges that should be there. In order not to restrict the graph manipulations in this manner, use of image edge polarity information is confined to the Node Collapse operations (section 5.2.9) in which the use of polarity is fundamental.

The range of operations that are necessary to manipulate the graph are described in the following sections as detailed below:

**5.2.1: The Node Data Object** - This section gives a description of the information that it is necessary to store in each node of the graph. Basically each node initially contains a single image edge object and later, as nodes are merged together, will contain a group of image edge objects.

**5.2.2: The Node Comparison Operators** - As the graph is intended to describe the physical relationship between image edges, a set of comparison operators are necessary. This section details the types of comparison that are possible and how they are implemented.

**5.2.3: Graph Creation** - The graph must be filled with nodes sequentially as it is obviously not possible to create the graph structure in a single step. This section describes the process.

**5.2.4: Node Insertion** - The insertion of nodes into the graph is one of the key issues. This section describes the possible ways in which the insertion of nodes may occur.

**5.2.5: Node Removal** - In order to provide a complete set of graph manipulation operations, node removal must be possible. As this section describes, certain nodes, by virtue of their particular relationship to their neighbours, may not be removable. However, a process is described for those which are. Node removal is required for the shuffling operations described in section 5.2.8

**5.2.6: Graph Display** - A simple method of depicting the graph structure is described in this section. This is most useful for seeing how the structure is changed by certain operations.

**5.2.7: Graph Shape** - This section discusses the effect that altering the order in which image edges are presented during graph creation has on the initial graph shape.

**5.2.8: Graph Node Shuffling** - A method to allow the graph to ‘relax’ into the most appropriate shape regardless of the order in which image edges were presented.

**5.2.9: Graph Collapsing** - This is the ‘raison d’être’ of the proposed adjacency graph technique. It is the process by which nodes in the graph are merged in order to group image edges together to indicate that they form part of the same projected edge. It is by repeating this process that the desired stereo correspondence is formed. This section also details the possible final structures of the graph after collapsing has taken place.

**5.2.10: Graph Processing Output** - This section describes the form of the output data which is to be presented to the surface generation process described in the next chapter.

### 5.2.1 The Node Data Object

Each node refers to a collection of image edges, each image edge being extracted from the camera image as shown in the previous chapter. Thus each node data object must contain all the image edge information as well as all the graph linking information. In the initial creation of the graph, each node will only contain one image edge data object but, as the graph is collapsed, nodes will necessarily refer to several image edge data objects and will thus mark them as having been created by the same projected edge.

The Node Data Object

Object Type	Object Name	Comment
integer	num_ie	number of image edges
IE	ie_data[num_ie]	pointers to image edge Data Objects
integer	num_next	number of nodes to the right
NODE	next[num_next]	pointers to nodes to the right
integer	num_prev	number of nodes to the left
NODE	prev[num_prev]	pointers to nodes to the left

Thus for nodes 15 and 16, for example, in figure 5.3 the Node Data Objects would look like this:

Node 15

Object Type	Object Name	Object Value
integer	num_ie	2
IE	ie_data[num_ie]	2 sets of image edge data
integer	num_next	2
NODE	next[num_next]	3,10
integer	num_prev	1
NODE	prev[num_prev]	1

Node 16

Object Type	Object Name	Object Value
integer	num_ie	2
IE	ie_data	2 sets of image edge data
integer	num_next	1
NODE	next[num_next]	8
integer	num_prev	2
NODE	prev[num_prev]	6,13

Thus a Graph Data Object need only contain a pointer to the first (leftmost) Node Object and then the Node Data Objects hold the connectivity information for the rest of the graph. For convenience, the last (rightmost) Node Object in the graph is also held.

Graph Data Object

Object Type	Object Name	Comment
NODE	first	pointer to first NODE in graph
NODE	last	pointer to last NODE in graph

The node data object implementation is described in sections B.2.6 and B.2.7.

### 5.2.2 The Node Comparison Operators

For the connectivity of the graph to represent the adjacency of the image edges, it is necessary to have a set of operators which can compare two lines and establish the relationship between them.

Two such operators are needed, each comparing two nodes and returning a boolean result. They are an ‘is to the left of’ operator and an ‘is to the right of’<sup>1</sup> operator.

<sup>1</sup>These slightly odd names were chosen so that, when implemented in the C++ language, statements

The first step is to establish two similar operators that would operate simply on two image edges. Then, given that nodes are collections of image edges, the operators could be expanded to operate on nodes.

### *Image Edge Comparison Operators*

The ‘A is to the left of B’ image edge operator returns True if and only if image edges A and B both overlap in the vertical direction and considered at each horizontal level of this vertical overlap, A is to the left of B.

The ‘A is to the right of B’ image edge operator returns True if and only if image edges A and B both overlap in the vertical direction and considered at each horizontal level of this vertical overlap, A is to the right of B.

Obviously, if A and B do not have a vertical overlap then both operators will return False.

The geometry of the operator is shown in figure 5.5 in which image edge B is clearly to the right of image edge A. The following checks are made on the two lines being compared:

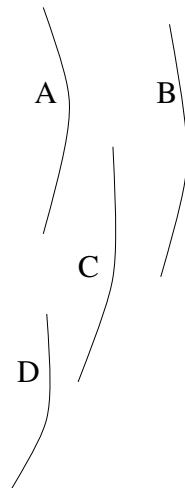
- Is the Vertical Overlap Distance greater than GraphLinesOverlap (an adjustable parameter) and
- at each Absolute Vertical Position within the vertical overlap, is the polarity of Xdiff consistent with the operator.

Operator Summary

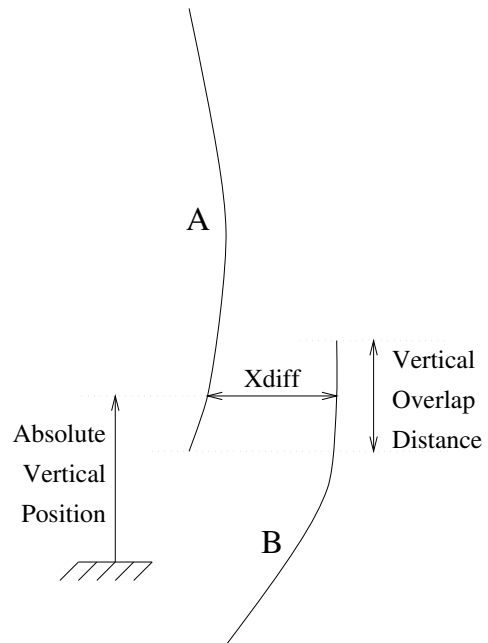
A is to the left of B	A is to the right of B	Conclusion
True	False	A is to left of B
False	True	A is to right of B
False	False	No vertical overlap
True	True	Impossible combination

In the example image edges shown in figure 5.4 the following comparisons would be made:

such as if `A.istothelleftof(B)` would give the boolean result of the comparison of A with B and would also appear to ‘make sense’ from an English language point of view



**Figure 5.4:** Sample image edges to demonstrate comparison operators



**Figure 5.5:** Operator Geometry

		X is to the left of Y				
		Y				
		A	B	C	D	
X	A			T	T	F
	B	F			F	F
	C	F	T			F
	D	F	F	T		

		X is to the right of Y				
		Y				
		A	B	C	D	
X	A			F	F	F
	B	T			T	F
	C	T	F			T
	D	F	F	F		

These comparison operators are implemented by routines discussed in sections B.2.3 and B.2.6.

*Node Comparison Operators*

Given that a Node in the Graph is a collection of one or more image edges, it is relatively straightforward to use the image edge operators to compare the edges within one node with the edges within the other node.

For example, to test whether ‘node X is to the left of node Y’, each image edge within

X is compared with each image edge within Y. The result of the operator is only true if at least one of the image edges in X is to the left of one of the image edges in Y *and* none of the image edges in X are to the *right* of any of the image edges in Y.

Similarly, to test whether ‘node X is to the right of node Y’, each image edge within X is compared with each image edge within Y. The result of the operator is only true if at least one of the image edges in X is to the right of one of the image edges in Y *and* none of the image edges in X are to the *left* of any of the image edges in Y.

These operators can then be used to allow the creation of the graph and the insertion of new nodes into the graph.

These node comparison operators are implemented by routines discussed in sections B.2.6 and B.2.7.

### 5.2.3 Graph Creation

Graph creation and node insertion have much in common as the process of creating a graph is simply a process of inserting nodes into an initially empty graph.

The graph creation process begins with the list of image edges detected in the previous chapter. Each of these is used to create a Node Data Object which contains just one image edge. Two extra nodes are created, each containing an artificial image edge of full image height, one to the extreme left of the image and one to the extreme right. The presence of these two special nodes, ensures that all subsequently inserted nodes are guaranteed to be to the left of one and to the right of the other. Thus these extra nodes will be ‘first’ and ‘last’ in the Graph Data Object.

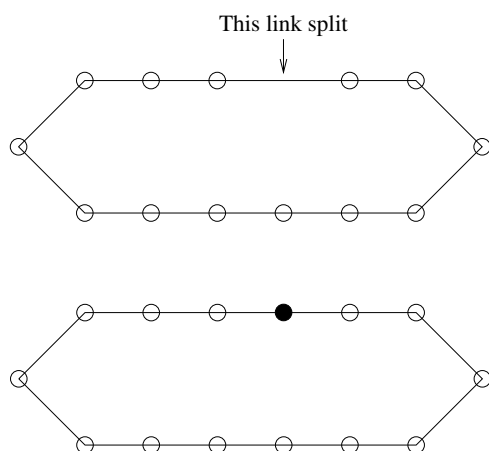
The remaining nodes are inserted into the graph in the order in which they were listed by the image edge extraction process.

The implementation of this routine is discussed in B.2.7.

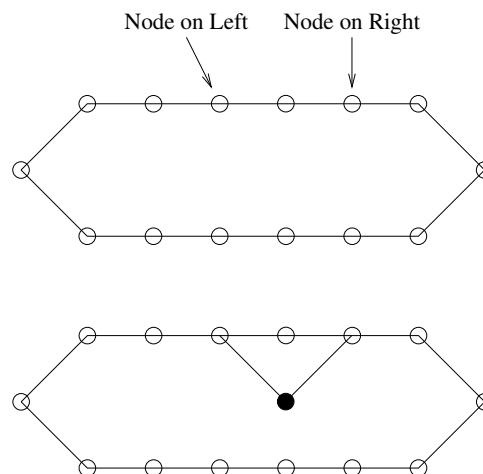
### 5.2.4 Node Insertion

Node insertion is the process of taking a new node and finding the best place in the graph for it. There are two forms that such insertion can take:

- Type 1 Insertion of new node between one or more pairs of nodes that are currently linked, thus splitting the links between each pair.
- Type 2 Insertion of new node between two or more unlinked nodes, creating new links from those nodes to the new node.



**Figure 5.6:** *Type 1 insertion*



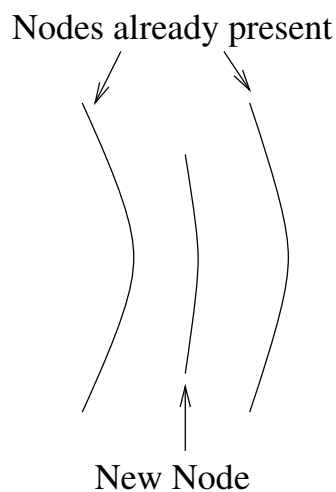
**Figure 5.7:** *Type 2 insertion*

A simple situation that would cause Type 1 insertion is illustrated in figure 5.8 and the effect that this has on the graph is shown in figure 5.6. Similarly a simple situation causing Type 2 insertion is illustrated in figure 5.9 and the effect on the graph is shown in figure 5.7. Note that the Type 2 insertion has increased the number of links from the nodes that were already in the graph.

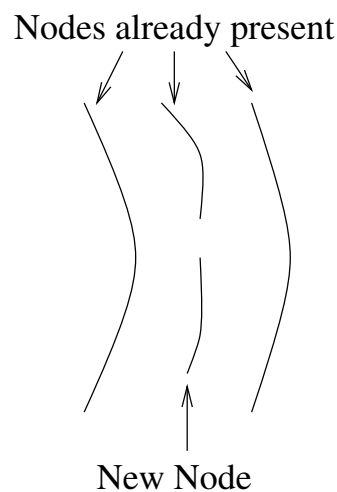
The graph is first searched for a Type 1 insertion and if no such situation is found, it is then searched for a Type 2 insertion (which is guaranteed to be found due to the presence of the artificial ‘first’ and ‘last’ nodes)

### *Type 1 Searching and Insertion*

The entire graph is searched to find all pairs of nodes such that the new node is to the right of the leftmost node of the pair and to the left of the rightmost node of the pair. If one or more such pairs is found, then the links between each pair are removed and the new node is linked between each pair. This allows for the slightly more complex insertion case than shown in figure 5.6, by allowing the removal of more than one link, as shown in figure 5.10. This type of insertion ensures that adjacent runs of nodes will be correctly created, where appropriate, regardless of the order in which the nodes are submitted for insertion. Figures 5.1 and 5.2 represent two such runs of nodes.




---

**Figure 5.8:** *Type 1 insertion situation*



---

**Figure 5.9:** *Type 2 insertion situation*

### *Type 2 Searching and Insertion*

This type of insertion is more complex as it involves the creation of new branches within the graph. The search process is in two parts and involves searching into the graph from one side and then searching into the graph from the other side.

Search part 1: search in from the left (from 'first') and locate all nodes which are to the left of the new node *and* whose 'next' nodes all have no adjacency at all with the new node. For the latter part of this test note that none of the 'next' nodes should have 'to the right of' adjacency with the new node as this would have been discovered in the Type 1 search and that although all the nodes to the left of the new node will have 'to the left of' adjacency with the test node, only the node nearest the new node should be linked to the new node.

Search part 2: search in from the right (from 'last') and locate all nodes which are to the right of the new node *and* whose 'prev' nodes all have no adjacency at all with the new node.

These two parts of the search will return one set of nodes that are immediately to the right of the new node and one set of nodes that are immediately to the left of the new node. The new node is then inserted and linked to these two sets of nodes. A simple example of this is shown in figure 5.11 where the short arrows show the nodes that have been selected by the two parts of the search.

Type 1 search Node Pairs

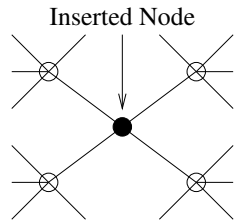
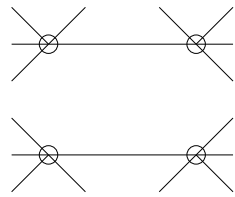


Figure 5.10: Type 1 insertion

Type 2 search Nodes

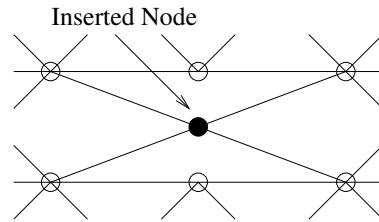
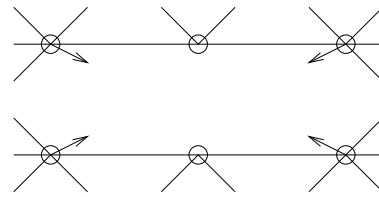


Figure 5.11: Type 2 insertion

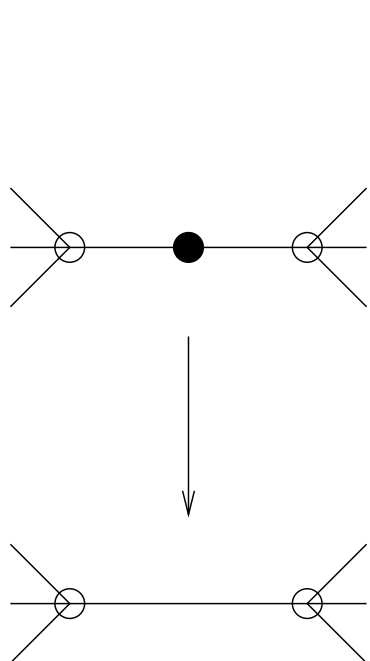
The implementation of the searching and insertion routines is discussed in section B.2.7.

### 5.2.5 Node Removal

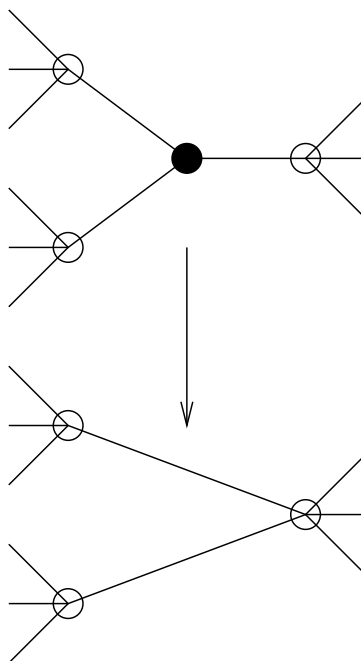
During any reconfiguration of the graph shape it will be necessary to remove nodes from the graph. In particular it is necessary for the operation of Graph Node Shuffling described in section 5.2.8. In certain circumstances node removal will not be possible as it will not be clear how the graph should be relinked across the vacancy, however in other circumstances it may be possible to just remove the node and its links, with no other reconfiguration necessary. This can be determined by number of 'prev' and 'next' nodes that the node has:

	'Prev' Nodes	'Next' Nodes	Removal Possible
1	1	1	Yes
2	>1	1	Yes
3	1	>1	Yes
4	>1	>1	No

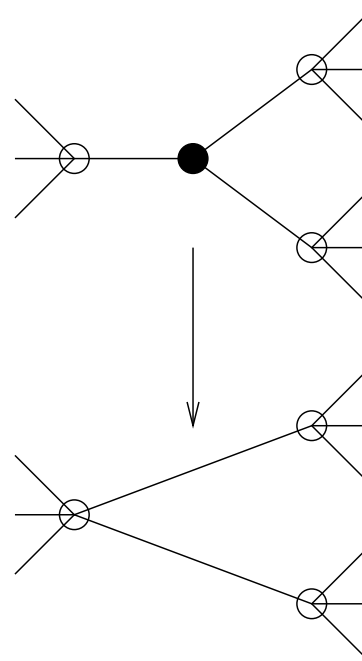
If, as in cases 1,2 and 3, removal is possible then relinking of the adjacent nodes may be necessary. If all of the 'prev' nodes from the node to be removed have 'next' nodes (not including the one to be removed) and all of the 'next' nodes from the one to be removed



**Figure 5.12:** *Case 1 Node Removal*



**Figure 5.13:** *Case 2 Node Removal*



**Figure 5.14:** *Case 3 Node Removal*

have ‘prev’ nodes (not including the one to be removed), then no relinking is necessary. If this condition is not true, then the nodes are relinked as shown in figures 5.12, 5.13 and 5.14. This can be summarised by stating that if the removal of the node will leave other nodes that cannot be reached by traversing using ‘next’s from the ‘first’ node or using ‘prev’s from the ‘last’ node, then the gap that has been created in the graph must be linked across as shown.

Note that this relinking may result in two nodes which are not adjacent to one another (e.g. the ‘to the left of’ and ‘to the right of’ operators would both return `False`) being linked as though they were. Although it is technically incorrect to allow the inclusion of such non-legitimate links, the alternative solution of recreating the graph from scratch whilst omitting the relevant node would present its own complications with regard to graph shape (see section 5.2.7). In practice the immediate re-insertion in to the graph of the node after its removal will result in the node being replaced back between the two nodes it was removed from between *and* may also result in it being placed between other nodes as well. For example, it may be re-inserted as in figure 5.6 *or* as in figure 5.10. It is the latter case which allows the reshaping of the graph by allowing all nodes a chance

to interact with all other nodes as described in sections 5.2.7 and 5.2.8.

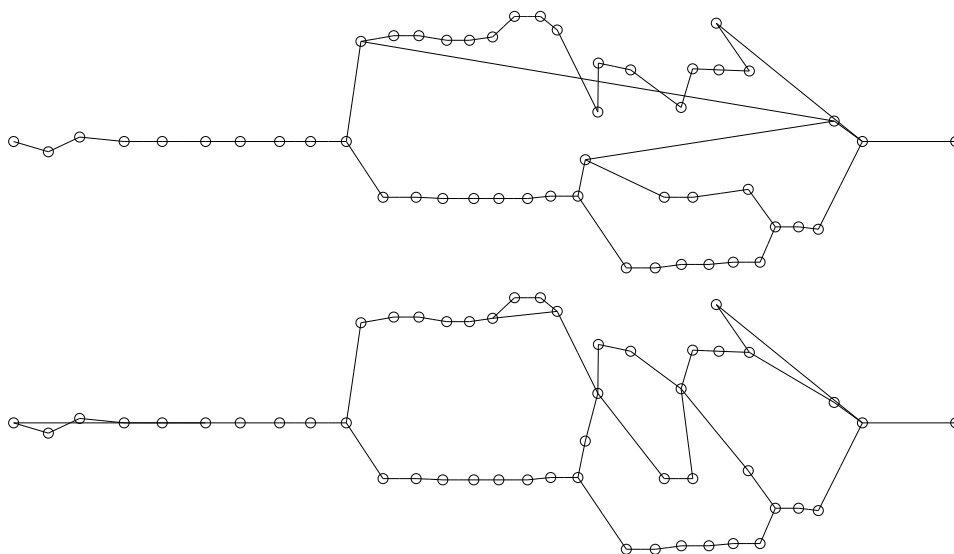
The implementation of the node removal routines is discussed in section B.2.7.

### 5.2.6 Graph Display

With traditional graph theory the placement of nodes when drawing the graph structure is often completely arbitrary and will often be adjusted to minimise the crossing of connections. In this case an obvious node placement strategy is to relate the node's physical position to that of the image edges that it represents. As the graph connectivity is supposed to represent the physical placement of the image edges with respect to each other, this would seem to be the simplest method of generating a picture of the graph. To that end, when the graph structures are drawn, the physical position of each node is the simple average of all the  $(x, y)$  coordinates in the image edges making up that node. For clarity, the x-axis is expanded to allow the inter-connections to be seen but it is generally clear which nodes represent which image edges when looking at the initial graph. As an example of this, compare the graphs in figure 5.15 with the data they were generated from in figure 5.23. Despite the different graph linking (explained in the next section) it is clear that the nodes at the top of the graph represent the image edges above the eye and nose and that the nodes at the bottom of the graph represent the image edges below the eye and nose. Obviously as nodes are merged, as described in section 5.2.9, some nodes will eventually represent several image edges and in this case the vertical placement of the node in the graph display is less significant but the horizontal positioning still clearly shows the 'to the left of' and 'to the right of' relationships. The implementation of this is discussed in the last section of this chapter and in appendix B.

### 5.2.7 Graph Shape

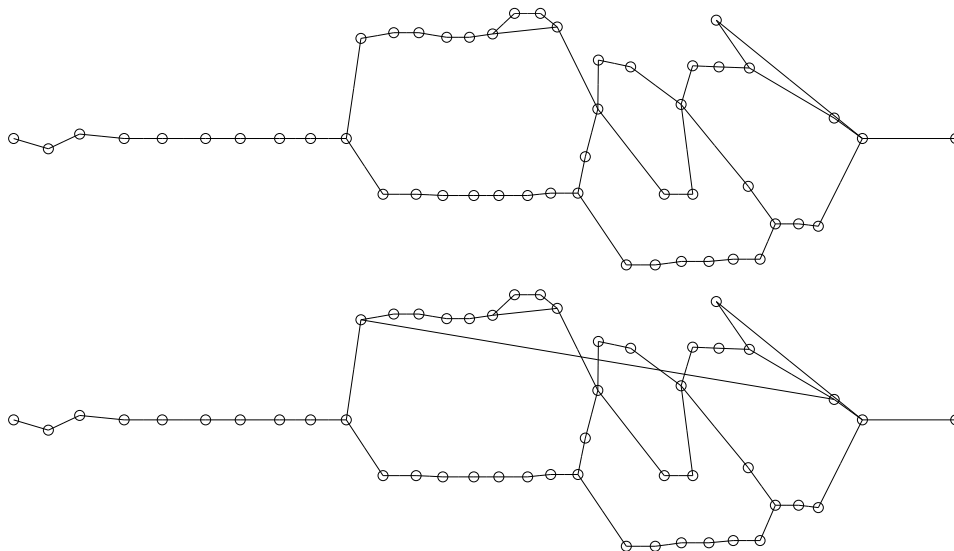
From the description of the node insertion method, it is clear that the order in which the nodes are presented for insertion will have an impact on the eventual shape of the graph. For example, in a situation in which a Type 1 insertion would be expected (e.g. a node being inserted directly between two other nodes), yet the outer pair of nodes are not yet present in the graph, the node would be inserted as a Type 2 insertion (e.g. branching) between other already present nodes. This will therefore cause a different branching structure to be created compared to the case where the Type 1 insertion was possible.



**Figure 5.15:** *Graph creation with differently ordered data*

An example of this graph shape difference is shown in figure 5.15 where the graphs were created using exactly the same image edge data (seen in figure 5.23), but presented for insertion in the opposite order. Note that, particularly on the right side of the graph, there is a substantial difference in the connectivity of the two graphs and that in the middle of the upper graph a key crossing over has not occurred compared to the lower graph. This would occur if that particular node was inserted early in the process, in which case it would be more likely to be inserted between a pair of nodes (as in figure 5.6) than between several pairs of nodes (as in figure 5.10), causing the crossover, as some of those nodes would not yet be present in the graph.

It would be hoped that this would not have any effect on the eventual aim of allowing the graph to be collapsed, as all such graphs are completely valid. The graphs were created according to the adjacency rules implied by the comparison operators thus each graph shape still contains all the necessary information for suitable collapse. However, as will become clear in the section on collapsing, it is desirable that the graph be as simple as possible and that nodes that are linked together should be of opposite image edge polarity. In particular the graph needs to show all the possible adjacencies that could be relevant to the collapse process and not just a valid but incomplete subset of them. Ideally, to generate such a graph every node needs the opportunity to interact with every



**Figure 5.16:** *Graphs from figure 5.15 after shuffling*

other node. This obviously cannot occur during the graph creation process as only the last node inserted can interact with all the others, although even in this case it is interacting with nodes that have not had the same opportunity.

It is thus useful to consider methods whereby any graph created from the same initial set of image edges but ordered differently, may be converted into the graph which best represents the supplied image edge data and in which all the nodes have had the opportunity to interact with all the others - a process termed Graph Node Shuffling

### 5.2.8 Graph Node Shuffling

The purpose of shuffling is twofold. Firstly, it is undesirable to have a graph creation process that produces different results depending on the order in which the image edges are initially presented. This order is obviously not random as it is a consequence of the image edge detection process described in the previous chapter but can be considered to be completely arbitrary. Secondly, it is important to form a graph in which each node can be said to have interacted with all the other nodes, thus generating a graph in which all the relevant adjacencies are shown. If this is not done there is a significant risk that there will be links present which are technically correct but join image edges which are so far apart that there is no possibility that they can be collapsed. This can cause a problem in

that the presence of this surplus link may prevent nodes being removed from the graph and thus stop the collapsing of nodes that would otherwise have occurred. There is also a risk that many useful adjacencies will be missing as the relevant interactions did not occur because of the order in which the nodes were first inserted into the graph. Thus many valid collapses will not be available as the necessary node interactions did not occur.

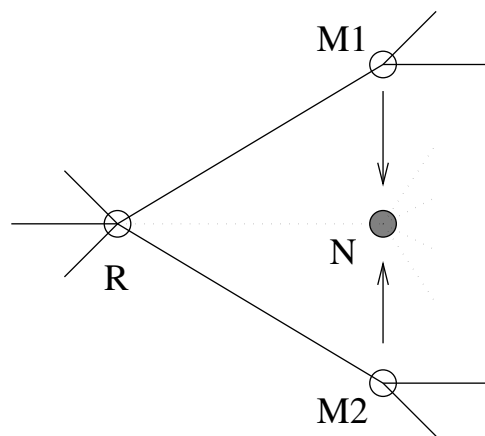
In order to resolve this problem, one solution would be to remove any nodes from the graph that appear to be ‘out of place’ and then re-insert them, with the re-insertion obviously then taking place in the presence of all the other nodes. Thus, subject to the fact that other nodes may also be ‘out of place’, the removed and re-inserted node should end up in a ‘better’ position. There are several difficulties with this solution, the most important being the identification of nodes that are ‘out of place’ - an inherently vague statement - or have not interacted sufficiently with the other nodes in the graph.

Another solution would be to remove and re-insert all of the nodes, one by one, on the basis that nodes that are in the ‘best’ place will return to that place but that nodes that are ‘out of place’ will be inserted elsewhere. Once again this solution has a problem in that the order in which the nodes are dealt with may have a significant bearing on the eventual shape of the graph. Also, a more elegant solution would not rely on external lists of nodes and would be based purely on information contained in the graph.

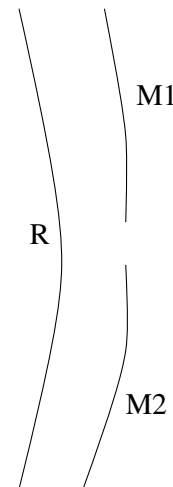
It is thus worth considering a removal/re-insertion order that has more relevance to the current shape of the graph and that the order in which the nodes are processed is related to their position within the graph.

The selected solution is to propagate through the graph from left to right and then right to left, removing and re-inserting all the nodes as they are traversed. Wherever there is a branch in the graph, a random choice is made as to which branch to take. Thus the removal/re-insertion process will occur along paths through the graph and the random nature of the branching ensures that there will be no systematic effect that a deterministic choice could have caused. It is thus impossible to ensure that all the nodes are traversed with this system but by making repeated traverses, there is a very high likelihood that this will be the case.

By doing this process a ‘few’ times (where ‘few’, from experimental observations is of the order of 20) it is observed that graphs that are created from different initial node insertion orders which therefore have a very different initial shape can be converted towards a consistent shape that is not dependent on initial node ordering. Figure 5.16 shows the effect of the node shuffling on the two graphs from figure 5.15. Note that, despite the



**Figure 5.17:** *Potential node collapse location in graph*



**Figure 5.18:** *Potential node collapse location - image edges*

initial very significant differences, there is only one difference between the two shuffled graphs. There will obviously be minor differences between shuffled graphs which could be reduced by further iterations of the shuffling process, but some variation will be inevitable due to the random nature of the traversal during the shuffling process and the fact that some nodes are not removable from the graph and cannot therefore be shuffled.

Thus, regardless of the initial node ordering, a consistent graph in which every node has had the opportunity to interact with all other nodes, can be presented to the Graph Collapsing process.

Graph node shuffling is implemented by routines discussed in section B.2.7.

### 5.2.9 Graph Collapsing

Graph collapsing consists of repeatedly sweeping through the graph and identifying nodes within the graph that could be merged together. Such nodes are identified in groups of three - two nodes that could be merged and a third reference node that is adjacent to both of the other two. This situation is illustrated in figures 5.17 and 5.18. Obviously the mirror image situation can occur with the reference node, R, being to the right of the merge nodes, M1 and M2.

*Node Collapsing*

Having identified the potential collapse locations in the graph purely on the basis of the presence of a ‘fork’ in the graph, the following tests are applied to check whether collapsing is actually possible. These tests can conveniently be separated into those tests that are related to the graph structure and those related to the image edges.

## Graph Structure Issues:

- Both nodes M1 and M2 must be removable from the graph (noting from section 5.2.5 that it is not always possible to remove particular nodes).
- Neither node may be reachable from the other by following ‘prev’ or ‘next’ links. This situation is shown in figure 5.19. In this example M1 is reachable from M2 by following the ‘prev’ links through node A and M2 is reachable from M1 by following the ‘next’ links through A. If M1 and M2 were allowed to merge then node A would point to the new node ‘N’ (following the labeling of figure 5.17) with a ‘next’ link *and* a ‘prev’ link - obviously an illegal graph configuration which would physically represent the combined image edges M1 and M2 ‘crossing’ the image edge A. The links, in this situation, would ‘correctly’ show that A was to the left of *and* to the right of the new node N. This case is tested for by following all possible ‘next’ links from M1 and ensuring that M2 is not encountered and then following all possible ‘prev’ links from M1 to ensure that M2 is not encountered on that side. This is a recursive function as all possible branches of the graph have to be followed in the relevant direction. This is illustrated in figure 5.20 in which the right hand graph shows mutually reachable potential collapse nodes and the left hand graph shows a mutually *unreachable* set of potential collapse nodes. The search path is shown with dotted arrows and clearly intersects M2 in the mutually reachable case which would therefore disqualify this set of potential collapse nodes.

## Image Edge Issues:

- Both nodes M1 and M2 must contain image edges of the *same* polarity (otherwise they could not possibly be part of the same projected edge).
- The polarity of the image edges in node R must be the *opposite* of nodes M1 and

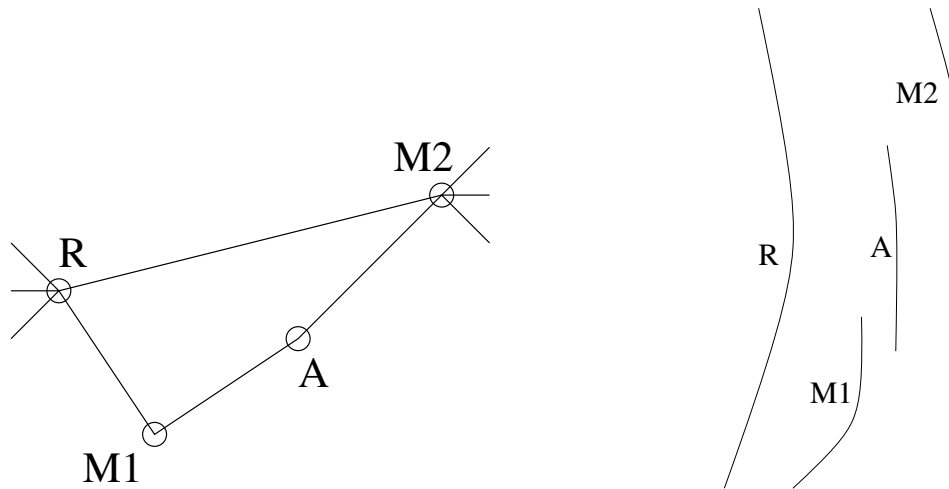


Figure 5.19: Node mutual reachability problem

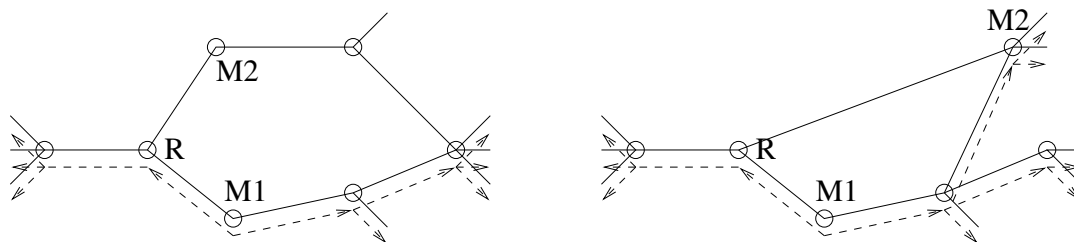


Figure 5.20: Mutual unreachability (left) and reachability (right)

M2 (the projected edges are of alternating polarity, therefore, ideally<sup>2</sup> the nodes in the graph should be too).

- The lowest point of the image edges in the upper node must be close enough in a *vertical* direction to the highest point of the image edges in the lower node.
- The lowest point of the image edges in the upper node must be close enough in a *horizontal* direction to the highest point of the image edges in the lower node.

If all the graph structure and image edge conditions are fulfilled then nodes M1 and M2 are removed from the graph and a new node N, is created which contains all the image

<sup>2</sup>in practice, as described in the introduction to section 5.2, this is not always possible - although in the case of node collapsing it is essential

edges of M1 and M2, has a single 'prev' pointer to R and 'next' pointers to all the 'next' nodes from M1 and M2. Obviously the 'next' pointers of R are updated to remove M1 and M2 and add N and the 'prev' pointers of N's 'next's are updated to remove M1 or M2 and add N. In the mirror image case of R being to the right of nodes M1 and M2, the above description is the same with 'next' and 'prev' interchanged.

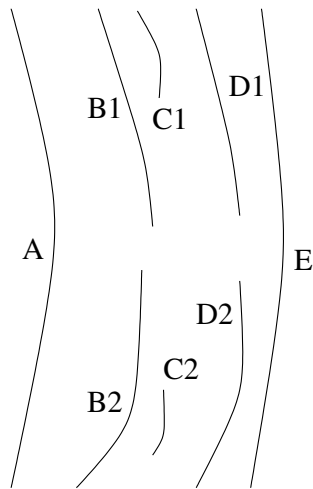
Note that this removal of two nodes and re-insertion of a single combined node is performed as a single step rather than as the separate removal of the two nodes (as described in section 5.2.5) followed by the re-insertion of the new node (as described in section 5.2.4). This prevents problems associated with the removal of two nodes that are 'one above the other' in the graph when non-legitimate links are formed by their removal, as described in section 5.2.5, allowing 'dangling' nodes to be created - nodes with either no 'prev's or no 'next's - which are obviously illegal in the context of the graph.

It is possible that there may be several nodes M1 to Mn that could be merged together. However, a process for merging them together in pairs has been described and repeated passes through the graph will allow any number of nodes to eventually be grouped into one node, if that is appropriate. Thus the nodes are sequentially accumulated into one node rather than attempting to capture all the relevant nodes at once. It is also worth noting that of the nodes M1 to Mn there may be some that should *not* be merged together. This situation would still be handled correctly by the sequential merging process, as these nodes would be offered as potential collapse candidates and rejected, so a branch in the graph would still exist.

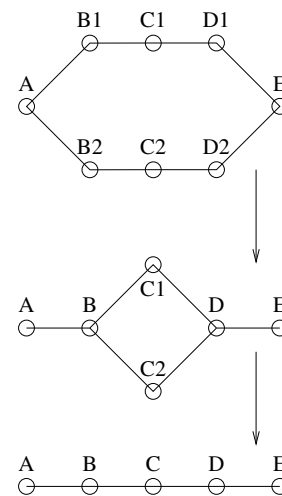
### *Graph Collapsing*

The graph is searched from left to right for node configurations such as figure 5.17 and these are merged, if appropriate. Then the graph is searched from right to left, searching for the reverse configuration and these are merged, if appropriate. As the topology of the graph has now changed, a shuffling process as described in section 5.2.8, is then performed. The justification for this is that essentially two nodes have been removed and replaced with a single node (representing a 'longer' image edge) which should be given the opportunity to interact with all the other nodes and vice versa. This is a similar situation to that of the initial graph creation and is dealt with using the same graph node shuffling procedure.

This sequence of left to right and right to left collapsing followed by shuffling is repeated until no more node collapsing occurs. Then the two conditions on image endpoint



**Figure 5.21:** *Widely separated image edges*



**Figure 5.22:** *Corresponding graph collapse iterations*

separation (the last two of the Image Edge Issues, above) are relaxed to allow image edges that are further apart to be merged together and the whole process is repeated again. This process continues until the graph becomes a single line or a predefined number of iterations is exceeded.

This gradual relaxing of the image edge separation limits is necessary, as can be seen from the experimental result, because there are occasions in which it is only after nearby node groups have been merged that it becomes clear that two vertically far apart image edges are actually part of the same projected edge. An artificial example of this situation is shown in figure 5.21 and the corresponding collapse sequence in figure 5.22. Note that in this example it would not initially be feasible to collapse nodes C1 and C2 as they are widely separated. However, once nodes B1 and B2 have been collapsed to form node B and C1 and C2 have formed node C, it is then acceptable to widen the limits on collapsing as nodes C1 and C2 are clearly defined as being between nodes B and D and the graph can be collapsed to a straight line.

A real example of this process starting with the upper graph in figure 5.15 is shown in figure 5.24. The image edges that this graph is created from are shown in figure 5.23 and were extracted from a reduced size camera image in order to make the graph clearer. In this example note the image edges above the break caused by the eye and eyebrow are typical of image edges that can be merged but only after the neighbouring image edges

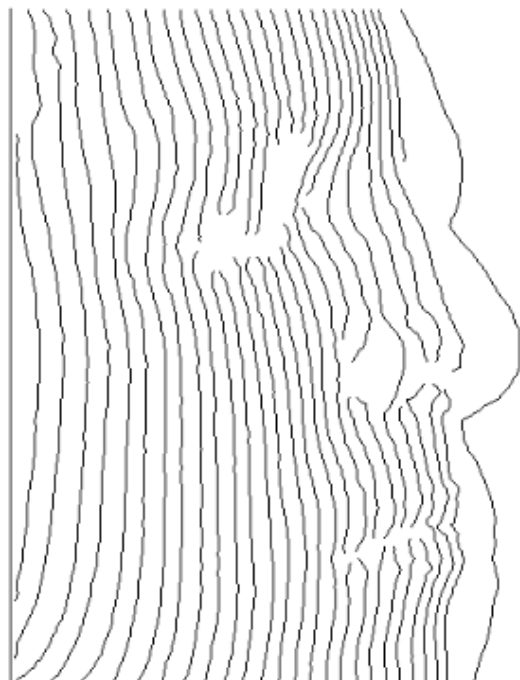
have been merged and the merging distance constraints have been relaxed, as described above. Eventually, even the rightmost image edges (not including the image edge caused by the edge of the face) can be merged. Note that merging of image edges does *not* imply that they will somehow be joined up. It merely records that they were created by the same projected edge. In the case of the rightmost image edges where one is on the forehead and one is on the chin, it would be impossible to determine how to join them together, yet the graph has revealed that they are caused by the same projected edge - a fact not immediately obvious by inspection of figure 5.23.

In the collapsing of the graph, as shown in figure 5.24, there are a number of ‘long’ connections such as that in the fourth stage. These are perfectly valid connections, but are not necessarily the ones that would be regarded as obvious by a human viewer. However, the whole intent of the graph based processing was to achieve something that was arguably ‘obvious’ by human standards but nevertheless difficult to implement with a computer - so it is to be expected that occasional non-obvious behaviour will be observed. In this case, these ‘long’ connections are created occasionally by the node shuffling process of section 5.2.8 as the random nature of the selection of nodes during this process implies that sometimes nodes will not relate to each other in the ‘obvious’ manner. However, these links will not necessarily prevent the successful collapsing of the graph for the following reasons:

- A potential collapse site or ‘fork’ can be approached from both sides. In stage 4 of figure 5.24, any approaches made from the right would still allow collapse to occur
- Later shuffling of the graph may remove the ‘long’ link in favour of the more ‘obvious’ one
- Collapses elsewhere in the graph followed by shuffling may change the circumstances that caused the ‘long’ link to occur
- The presence of links to nodes that are so ‘far away’ as to not represent viable collapsing opportunities does not prevent a node from collapsing with respect to others of its neighbours

Hence, in this example, the graph continues to collapse successfully to a linear list.

Graph and node collapsing is implemented by the routines discussed in sections B.2.6 and B.2.7.



**Figure 5.23:** *Image lines for graph in figure 5.24*

---

#### *Full Collapse Failure*

In the example of figures 5.23 and 5.24, a small part of an image was chosen that would create a set of image edges that would enable the adjacency graph to collapse fully to a single line. In practice this will not necessarily occur as they may be extraneous detected image edges that are created by features in the original camera image that are not projected edges. There may be missing or extra (not caused by a projected edge) image edges which break the alternation of image edge polarity (discussed in the introduction to section 5.2) and thus prevent particular sets of nodes collapsing. There may also be missing image edges that prevent the creation of a node that should have spanned the full image height and thus prevents the collapsing of the graph on either side of that node. In the circumstance of full collapse failure, the longest linear section of the graph is extracted and used on the basis that this is the most likely section of the graph to contain valid projected edges.

Figure 5.25 shows a set of image edges and the corresponding final graph structure,

that have this problem. In this case it can be seen that the problem at the right end of the graph is created by an incorrectly overlapping pair of image edges on the side of the nose and at the left end, the problem is created by a single image edge caused by the edge of the ear that cannot be matched to any of the image edges below it. In this example it is clear that taking the long linear sequence in the middle is the best solution.

### 5.2.10 Graph Processing Output

The output from the adjacency graph processing stage is a linear list of nodes going from left to right across the image. Each node consists of a list of one or more image edges, and the fact that they are grouped within a node implies that they are part of the same projected edge.

However, there may be circumstances in which a node may not represent a projected edge. This may occur due to other artifacts in the initial camera image and a common example is the prominent image edge caused by the silhouette of the face. This image edge can be seen as the rightmost line in both figures 5.25 and 5.23. If this image edge is assumed to be a projected edge then, after the 3D surface creation, there is likely to be a section of the surface that is in error. It is however, far easier to correct this mistake at that stage, using geometrical considerations rather than to attempt to check the list of nodes for image edges that seem erroneous.

### 5.2.11 Algorithm Implementation and Debugging

Obviously considerable programming effort was involved in the development of the techniques described in this (and the previous) chapter. This work is described in more detail in appendix B. However, a few details are worth noting here.

#### *Implementation*

The algorithms were coded using C++, a language highly suited to the handling of data objects such as those described in this chapter.

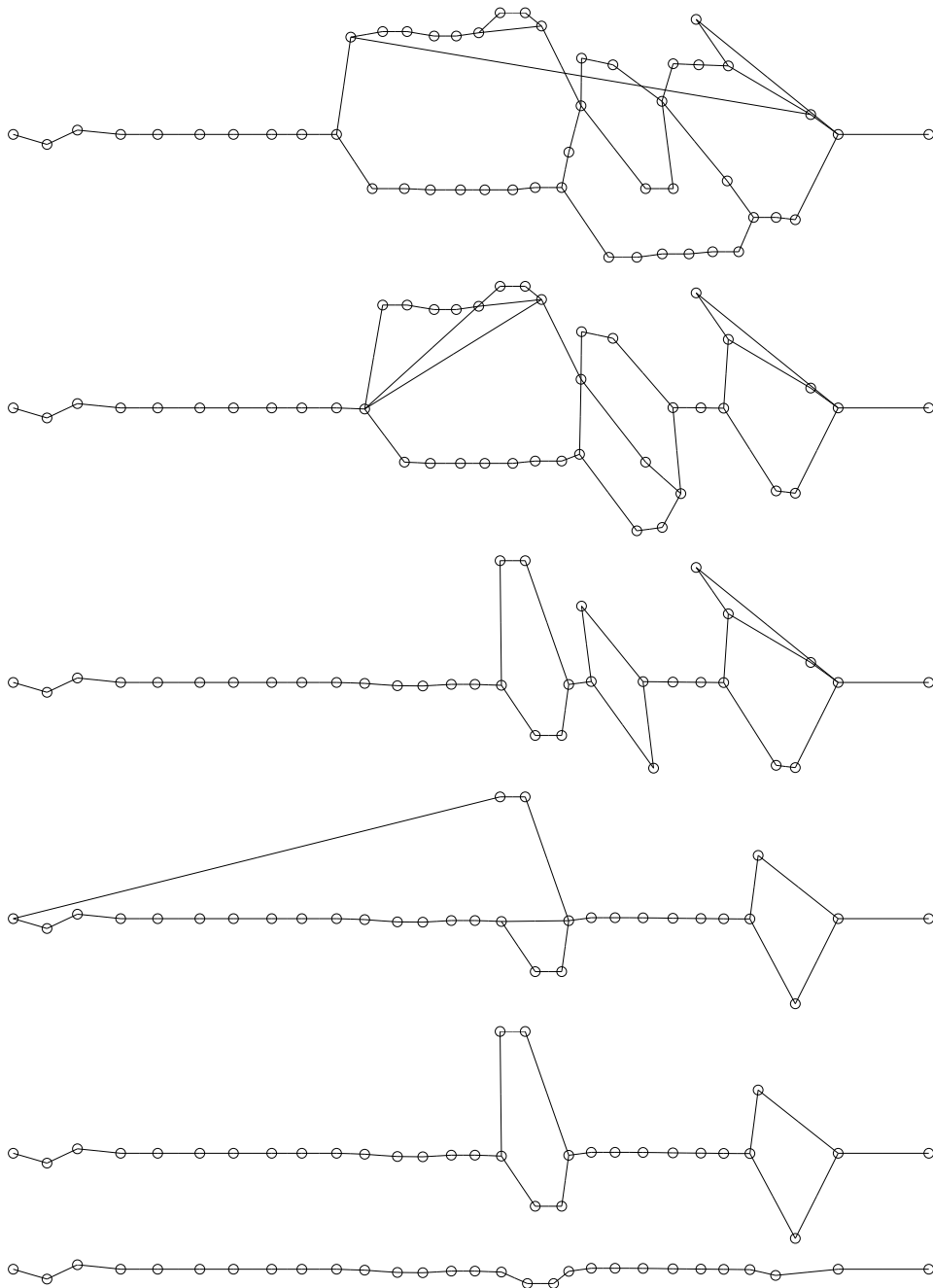
A number of display routines were developed in Matlab to display the graph structure at any stage in the processing in the manner used in the figures in this chapter. In practice, colour was used more extensively to show the image/projected edge polarity and to distinguish ‘next’ and ‘prev’ links. This was also the case with the routines to display the zero crossing lines and image edges.

*Debugging*

In the development of such a complex set of techniques as described in this chapter, mistakes are obviously made and the manner in which these are discovered and corrected is significant. This is discussed in greater detail in appendix B. However, the use of the colour Matlab display routines for the graph structure was most informative, particularly the displaying of ‘prev’ and ‘next’ links separately. i.e. it is better to display a ‘next’ link pointing from node A to node B and a ‘prev’ link pointing in the opposite direction, than to assume that the presence of one implies the other. Obviously this *should* be the case, but during development this type and other similar types of error were more rapidly traced and corrected through the use of the set of Matlab and C++ (described in appendix B) debugging and display routines that were written.

### 5.3 CONCLUSIONS

The techniques described in this chapter for determining the correct ordering and assembly of the projected edges from the image edges seem to overcome the main difficulty of using a single frame, structured light approach to determining three dimensional surface shape. In comparison with other systems using single images from structured light projection which are restricted to simple, smooth surfaces, this system represents a considerable advance in that it can successfully deal with more complex surfaces. It is obviously not infallible, but considering that all the information is derived from a single image, with all the convenience in hardware, processing and impact upon the human subjects that this implies, this novel process should prove useful in practical situations.



**Figure 5.24:** *Collapse iterations*

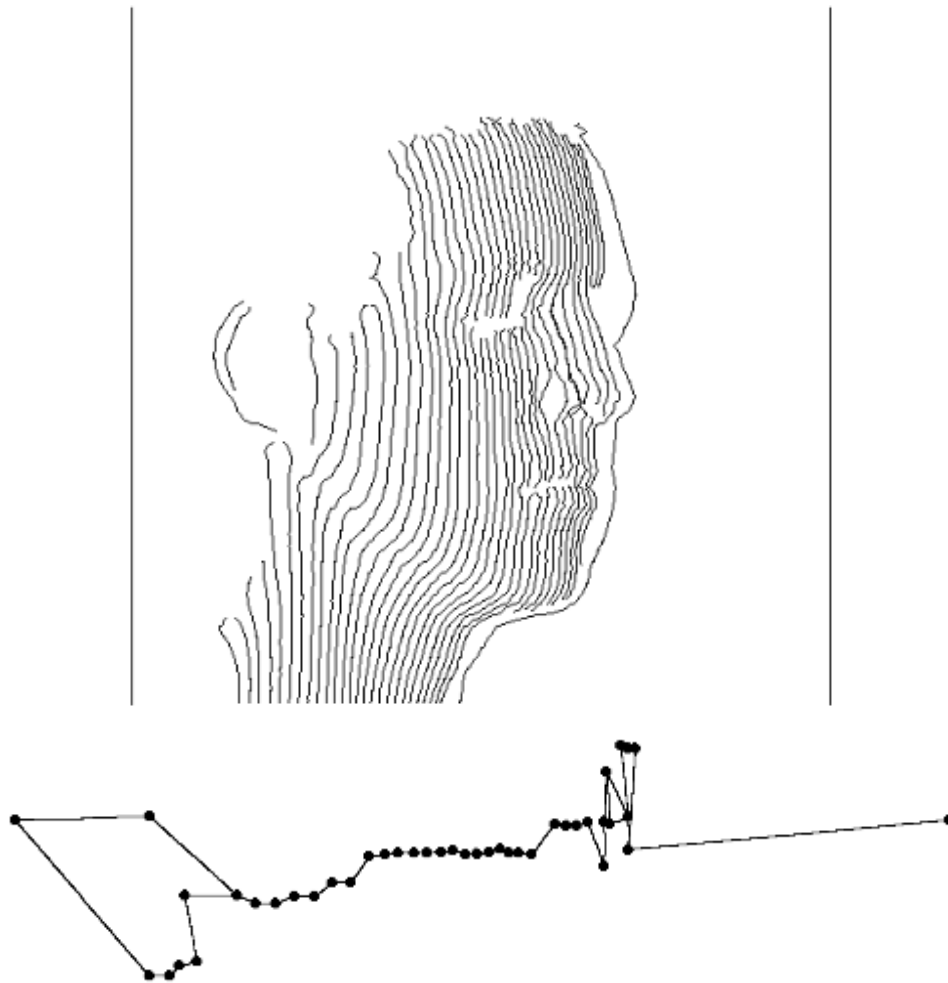


Figure 5.25: *A full collapse failure*

## Surface Generation

---

The adjacency graph processing output consists of a linear list of nodes, each node containing one or more image edges. These image edges are each a list of image coordinates and it is these image coordinates which need to be transformed into three dimensional coordinates. This can only be done if it is known which projected edge created the image edges in each node. The assumption for surface generation purposes is that each consecutive node was created by a consecutive projected edge.

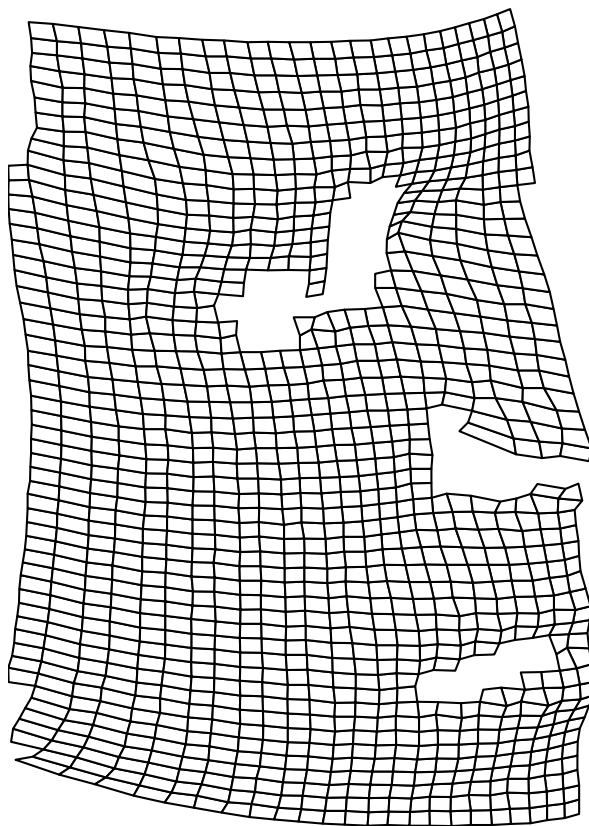
Occasionally, this will not be a correct assumption, for example, a complete projected edge may be missing in the node data or a strong image edge that is not caused by a projected edge may be present in the node data. In this case, a simple analysis of the curvature of the surface is used to determine which data nodes are invalid.

As mentioned in chapter 4 the original camera image may be a left or right face view. The techniques described in previous chapters are not affected by the sense of the image but, obviously, the surface creation process must be aware of which side of the face the camera was looking at. In general, the algorithm can simply be informed, per image, which side of the face it is, but in practice it is simple to deduce from the shape of the presented image edges.

Finally, where a double camera image was taken (both left and right views of the face) these two data sets can be combined, if possible, into a single surface. Up until this stage, both camera images would be treated separately to produce two surfaces. A simple volume error technique is used to align the two surfaces and fuse them together.

### 6.1 SURFACE CREATION

The surface data is stored in a manner to make the visualisation and display of it easier. Some surface representations involve the use of polygons or other more complex patch



**Figure 6.1:** *A mesh surface*

shapes, which are stored as list of vertices containing  $X, Y, Z$  data and a list of patches containing vertex data. In this case the most obvious manner in which to store the data is as a mesh. The projected edges traverse the image horizontally and the image edges are all sampled along horizontal lines. Thus the three dimensional data can be stored as three matrices (one for each coordinate) each with a number of columns equal to the number of projected edges and a number of rows equal to the number of horizontal lines in the image (the vertical resolution). Thus the surface is a set of quadrilateral patches formed by adjacent cells in the matrices. This type of mesh surface visualisation is shown in a significantly sub-sampled example in figure 6.1. If this mesh were viewed from ‘straight ahead’ then it would obviously appear as just a grid.

The routines implementing the algorithms described in this chapter are discussed further in section B.2.9.

### 6.1.1 The Surface Data Object

The surface data object needs to contain storage for the X,Y and Z values at each point in the mesh and, as will be shown later, storage for a surface normal vector and two curvature values at each point. By storing the data as such a set of matrices, each vertex is implicitly connected to the four adjacent vertices in the mesh thus preventing the need for a complex vertex connectivity list.

The Surface Data Object

Object Type	Object Name	Comment
integer	num_lines	number of projected edges
integer	ys	Vertical Resolution
VECTOR	$\mathbf{r}[\text{num\_lines}][\text{ys}]$	mesh of 3D vectors
VECTOR	$\mathbf{n}[\text{num\_lines}][\text{ys}]$	mesh of vertex unit normals
float	ka[num_lines][ys]	maximum principal curvature
float	kb[num_lines][ys]	minimum principal curvature

### 6.1.2 Surface Filling

The list of nodes created by the graph ordering process is traversed, where each node corresponds to a column in the surface data object mesh. Each image edge within the node is taken, one at a time and is used to fill in the relevant cells within the column corresponding to that node. The three dimensional conversion equations are based on the geometric situation shown in plan view in figure 6.2 and are as follows:

$$r_x = n \times \text{PhysicalSpacing} = X_n \quad (6.1)$$

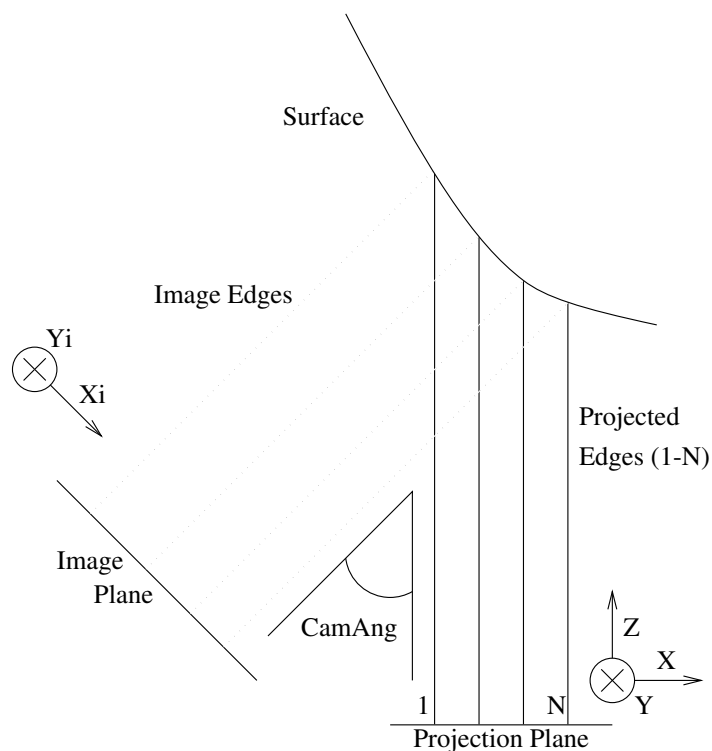
$$r_y = Yi \quad (6.2)$$

$$r_z = \frac{X_n}{\tan(\text{CamAngle})} - \frac{Xi}{\sin(\text{CamAng})} \quad (6.3)$$

where  $n$  refers to the node number (from left to right) and hence to the column number in the mesh, PhysicalSpacing to the distance between projected edges and CamAngle to the angle of the camera with respect to the projected edges.

Each set of coordinates  $Xi, Yi$ , in the image, represents a line in 3D space which intersects with the projected edge  $n$ , defined by  $X = X_n$ , at the point given by  $\mathbf{r}$ .

The origin of the coordinate systems is completely arbitrary as all the significant measurements are relative. However, the origin of the image coordinate system used by

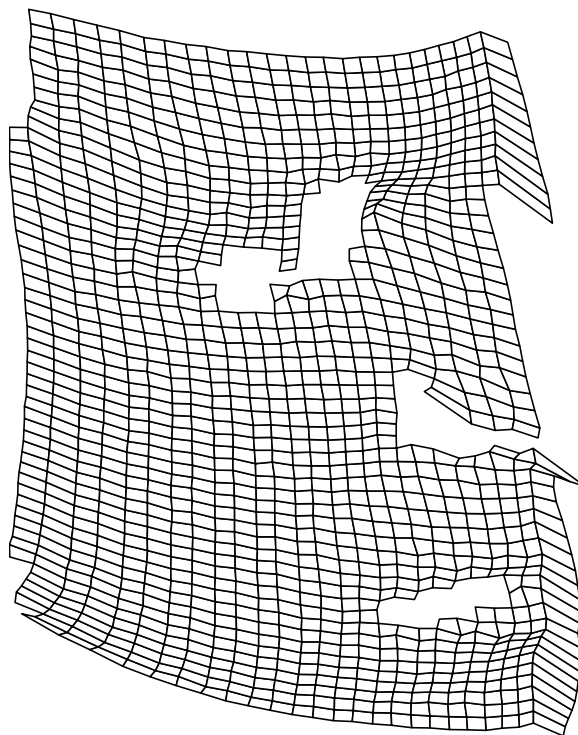


**Figure 6.2:** *Projection geometry*

$X_i, Y_i$  corresponds to the upper left corner of the camera image and the origin of the 3D space coordinate system used by  $\mathbf{r}$  corresponds to the intersection of the leftmost detected projected edge (a plane) with the line representing the upper left corner of the camera image.

The assumptions made at this stage are that the projected edges can be viewed as parallel planes and that the 'image rays' into the camera are also parallel. Obviously this is not the case in practice and a refinement of this would be to include a pinhole camera model. However, for a basic demonstration of the techniques involved, the parallel model is more than adequate.

The sign of the camera angle (CamAng) describes whether the camera has a view of the left or right of the face. This can be automatically determined by taking the average curvature of every image edge within the supplied set of projected edges and summing them up. The sign of the resultant total describes whether the image edges are predominantly concave from the left or right. Given the very approximately spherical



**Figure 6.3:** *A mesh containing an erroneous strip on the right*

---

shape of the human face, this simple test can deduce which side of the face is shown in the camera image.

### 6.1.3 Erroneous Surface Regions

As all the nodes are assumed to represent projected edges in sequence, then any nodes that occur from a different source (such as the rightmost line, caused by the face profile in figure 5.23 ) will create regions of the surface which are invalid. Using the assumption that, in practice these image edges will tend to be significantly separated, in the image, from the valid image edges, it is clear that those areas will create an obvious fold in the surface and there will be a line of high curvature running vertically down the surface in that section, as shown by the rightmost strip of the subsampled surface mesh in figure 6.3. Thus if the principal curvatures are calculated at each vertex in the mesh, then erroneous image edges will create columns of curvature in the mesh of unusually high average value.

### 6.1.4 Surface Curvature

A further advantage of using a rectangular mesh to store the surface data is apparent when the surface curvatures are calculated as described in many basic texts on three dimensional surface geometry[15]. The standard method for calculating principal curvatures of a surface using the first and second fundamental matrices, assumes the surface is parameterised in two dimensions. The mesh is obviously already parameterised by row ( $v$ ) and column ( $u$ ). The first fundamental matrix  $G$  and second fundamental matrix  $D$  are calculated at each valid vertex  $\mathbf{r}$ , in the mesh:

$$G = \begin{bmatrix} \frac{\partial \mathbf{r}}{\partial u} \cdot \frac{\partial \mathbf{r}}{\partial u} & \frac{\partial \mathbf{r}}{\partial u} \cdot \frac{\partial \mathbf{r}}{\partial v} \\ \frac{\partial \mathbf{r}}{\partial v} \cdot \frac{\partial \mathbf{r}}{\partial u} & \frac{\partial \mathbf{r}}{\partial v} \cdot \frac{\partial \mathbf{r}}{\partial v} \end{bmatrix} \quad (6.4)$$

$$D = \begin{bmatrix} \mathbf{n} \cdot \frac{\partial^2 \mathbf{r}}{\partial u^2} & \mathbf{n} \cdot \frac{\partial^2 \mathbf{r}}{\partial u \partial v} \\ \mathbf{n} \cdot \frac{\partial^2 \mathbf{r}}{\partial v \partial u} & \mathbf{n} \cdot \frac{\partial^2 \mathbf{r}}{\partial v^2} \end{bmatrix} \quad (6.5)$$

where  $\mathbf{n}$  is the unit normal at each valid vertex,  $\mathbf{r}$ . The values of the partial derivatives are calculated approximately from the values of  $\mathbf{r}$  in the mesh, for example:

$$\frac{\partial \mathbf{r}}{\partial u} = \frac{\mathbf{r}[n+1][m] - \mathbf{r}[n-1][m]}{2} \quad (6.6)$$

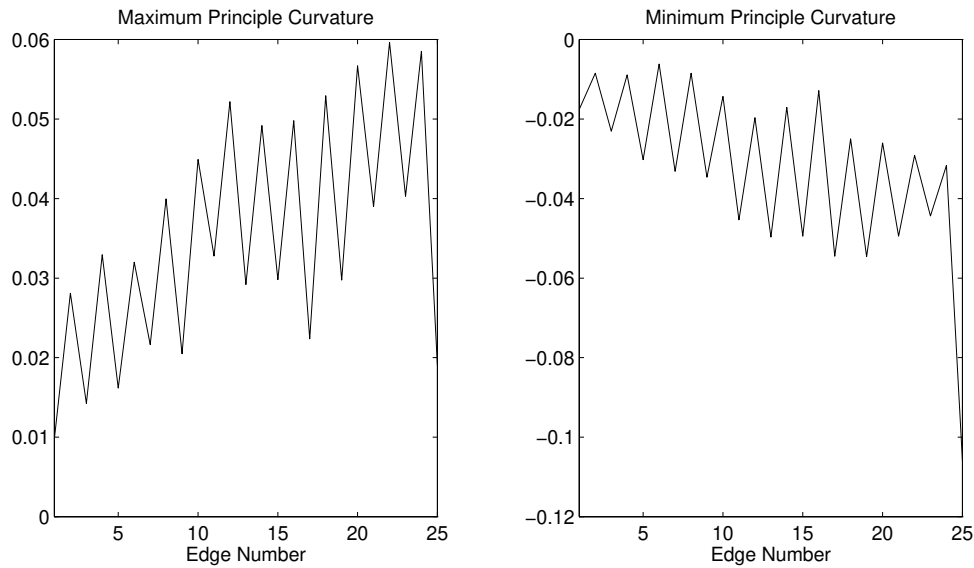
$$\frac{\partial^2 \mathbf{r}}{\partial v^2} = \mathbf{r}[n][m+1] - 2\mathbf{r}[n][m] + \mathbf{r}[n][m-1] \quad (6.7)$$

and it is the cross product of the first partial derivatives that is used to calculate the surface normal,  $\mathbf{n}$  at each vertex. From this step it is clear that to calculate any of the partial derivatives for a vertex, it must have neighbours in the appropriate directions. For this reason the leftmost and rightmost columns in the mesh will not have curvature values but will merely contribute to the curvature values of the adjacent columns. Note that the first derivative is taken over three points as a two point derivative would produce results between the sample points. The second derivative based on three points produces results on the sample points without further modification.

The principal curvatures, which are the maximum and minimum curvatures,  $ka$  and  $kb$  are the two solutions of the following equation:

$$|G|k^2 - (g_{11}d_{22} + d_{11}g_{22} - 2g_{12}d_{12})k + |D| = 0 \quad (6.8)$$

It is also possible, although in this case not necessary, to calculate the directions of the principal curvatures from the fundamental matrices,  $G$  and  $D$ . The two solutions are

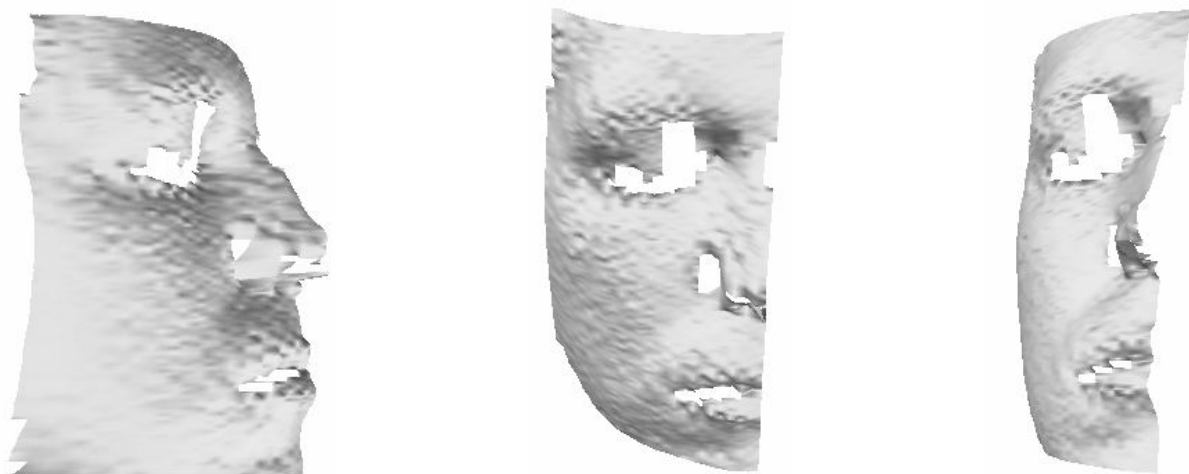


**Figure 6.4:** *Principal curvature values averaged over each projected edge*

calculated and stored in the surface data object at each valid vertex. If an average is taken of the values of  $ka$  and  $kb$  down each column then columns of high average curvature can be selected. Note that these columns will be *next* to the columns that should be removed. For example, the data in figure 6.3 (which was created from the data in figure 5.23 ) would show a high value of average curvature in the first column in from the right, therefore showing that the rightmost column contains suspect data. Figure 6.4 shows these average principal curvatures for the data in figure 6.3.

From the maximum principal curvature graph it is possible to see that edge 25 is clearly unusual and on this simple thresholding basis, the edge next to it (edge 26) will therefore be removed from the mesh. As the process of differentiating increases the high frequency noise present in a signal, it is unlikely that any useful accurate measurements could be made with the curvature values, although they are more than adequate for the rejection of columns of mesh data.

It is also interesting to note the periodicity present in both the average principal curvature graphs. This is because the projected light stripes were slightly wider than the dark stripes due to problems in manufacturing a suitable slide that could be used in the slide projector.



**Figure 6.5:** *Three views of the surface created from figure 5.24*

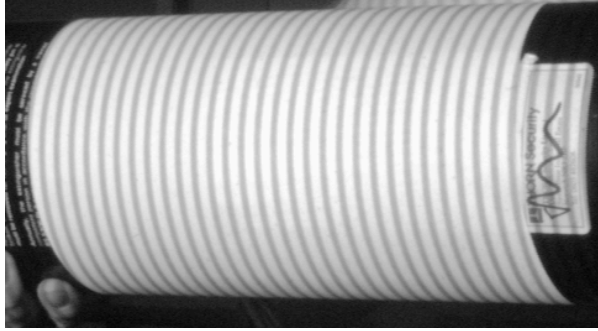
---

## 6.2 SURFACE DISPLAY

Displaying a surface such as this is a reasonably simple process. The calculation of the two dimensional projection of the three dimensional data onto a screen is a standard computer graphics problem, as is the issue of modelling the surface reflectance properties. Suitable algorithms are described in such books as ‘Computer Graphics, Principles and Practice’ [16] and other related books on three dimensional surface rendering.

In these examples the surface shading is based on a mixture of ambient, diffuse and specular lighting models. Ambient lighting represents uniform illumination from all directions, similar to the illumination that a bright sky would create. Diffuse reflection assumes a light source at a particular point and the brightness of a point on the surface is given by the cosine of the angle between the surface normal and the vector from that point on the surface to the light, which generates ‘matt’ surfaces. Specular reflection assumes that the surface will have a certain amount of mirror-like reflectivity and will thus reflect the image of any light sources present.

The calculations for the intensity of each point on the surface use the normal vector of the surface at that point, hence its inclusion in the surface data object. The rendering was performed using the OpenGL graphics library on a Silicon Graphics workstation.



**Figure 6.6:** *The camera image of the cylinder*

This is a three dimensional graphics library developed by Silicon Graphics specifically for the rendering of 3D objects and surfaces. The library has been implemented for many platforms in both hardware and software and is fully described in its reference manual [4].

The surface generated by the reduced size data of figure 5.23 is shown as an example in figure 6.5.

### 6.2.1 Testing

A number of calibration objects were used to ensure that the surface display was providing accurate information. The test cylinder, shown in figure 6.10 extracted from the camera image of figure 6.6, is a 60.5mm radius metal cylinder (a fire extinguisher). The surface data was imported into Matlab and a mean squared error best fit analysis was made of the data points,  $\mathbf{r}_i$ , using the scheme illustrated in figure 6.7:

$$\mathbf{a}_i = \mathbf{r}_i - \mathbf{p} \quad (6.9)$$

$$\text{minimise } S = \sum_i (|\mathbf{a}_i \times \hat{\mathbf{n}}| - R)^2 \quad (6.10)$$

$$\text{err}_i = (|\mathbf{a}_i \times \hat{\mathbf{n}}| - R) \quad (6.11)$$

where  $\hat{\mathbf{n}}$  is a unit vector along the axis,  $\mathbf{p}$  is a vector to a point on the axis and  $R$  is the radius of the cylinder.

If the variable  $S$  is minimised numerically with respect to  $\hat{\mathbf{n}}, \mathbf{p}$  and  $R$  then the error,  $\text{err}_i$ , between every point and the matched cylinder can be determined. In practice, one component of  $\mathbf{p}$  and one component of  $\hat{\mathbf{n}}$  must be fixed to ensure convergence as this is a

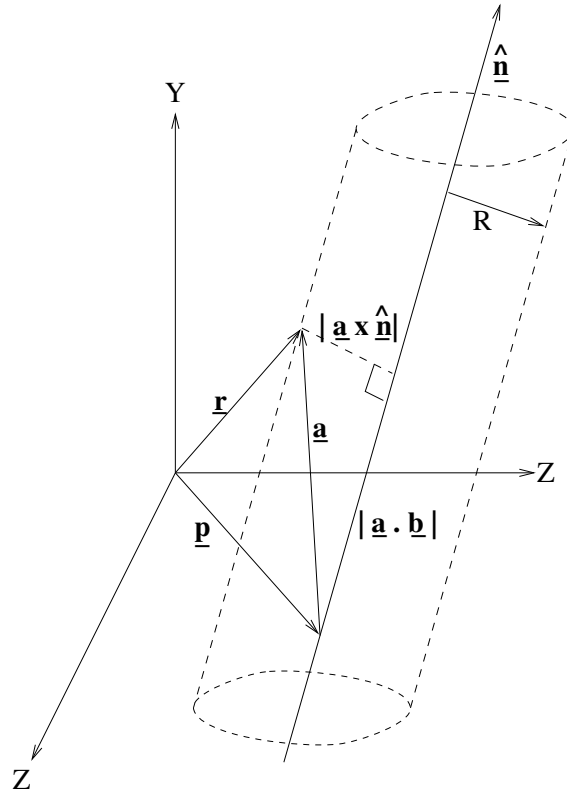


Figure 6.7: Fitting a cylinder to data  $\mathbf{r}_i$

5 degree of freedom system. A histogram of the errors,  $err_i$ , is shown in figure 6.8. These results represent a standard deviation error of 3.022mm to the nearest point on the ideal cylinder.

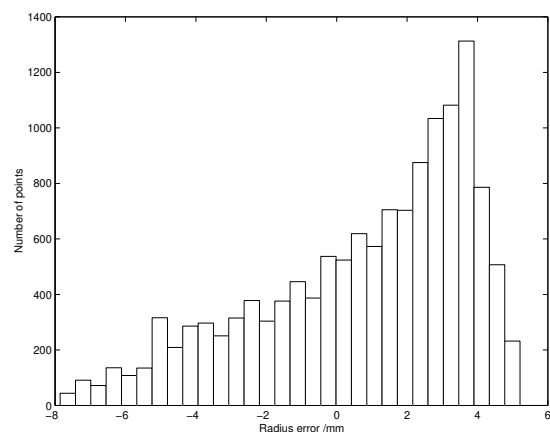
However, the shape of the histogram is not Gaussian and thus suggests that there might be a systematic error. Closer inspection of the point data,  $\mathbf{r}_i$ , reveals that the cylinder is actually slightly curved. This is a consequence of the parallel plane approximations used in section 6.1.2. It is therefore reasonable to introduce such a ‘bow’ into the shape model to compensate for this approximation:

$$\mathbf{a}_i = \mathbf{r}_i - \mathbf{p} \quad (6.12)$$

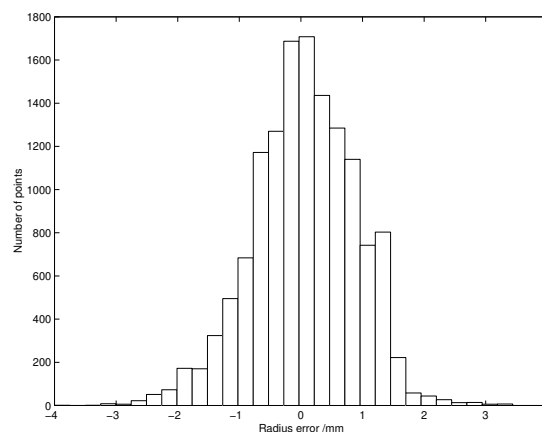
$$\text{minimise } S = \sum_i (|\mathbf{a}_i \times \hat{\mathbf{n}}| - (\mathbf{a}_i \cdot \hat{\mathbf{n}})^2 C - R)^2 \quad (6.13)$$

$$err_i = (|\mathbf{a}_i \times \hat{\mathbf{n}}| - (\mathbf{a}_i \cdot \hat{\mathbf{n}})^2 C - R) \quad (6.14)$$

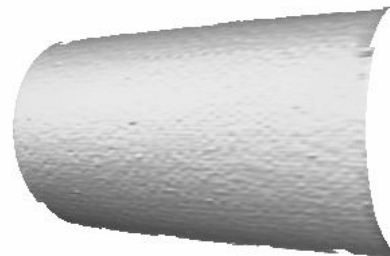
where  $C$  is the curvature correction factor and the other variables are as before. If the



**Figure 6.8:** *Uncorrected radius error histogram*



**Figure 6.9:** *Corrected radius error histogram*



**Figure 6.10:** *The test cylinder*

variable  $S$  is minimised numerically with respect to  $\hat{\mathbf{n}}, \mathbf{p}, R$  and  $C$  then the new errors,  $err_i$ , between each point,  $\mathbf{r}_i$ , and the ideal surface can be calculated. In this case only one component of  $\hat{\mathbf{n}}$  must be normalised to create a 7 degree of freedom system and  $\mathbf{p}$  will converge to the point on the ‘cylinder’ axis with the largest radius which will be given by  $R$ . A histogram of the error values is given in figure 6.9. This histogram is much closer to the expected Gaussian profile and represents a standard deviation of 0.839mm. The curvature correction factor,  $C$ , converges to  $-8.178 \times 10^{-4}$ .

These results are from an image of a 60.5mm radius cylinder which was at a distance of 1100mm from the camera. At this distance the camera’s field of view was approximately 400mm.

## 6.3 SURFACE COMBINATION

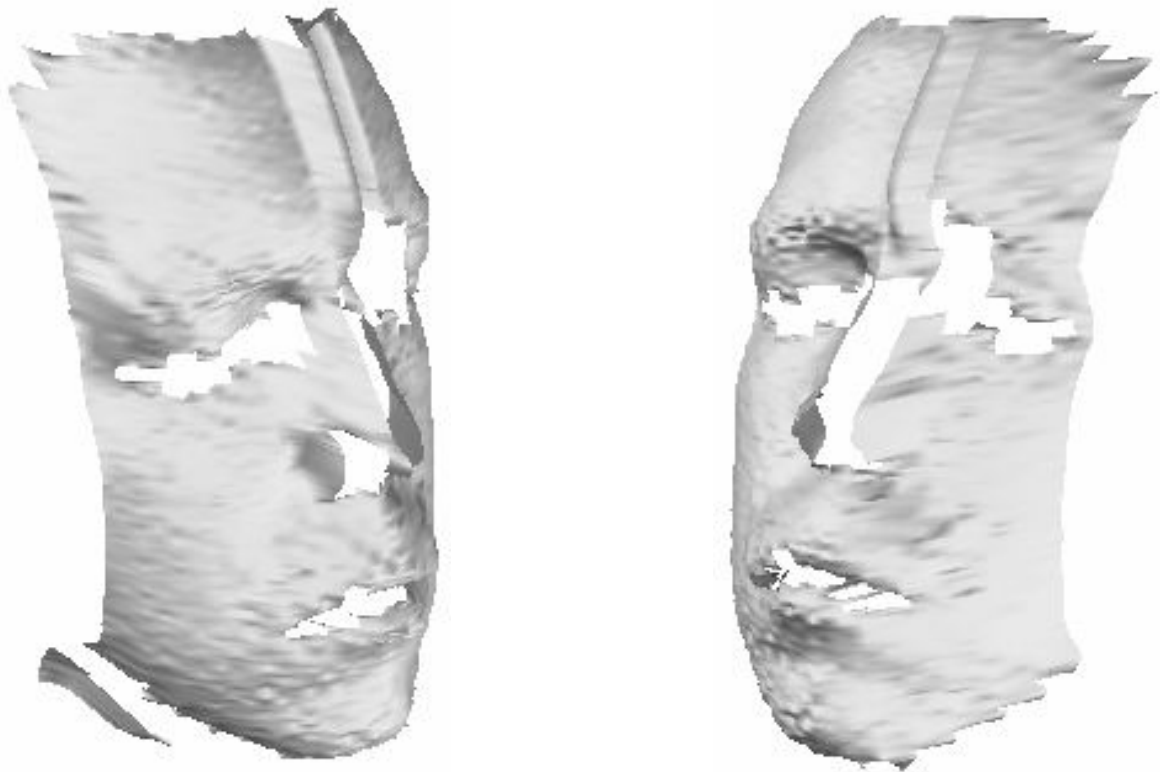
If two cameras were used to simultaneously acquire left and right views of the subject face, then it is necessary to combine the two surfaces together. Essentially this is a matter of searching all possible orientations and locations of one of the surfaces with respect to the other and selecting the orientation and location that, in some sense, provides a ‘best fit’. Potentially this task may also involve scaling or skewing one of the surfaces with respect to the other.

This potentially computationally expensive, multidimensional search task is simplified for the following reasons:

- the cameras used have very low values of CCD skew and are extremely similar in all respects.
- the camera lenses are also extremely similar and of high quality.
- the search locations in the X direction are limited to a set of discrete intervals as information is only gathered on the projected line edges.
- the Z dimension values of the surface data have arbitrary mean therefore the search can assume that the Z position of the surfaces is selected to set the mean Z dimension difference to zero.

This means that the search is reduced to two dimensions. Visualising the two surfaces as a mesh (as in figure 6.1), one of the search dimensions is to step one mesh horizontally across the other (as the vertical mesh lines must line up this must be a set of discrete displacements) and the other is to slide one mesh up and down the other. At each position, the mean difference in the Z values of the overlapping regions of the surface is set to zero and the error is recorded as the mean squared difference of Z values in the overlapping region. The offset for the lowest mean squared difference position is used as the offset to combine the two surfaces together.

The combined surface is created by filling a new surface object with both sets of surface data, where one set is appropriately offset. Where the two surfaces overlap, the Z dimension data is averaged between the two surfaces. An example of this is shown in figure 6.11.



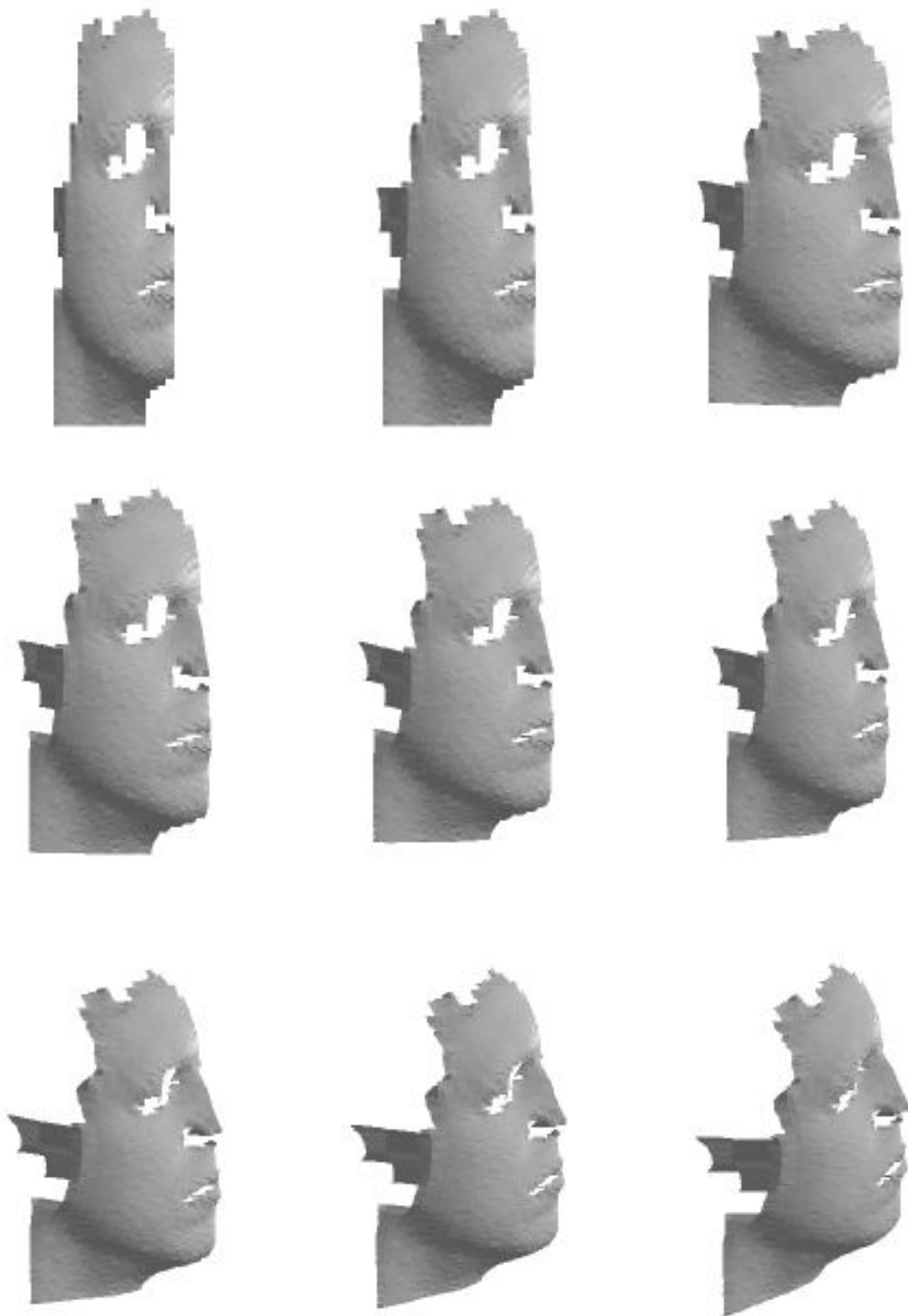
**Figure 6.11:** *Two views of a combined surface*

---

## 6.4 CONCLUSIONS

An example surface of the authors face is shown in figure 6.12, showing the surface from different angles in order to demonstrate the true three dimensional nature of the surface information. Each surface, in this example, uses a 512 by 34 mesh containing over 5,800 vertices.

It is important to remember that all of the information used to generate this fully three dimensional surface was contained within the single image shown in figure 4.4, in spite of the difficulties of surface discontinuities, surface colouration and feature ambiguity discussed in earlier chapters.



**Figure 6.12:** *A surface of the author's face viewed from different positions*

---

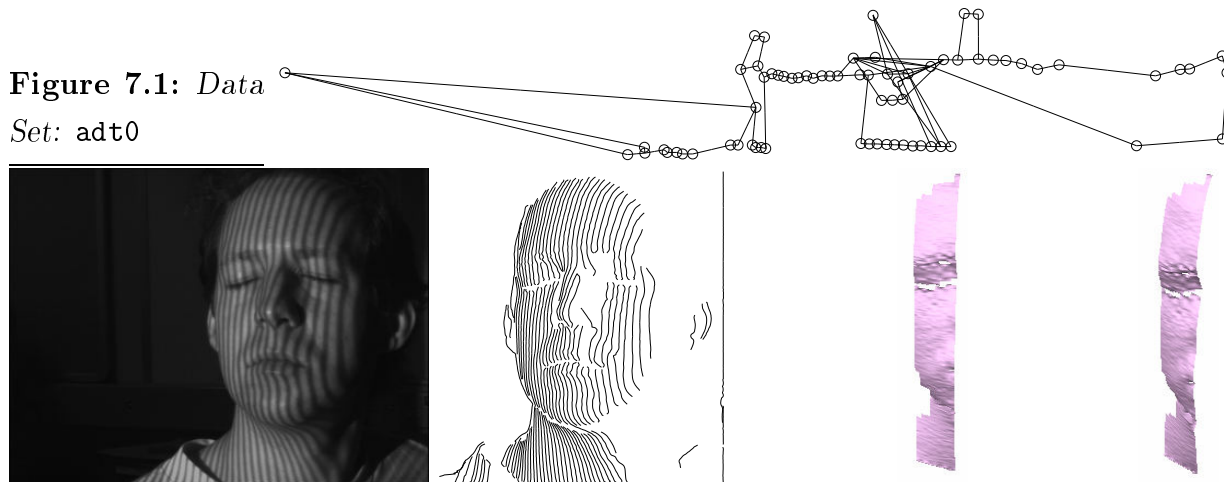
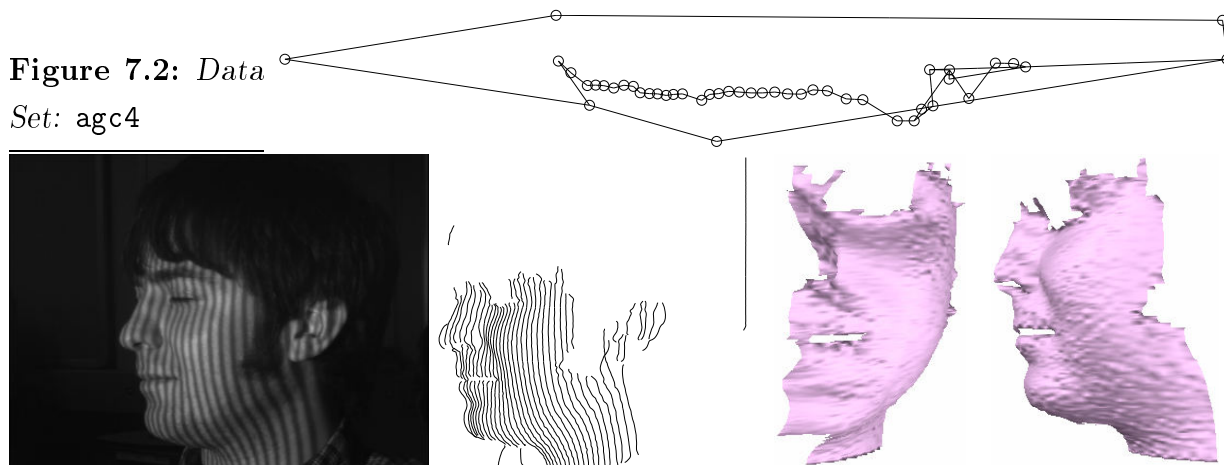
## 7.1 INTRODUCTION

This dissertation has dealt with the issues involved in developing a useful, practical system for ‘real world’ identification and recognition of human faces and has discussed the following:

- The benefits of three dimensional versus two dimensional techniques
- The issues affecting stereo vision
- The process of image feature extraction
- The process of image feature and projected feature correspondence
- The generation of the three dimensional surface
- The display of such a surface

The research so far has demonstrated the following:

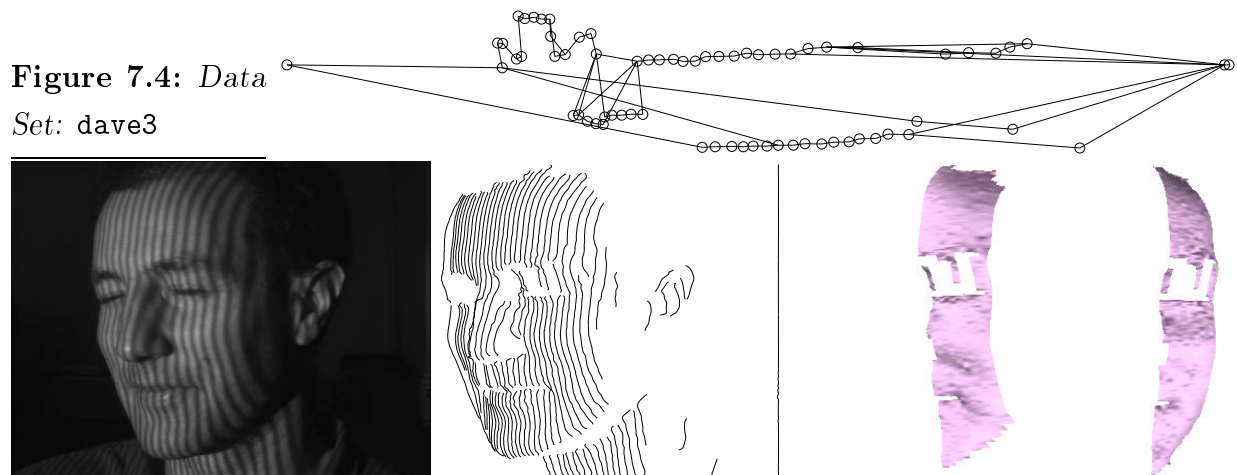
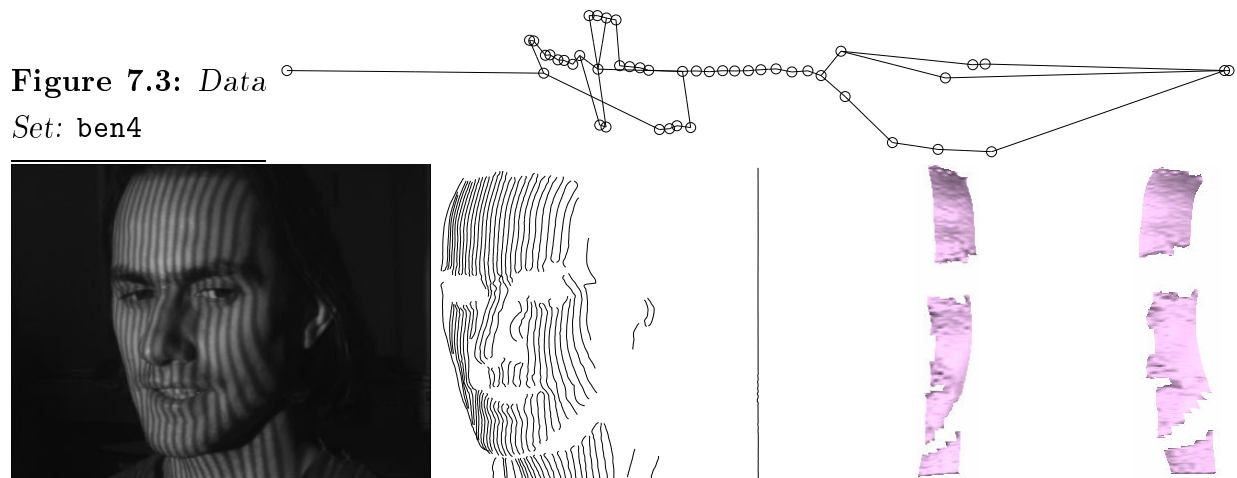
- A simple physical equipment set-up to acquire structured light stereo images
- A robust technique for extracting image features from the camera image
- A novel and robust technique for resolving feature correspondence ambiguity using the adjacency graph
- A simple process of surface generation and display

**Figure 7.1:** *Data**Set: adt0***Figure 7.2:** *Data**Set: agc4*

Possible further work would be to address the issues of recognition and identification, although it should be noted that in the area of three dimensional computer vision, this system has already demonstrated the extraction of a complex surface shape from a single stereo view, using a novel method to resolve the stereo matching ambiguities.

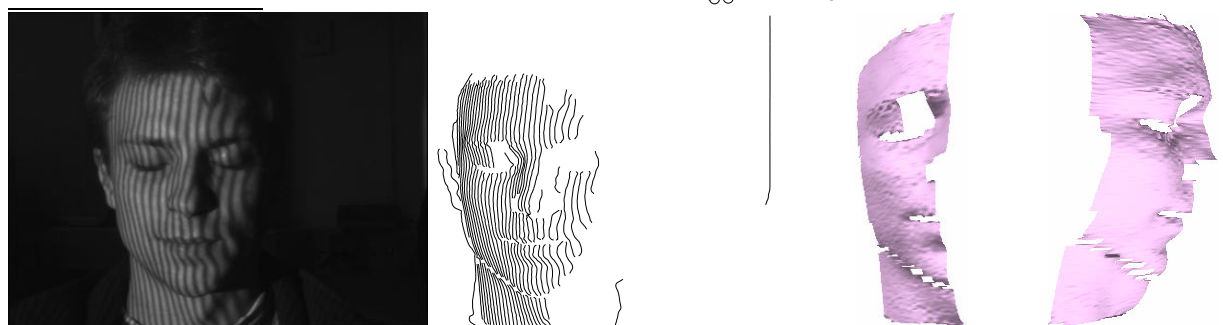
### 7.1.1 Results

Figures 7.1 to 7.16 show a number of typical results from the software. In each case five similar views were taken of the subject and the best result is displayed. Each figure consists of: the original camera image; the extracted lines; the final graph collapse state (all of which show full collapse failure); and two views of the 3D surface.



### Comments

- Even where the line extraction process has failed to identify lines or has made an error (such as allowing two lines to cross) the graph process still correctly determines the largest valid section of surface that can be extracted
- Non-optimal performance is generally caused by the non-ideal behaviour of the edge detector. For example, it allows image edges to touch and cross over which immediately causes sections of the resultant graph to be uncollapsible.
- The success of the graph algorithm, even with the non-ideal image edge data, would suggest that a simple method of improvement would be to make less effort to combine the zero crossing lines into unbroken image edges and thus reduce the number

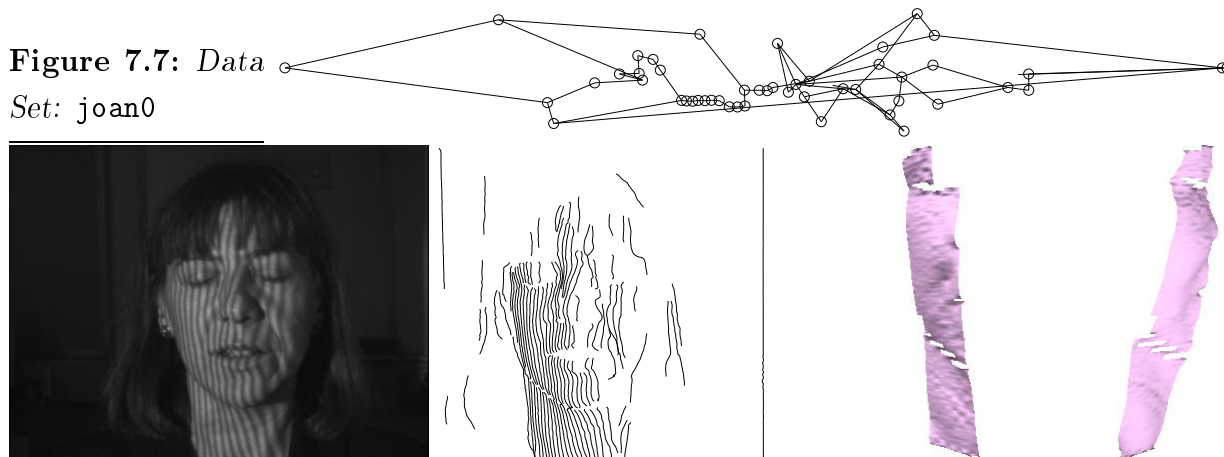
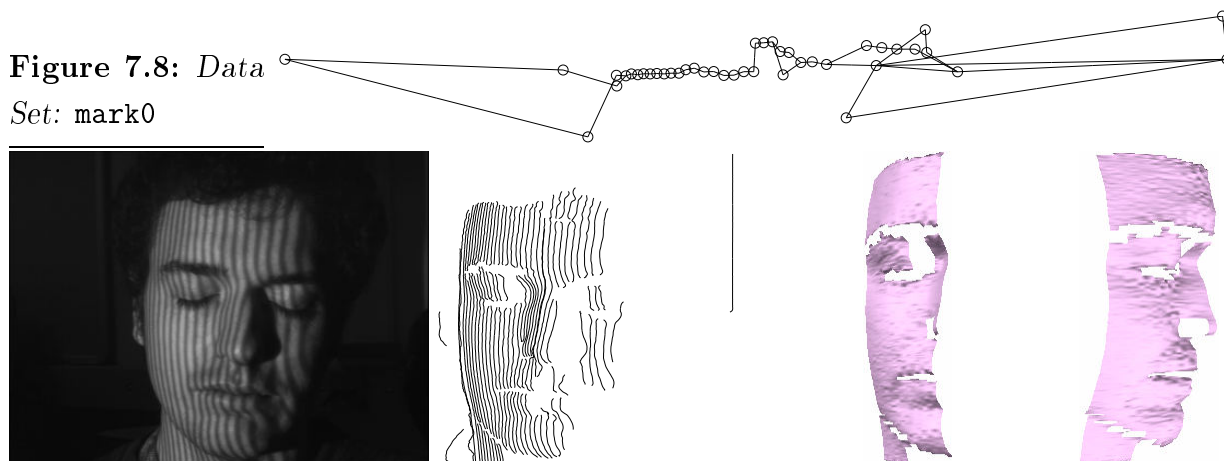
**Figure 7.5:** *Data**Set: davec4***Figure 7.6:** *Data**Set: jln0*

of mistakes causing image edges to cross<sup>1</sup>. This would, unfortunately, cause the image edge data to be more fragmented but as the graph algorithm has proved its ability to assemble image edges into projected edges (whereas it cannot deal with image edges that cross, at all), this would be of overall benefit.

### *Comparisons*

It is of interest to make a comparison of the results of this system with those of Van Gool [37] and Urquhart [44] as these are the most relevant to the system described in this thesis. In particular Van Gool's group appear to have followed a similar 'train of thought' in selecting a stereo structured light based system. However, they chose to use

<sup>1</sup>Zero crossing lines cannot cross each other - it is the merging described in section 4.4 that can allow the crossing of image edges

**Figure 7.7:** *Data**Set: joan0***Figure 7.8:** *Data**Set: mark0*

a grid pattern and attempt the problem of line detection rather than edge detection. A line detector (equivalent to an impulse detector in one dimension) is necessarily much more sensitive to changes in scale than an edge detector - if the line gets too thin then it 'disappears' and if it gets too wide or the edge intensity gradient drops too much then the algorithm will have difficulty finding and then placing the line. An edge detector merely has to cope with as wide a range of edge gradients as possible - accomplished in this case using a range of filters that encompass a scale change of just over a decade. Thus the use of a line detector implies considerable constraints on the spacing of the grid and the positioning of the camera and projector with respect to the subject individual. Also the use of a grid means that line detection is not possible over the cross points of the grid, so the output from the line detector is a set of line fragments that correspond to sections from the edges of each grid square. The corner positions are then inferred from these edge

Figure 7.9: Data

Set: nick4

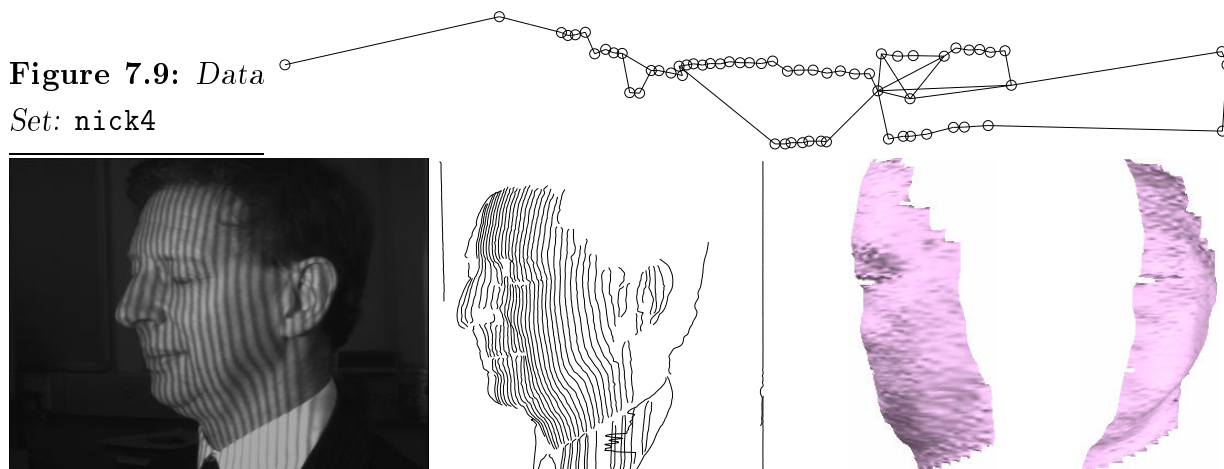
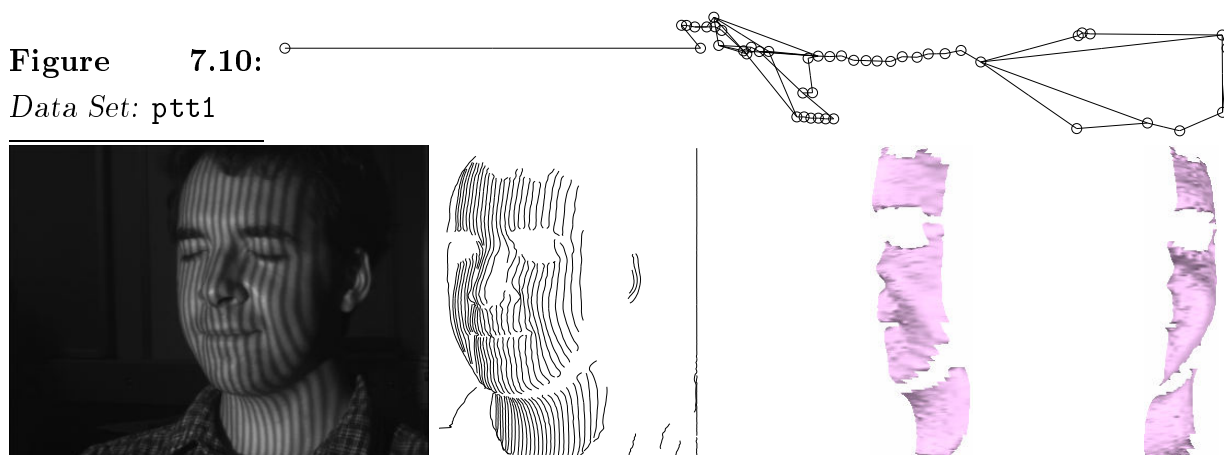


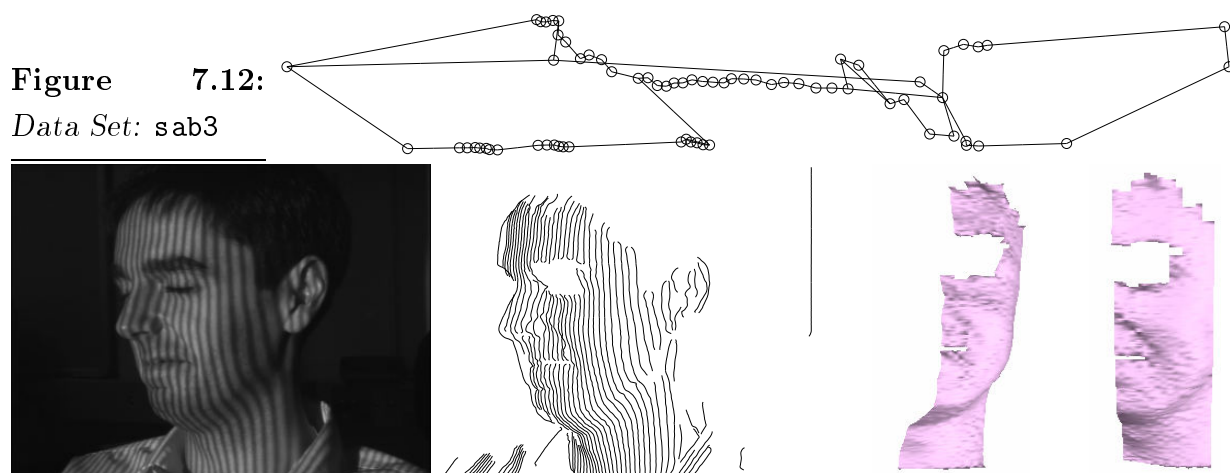
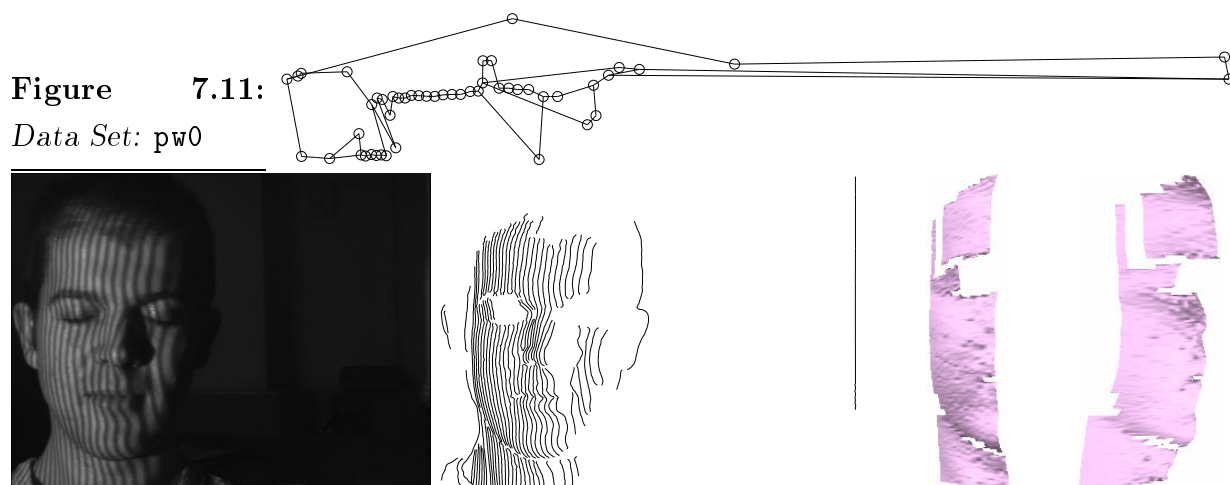
Figure 7.10:

Data Set: ptt1



fragments. In summary, the implication is that this system would only be suitable for a tightly confined volume situation so that the face was at the correct range to develop the correct size grid.

Urquhart's system basically takes the alternative approach of using stereo cameras, accepting that the projection of a pattern may be necessary simply to introduce enough features into the images to allow stereo correlation. In this case the nature of the correlation algorithm implies that the allowable disparity between the images must be limited both to control the execution time (which increases approximately with the square of the disparity range) and to prevent false correlations. This causes a trade-off between accuracy and ease of processing, although depth accuracies of 1 in 500 with respect to the viewed volume are reported. The disparity map is very dense (equivalent approximately to one disparity value per pixel in the camera image) although it is accepted that the



accuracy is compromised further by the use of a random texture to allow correlation over smooth areas of the face.

Both these systems address slightly different requirements from the system described in this thesis which would appear to be significantly more adaptable to scale changes than Van Gool's system and considerably faster in execution than Urquhart's system.

### 7.1.2 Program Execution Time

Program execution time is dependent on the size and content of the supplied camera images. However, the typical time to perform all the processing on the image and display one view of the final surface varies (on the SGI O2) from under 5 seconds for the single reduced size image (207x272 pixels) used to create many of the figures in chapter 5 to

Figure 7.13:

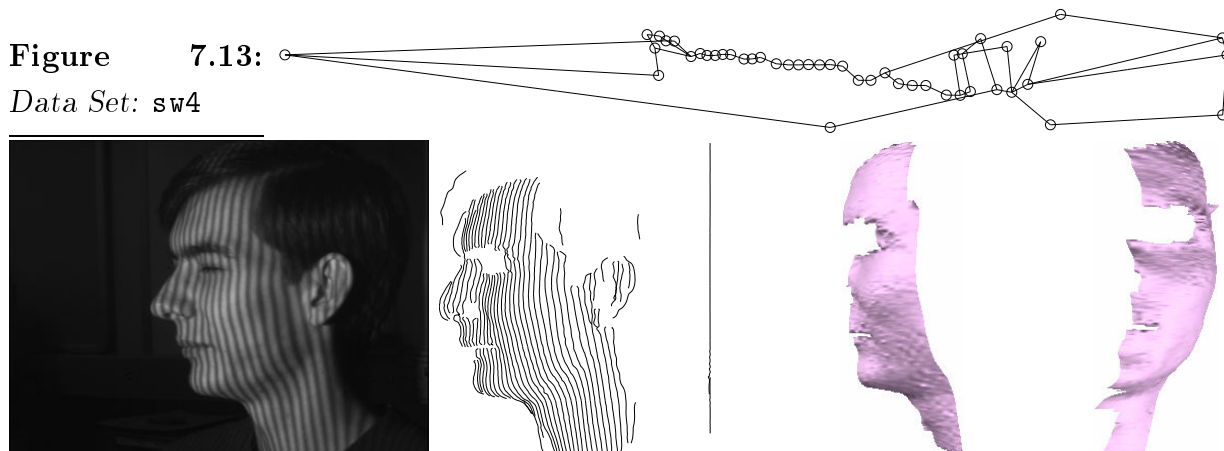
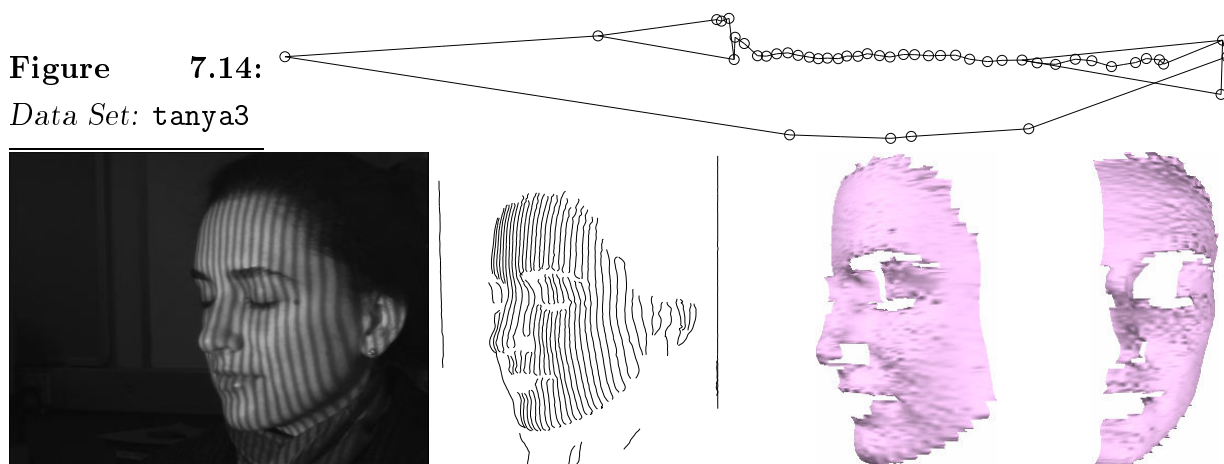
*Data Set: sw4*

Figure 7.14:

*Data Set: tanya3*

about 40 seconds for a two camera view using larger size images (512x512 pixels). These times include the output of files containing the zero crossing lines for each filtered image, the set of image edges and the graph structure at each stage of collapsing. The program and computer equipment details are discussed further in appendix B.

## 7.2 CLASSIFICATION

Classification is the process by which an object may be classified into one of a set of predefined classes. In the case of classification applied to recognition or identification, each individual is regarded as a unique class. Hence, identification would be confirmation that the subject individual was of the class type that was claimed and recognition would

Figure 7.15:

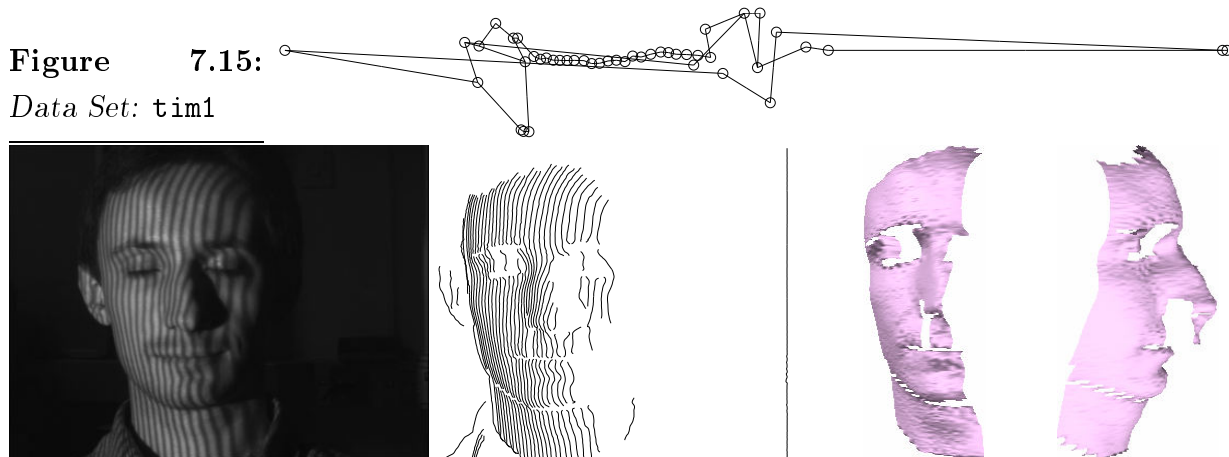
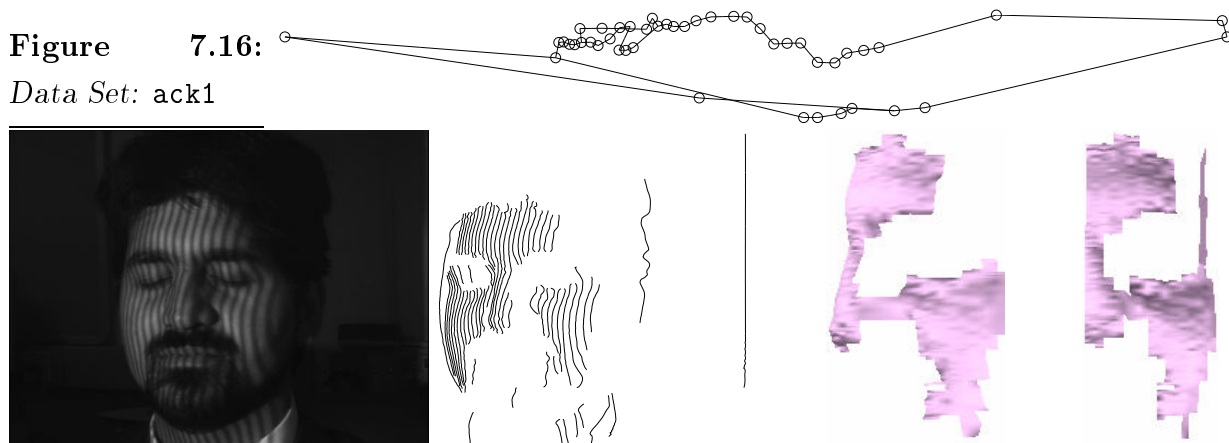
*Data Set: tim1*

Figure 7.16:

*Data Set: ack1*

be the determination of which class, if any, the subject individual belonged to.

For the purposes of such a project, there are many suitable classification algorithms in the literature and many books on the subject such as the classic work by Therrien[42]. The intent is that this surface extraction process would be used to provide a ‘front end’ to a relatively standard classifier and demonstrate the benefits of the three dimensional approach.

To this end, work would have to be done to determine the features that will be used to supply information to the chosen classifier. The range of possibilities include:

- Use of the mesh data for comparison (e.g. generate volume errors between subject mesh and library meshes)
- Extract distances and shapes from the surface (e.g. eye size, cheek shape, forehead shape, nose length, etc.)

- Use surface curvature information (as suggested by Gordon [19])

From there, standard classifier training and testing methods could be used to assess the performance of the system.

## 7.3 FURTHER WORK

Areas of the 3D acquisition system which could form the basis for further work include:

- The image edge extraction process could include more information relating to the projected feature type, size and spacing. This may create a more robust feature extraction process that would allow the use of a smaller feature spacing, thus creating a denser surface mesh. In particular, prevention of the crossing of image edges would reduce many of the problems in the graph collapse phase.
- Improve the stereo geometry calculations to include pinhole camera and projector models. As shown in section 6.2.1, the approximation of parallel plane camera and projector models may not be sufficiently accurate for some applications. It would be a relatively straightforward process to include a pinhole camera model instead. This would, however, require significantly more parameters including at least the following: the exact location of the cameras and projector with respect to the subject head, the focal length of the camera lenses (including possible changes with focus<sup>2</sup>), the CCD cell size and the CCD position with respect to the optical centre of the lens. Possible further parameters would include lens distortion and CCD array skew.
- Investigate the possibility of using infra-red illumination to minimise the subjects awareness of the process. Currently with a system using a slide projector, there is some discomfort experienced by the subject due to the high intensity light. Use of an infra red source would remove this problem. However, it would require a specialised projector, possibly involving a laser source and would thus require considerable effort to ensure its safety. Most monochrome CCD cameras are sensitive in the near infra-red so would only require an optical filter to remove the visible spectrum.

---

<sup>2</sup>Most camera lenses only have their specified focal length when focused at infinity - at any other focus the focal length is lower than the specified value

- 
- Assess the maximum feature density possible in order to increase the mesh size and thus increase the information in the surface model. Currently due to the time required to construct the slides for use in the projector, it was not possible to investigate fully the range of projected edge densities over which the system would function. If a specialised projector was constructed, particularly if it involved a coherent light source, then this adjustment could easily be included in its design.
  - Consider the use of two camera images, one with the projected pattern and one without, in order to use conventional two dimensional techniques augmented with the three dimensional information. This would require a projector that could be switched off between one camera frame and the next. Alternatively, the projector could use a light source that could be switched on for all or part of one single frame. In conjunction with a fast shuttered camera (the cameras used in this project were capable of 1/10 000 s shutter speeds) synchronised to the projector, it should be possible to acquire alternate images of the subject with stripes and under natural lighting. This would however, introduce the problem of motion between frames as, without the use of specialised high frame rate cameras, 1/25 s interval between frames is sufficient to allow a considerable frame difference for a subject who is not consciously keeping still. If this problem were successfully overcome, then there would be potential for the interesting combination of two dimensional and three dimensional information. For example, if a colour camera was used to acquire an image of the naturally lit subject, then the colour information could be directly imposed onto the surface generated from the image of the stripe illuminated subject - allowing a full colour reconstruction. This would be possible without the problem of motion between images, if infra-red stripe illumination was used on an otherwise naturally lit subject and images were acquired using a 4 CCD camera - a camera in which the light is equally split onto 4 CCD arrays, each with its own optical filter (red,green,blue and infra-red). Three CCD cameras are commonly available at the mid to upper range of the video camera market, but such a 4 CCD system would likely need to be specially made.
  - Use of multiple frames between which there has been motion could also improve the surface density, provided sufficiently accurate matches could be made between the generated surfaces in order to combine them. This would be akin to the 'super resolution' techniques used in image processing in which multiple images of a subject

are combined to form a single ‘super resolution’ image - the problem clearly being the successful registration of the images to sub-pixel accuracy.

# Glossary

---

# A

**Image Feature** : Any artifact in an image which is clearly and repeatably identifiable

**Projected Feature** : An artifact that is deliberately projected into the view of the camera to create Image Features

**Original Image** : The greyscale image acquired by the camera

**Filtered Image** : One of the set of images created by using different filters on the Original Image

**Zero Crossing Line** : An edge detected in one of the Filtered Images

**Image Edge** : An edge in the Original Image which is detected by the presence of Zero Crossing Lines in the Filtered Images

**Projected Edge** : The actual edge caused by the projected pattern intersecting the surface

**Edge/Feature Polarity** : In the greyscale image the polarity is the sign of the gradient of the greyscale intensity of the feature/edge, from left to right.

## B.1 INTRODUCTION

The use of an Object Oriented Programming language has been key to the successful development of the techniques described in this thesis and it is beneficial to show the manner in which these techniques have been implemented. In this type of language, the program structure is centered around the creation and manipulation of data objects. The language used in this case was C++<sup>1</sup>. As any more than a very simple introduction to C++ is beyond the scope of this dissertation, reference to a suitable textbook is recommended such as ‘The C++ Programming Language’ [41] or ‘Advanced C++ - Programming Styles and Idioms’ [13].

Unfortunately, it is necessary to assume a basic familiarity with C++ for the rest of this appendix. The key concepts are that:

- Data, or pointers to data, are stored in the member variables of each object
- Member functions are used to access and manipulate the member variables of an object.
- Constructor functions create the data objects and can be trivial or quite complex, involving, for example, the allocation of other areas of memory to store the object data in. They have the same name as the class.
- The destructor function is responsible for the cleaning up of the object and may be trivial or quite complex, involving, for example the deallocation of memory allocated

---

<sup>1</sup>For a somewhat bizarre interpretation of the choice of name for this language, refer to the appendix of ‘1984’ by George Orwell [35].

by a constructor. They have the same name as the class, prefixed with a `.`

- Operators can be ‘overloaded’ so that, for example, the concept `a>b` would be valid even if `a` and `b` were both image edges.
- Data and functions can be ‘hidden’ inside an object by declaring them `private:` (as opposed to `public:`) and can then only be manipulated by other member functions. This ensures that data can only be accessed by the (carefully written) member functions that are relevant.
- Classes can be ‘derived’ from other classes, by which they acquire the functionality of the ‘base’ class and add new functionality on top of it. An example of this is the `DLINK` (described later) - any class derived from a `DLINK` can be inserted into a `DLLIST` (a double linked list - described later). The `DLINK` and `DLLIST` routines can thus be written without the knowledge of whatever object may be inserted into the list and the derived class routines can be written without needing to know the details of how an object is stored in a list.

If a C++ program is written correctly then the resultant program flow should be extremely clear and very similar to the pseudo-language of section B.2.10 in which the detail of each process is hidden away inside the class member functions. For example, the detailed manipulation of linked list objects is performed by classes `DLINK`, `DLLIST` and associated classes (in software module `dlinkdef`) yet the routines dealing with the listing of `LINE` objects need only know that functions `insert()` and `remove()` (which are defined in `dlinkdef`) will put an object in a list and remove it, respectively.

It is fair to say that without the use of C++ the development of the graph handling routines described in chapter 5 would have been extremely difficult and their debugging (discussed later), virtually impossible.

## B.2 THE C++ PROGRAM STRUCTURE

This section describes the individual modules that make up the overall program. Each module or file created the functionality for a C++ class or related set of classes. Broadly the modules can be grouped together by chapter. The image filtering and image edge extraction of chapter 4 is provided by modules `filtdef`, `imagedef`, `linedefs`, `leveldef` and

`dlinkdef`. The graph creation and manipulation of chapter 5 is provided by `graphdef`, `glinedef` and `minkdef`. The surface creation and display of chapter 6 is provided by `surfdefs`.

For brevity in the code fragments, references to trivial Constructor and Destructor functions have been omitted. Likewise minor member functions that are subsidiary to other member functions whose operation is clear have not been mentioned. Member variables have brief comments next to them, as do some of the simpler functions. More detailed comments about the functions and some of the variables occur in the text following each code fragment. Where a variable is referred to as a ‘pointer’ this implies that the actual data storage has to be allocated by the constructor (and removed by the destructor). This is usually the case for data where the size of the array is not known until run time or where a large amount of data storage is required. Input and output functions have not normally been described although most classes were provided with functions to load and save their data to a file using the `ifstream` and `ofstream` classes respectively.

During the text following the code fragments, references have been made to the relevant sections of the dissertation that are implemented by the various functions in order to prevent duplication of the explanation of their purpose. However, to keep down the frequency of these references, there may only be a general reference to the chapter or relevant section of the dissertation, where it is clear which sub-sections would apply to specific functions.

### B.2.1 `filtdef` - Filter definitions

The file `filtdef.hpp` contains the definitions and functions prototypes used for the creation of the Gaussian filters used in the difference of Gaussian filtering process.

```
class FILTER {
private:
    int length;        // Must be odd
    int half;         // Must be (length-1)/2;
    int interval;     // tap spacing
    float *fval;      // Coefficients. Must sum to 1
public:
    FILTER(float FilterRatio,int FilterLevel);
    friend IMAGE& IMAGE::operator*=(const FILTER& filt);
};
```

The function `FILTER(float FilterRatio,int FilterLevel)` creates a Gaussian filter suitable for use at the given `FilterLevel` as described in section 4.2.1. The coefficients of the filter are stored in the array `fval`.

The overloaded operator `IMAGE::operator*=(const FILTER& filt)` performs convolution of an `IMAGE` (described later) with the given `FILTER`, `filt`.

### B.2.2 imagedef - Image definitions

The file `imagedef.hpp` contains the definitions and function prototypes used for the creation and manipulation of images, both filtered and un-filtered.

```
class IMAGE {
private:
    int xs,ys;           // Image size in pixels
    float maxpval,minpval; // Peak pixel values
    float **pval;       // Pointer to pixel values
public:
    IMAGE(const char *inputfile); // Load IMAGE from file
    void scale(double newmaxpval,double newminpval); // Scale image
    IMAGE& operator*=(const FILTER& filt); // Image/filter convolution
    IMAGE& operator-=(const IMAGE& im); // Image Subtraction
};
```

`IMAGE(const char *inputfile)` loads a `.pgm` (Portable Grey Map) file with filename `inputfile` from disk.

Scaling of the pixel values of the image to set the peak grey values to new limits is performed by `scale(double newmaxpval,double newminpval)`

Convolution of an `IMAGE` with a `FILTER` is performed with `IMAGE& operator*=(const FILTER& filt)`. This function uses image boundary reflection where necessary.

`IMAGEs` are subtracted from each other (to form difference of Gaussian images) using `IMAGE& operator-=(const IMAGE& im)`

### B.2.3 linedefs - Line definitions

The file `linedefs.hpp` contains the definitions and function prototypes used to manipulate zero crossing lines:

```
class LINE : public DLINK {
private:
```

```

    int arraylength;    // length of line vertically
    int topend;        // start position (vertically)
    int botend;        // end position (vertically)
    int sgn;           // polarity of line
    float xposaverage; // average x value
    float *xpos;       // pointer to x values
    float *slope;      // pointer to gray scale gradients
public:
    int comp(LINE *lin,int n); // Compare lines (this and lin)
    int comp(LINE *to);        // Compare lines for left/right-of ness.
    int istotherightof(void* a); // Compare adjacency for use in glinedef
    int istotheleftof(void* a); // Compare adjacency for use in glinedef
};

class LINETAG : public DLINK {
public:
    LINE *lin;
    DLLIST<LINE> *s;
};

```

Class LINE objects are stored in a Double Linked List when grouped into LEVELs, hence the reference to class DLINK described later.

The comparison routine `comp(LINE *lin,int n)` is used to decide whether a line in one level is similar to a line in another as described in section 4.4.1

The comparison routine `comp(LINE *to)` is used by functions `istotherightof(void* a)` and `istotheleftof(void* a)` to determine LINE adjacency. These functions are used by GLINE functions, invoked by the Graph adjacency tests described in section 5.2.2

Class LINETAG objects are used to mark the lines in the pseudo-function `find_line` described in section 4.4.1. The list of LINETAG objects record the LINES that will be merged or removed and the `line_set` that they are in (see next sub-section).

#### B.2.4 leveldef - Level (line set) definitions

The file `leveldef.hpp` contains the definitions and function prototypes used to create the zero crossing line sets.

```

class LEVEL : public DLINK {
private:
    DLLIST<LINE> line_set;    // The list of LINES in this line set

```

```
    IMAGE *im;                // Pointer to the filtered IMAGE
    int n;                    // Line set number
    int xs;                   // x-size of image
    int ys;                   // y-size of image
    void add_to_list(float xpos,int y,float slope);
    void purge_lines(int ycurr);
    void find_match_lines(LINE *lin);
    void merge_tag_set(LINE *lin);
public:
    int zcreate();
    void match(DLLIST<LEVEL> &levels);    // level merging
};
```

As with class `LINE`, class `LEVEL` is stored in a Double Linked List as there are several line sets. Class `LEVEL` is used both for the sets of zero crossing lines extracted from each filtered image and to store the final merged set of lines.

The function `zcreate()` is the most important to this class and class `LINE`. This function is called to find and record the zero crossing lines for a particular filtered image as described in section 4.3. It scans the image horizontally from the top down searching for zero crossing points. Having found one it attempts to add this point to one of the `LINES` in the `line_set` using function `add_to_list(float xpos,int y,float slope)`. Thus the `LINES` in `line_set` 'grow' down from the top of the image. If there is no suitable `LINE` to add the point to, a new `LINE` is created.

Function `purge_lines(int ycurr)` is used to clear out lines that are too short to be significant. The variable `ycurr` is the current scan line being searched by `zcreate()` so that lines which are short but may still be growing (i.e. their end point is `ycurr`) are not removed.

When this class is used to store the final merged set of lines, the merging process is initiated by function `match(DLLIST<LEVEL> &levels)`. This function collects sets of lines, as described in section 4.4, using the following functions.

Function `find_match_lines(LINE *lin)` is used in the line set merging searching process of section 4.4.1. It searches in its `line_set` for a `LINE` similar to `lin` which is from another line set. It uses the function `comp(LINE *lin,int n)` from class `LINE`.

Having found and marked a suitable set of lines using a list of `LINETAG` objects, they are either discarded or merged together using function `merge_tag_set(LINE *lin)` which forms an image edge from the set of `LINES` presented to it.

### B.2.5 dlinkdef - Double Linked List definitions

The file `dlinkdef.hpp` contains the class definitions and function prototypes necessary to create a double linked list system. These lists are used to contain `LEVEL` objects when attempting to merge line sets and are also used to contain `LINE` objects to make up a line set within each `LEVEL`.

```
class DLINK {
private:
    DLINK* prev;           // Previous link in list
    DLINK* next;          // Next link in list
};

class DLLIST_BASE {
protected:
    DLINK* first;         // First link in list
    DLINK* last;         // Last link in list
    DLINK* current;      // Currently selected link

    void preinsert(DLINK*); // Insert this link after current
    void postinsert(DLINK*); // Insert this link before current
    void unhitch(DLINK*); // Unhitch this link
    int CountItems(); // Count items in list
};

template<class T>
class DLLIST : public DLLIST_BASE {
public:
    void preinsert(T* a); // Insert this item after current
    void postinsert(T* a); // Insert this item before current
    void remove(T* a); // Remove and delete this item
    void ClearAll(); // Delete all items in list
    int CountItems(); // Count items in list
};

class DLLIST_BASE_ITER {
private:
    DLINK* ce;           // Current link in list
    DLLIST_BASE* cl;    // Pointer to list
public:
    DLINK* operator() (); // Return current link
};
```

```

    DLINK* preincrement(); // Move to next link
    DLINK* predecrement(); // Move to prev link
    DLINK* postincrement(); // Move to next link
    DLINK* postdecrement(); // Move to prev link
    DLINK* first(); // Move to first link
    DLINK* last(); // Move to last link
};

template<class T>
class DLLIST_ITER : private DLLIST_BASE_ITER {
public:
    T* operator() (); // Return current link
    T* operator++(int); // Move to next link
    T* operator++(); // Move to next link
    T* operator--(int); // Move to prev link
    T* operator--(); // Move to prev link
    T* first(); // Move to first link
    T* last(); // Move to last link
};

```

This double linked list is an intrusive list. This means that to be put in a list an object must be derived from (attached to) a link object in order to ensure it has the necessary pointers. In this case the link object is class `DLINK` and contains pointers to a next link and previous link - hence double linked list.

A list 'base' class `DLLIST_BASE` is necessary in which to store at least one pointer to an item in the list. In this case two are used, `first` and `last`, for convenience. Items are inserted into the list using `preinsert(DLINK*)` and `postinsert(DLINK*)` relative to the link stored in `current`. i.e. `postinsert` puts the new item between `current` and `current->next` whereas `preinsert` puts the new item between `current->prev` and `current`, in both cases obviously separating the link that was previously there. Items are removed from the list using `unhitch(DLINK* a)` which joins `a->prev` to `a->next` and vice versa, thus cutting out item `a`. The total number of items in the list is returned by `int CountItems()`.

It is usual to write list handling functions that just refer to `DLINK` objects as then they are completely general to the list class, but in use the list will obviously contain more complex objects and it should appear as though the larger object is being manipulated, not just a `DLINK`. The neatest way to achieve this is through the use of templates. This is a 'wrapper' class that contains the necessary conversions from complex object to `DLINK`

and back again without them being visible to the body of the program. In this case the class is `template class<T> class DLLIST`. This would allow the main program to create a list for LINES by using `DLLIST<LINE> linelist`. The functions in class `DLLIST` just call their equivalent named function in class `DLLIST_BASE`, but will appear to be handling objects of type `T` (in this case `LINES`).

To traverse and manipulate a list, an ‘iterator’ class `DLLIST_ITER` is introduced. It also requires the use of a template to ‘hide’ the presence of the `DLINK` objects. So an iterator for scanning line set `linelist` would be created using `DLLIST_ITER<LINE> iter(linelist)` which creates an iterator called `iter` that can traverse a list containing type `LINE`, in particular the list called `linelist`. Having done this, the current item the iterator points to can be moved forward using `iter++` or `++iter` and moved back using `iter--` or `--iter`. The current item is returned using `iter()` and the iterator is reset to the first or last item in the list using `iter.first()` or `iter.last()` respectively. The iterator functions all set the value of `current` in the `DLLIST_BASE` object so that future use of the `preinsert` or `postinsert` functions will be with respect to this item in the list.

### B.2.6 `glinedef` - A line grouping definition

The file `glinedef.hpp` contains the definitions and function prototypes for a class in which several image edges (`LINES`) are grouped together. This and the multiple linked list object are the principal storage objects for use with the Adjacency Graph described in chapter 5.

```
class GLINE : public MLINK {
private:
    int nlines;                // Number of lines in this grouping
    LINE* lines[MAX_MLINKS]; // Pointers to stored lines
    float xposaverage;        // Average x value of lines
    float yposaverage;        // Average y value of lines
public:
    GLINE(GLINE* a, GLINE* b);
    int istotheleftof(void* a);
    int istotherightof(void* a);
    int iscollapsable(void*, void*);
};
```

As the `GLINE` object is to be stored in a graph or Multiply Linked List, it must be derived from (attached to) a Multiple Link (`MLINK` - described later). A set of `nlines`

LINES is stored in `lines`. The average x and y values are stored for use with the graph display routines described in section 5.2.6.

As nodes in the graph are collapsed as described in section 5.2.9 a new node is created from the two that are being joined using the constructor `GLINE(GLINE* a, GLINE* b)` which creates a new `GLINE` that contains all the `LINES` formerly contained in `a` and `b`.

The testing functions `istotherightof(void* a)` and `istothelleftof(void* a)` are as described in section 5.2.2. They obviously make considerable use of the `LINE` comparison operator described with class `LINE`.

The function `iscollapsable(void*, void*)` is used to determine if two nodes may be collapsed or not as described in section 5.2.9

### B.2.7mlinkdef - The Multiply Linked List

In the same way that a doubly linked list contains items that refer to a previous item and next item, a multiply linked list contains items that can refer to several previous items and several next items. This is the basic node object used in the creation of the adjacency graph as described in section 5.2.1. As with the `DLLIST` the `MLINK` object contains the necessary connectivity information with the attached data object carrying other information not related to connectivity, in this case this is a `GLINE`. As with the `DLLIST` the manipulation of the graph is done at the `MLINK` level with a template to interface to the rest of the program.

This set of classes provides the key functionality for the whole of the Adjacency Graph processing.

The file `mlinkdef.hpp` contains the definitions and function prototypes necessary to form the multiply linked list class.

```
class MLINK {
private:
    MLINK* mnext[MAX_MLINKS]; // Pointer to next items
    int num_next;             // Number of next items
    MLINK* mprev[MAX_MLINKS]; // Pointer to previous items
    int num_prev;            // Number of previous items
    static int next_label;   // The next available label
    int tag;                 // \
    int rtag;                // | See text
    int tag2;                // |
    int rtag2;               // /
```

```
    int search_left_for(MLINK*);
    int search_right_for(MLINK*);
    void iscollapsable_reset();
    void iscollapsable_set();
public:
    int label;           // A Unique label
    void mark();
    void unmark();
    int istagged();
    virtual int istotheleftof(void*);
    virtual int istotherightof(void*);
    virtual int iscollapsable(void*,void*);
    int isunhitchable();
};

class MLLIST_BASE {
private:
    void set(MLINK *);
    void rst(MLINK *);
protected:
    MLINK* first;       // First item in list
    MLINK* last;       // Last item in list

    void insert(MLINK* a,MLINK* b);
    int unhitch(MLINK*);
    void tidylink(MLINK*);
    void reset_tags();
    void search(MLINK*,MLINK*);
    void recursive_test(MLINK*,MLINK*,MLINK*);
    void recursive_toright(MLINK*,MLINK*,MLINK*);
    void recursive_toleft(MLINK*,MLINK*,MLINK*);
    int islinear();
    int count();
    int linearlength( MLINK*);
    MLINK* linearlengthcheck();
public:
    int integrity_left(MLINK* =NULL);
    int integrity_right(MLINK* =NULL);
};

template<class T>
```

```
class MLLIST : public MLLIST_BASE {
public:
    void insert(T* a,MLINK* b);
    int remove(T* a);
    int unhitch(T* a)l
    void printl(ostream& s,T* a);
    void printr(ostream& s,T* a);
    void search(T* test_line,MLINK* test_results);
    int islinear();
    int linearlength(T* a);
    T* linearlengthcheck();
    int count();
};

class MLLIST_BASE_ITER {
protected:
    MLLIST_BASE *cl;    // Pointer to list
    MLINK* ce;         // Current element in list

    void first();
    void last();
    MLINK* rand_to_right();
    MLINK* rand_to_left();
    void csr(MLINK*);
    void csl(MLINK*);
    int collapse_search_right();
    int collapse_search_left();
    void collapse_insert(MLINK*,MLINK*,MLINK*);
    MLINK* operator()();
};

template<class T>
class MLLIST_ITER : public MLLIST_BASE_ITER {
public:
    T* rand_to_right();
    T* rand_to_left();
    int collapse_search_right();
    int collapse_search_left();
    T* operator()();
    void first();
    void last();
};
```

```
};
```

As with the doubly linked list, this is an intrusive list class, requiring objects that will be listed to be derived from class `MLINK`. `MLINK` thus contains pointers to several next pointers and several previous pointers (`mnext` and `mprev` respectively) and a count of how many are in use (`num_next` and `num_prev`). Each `MLINK` has a unique `label` and the next free label is stored in `next_label`. As this is a `static` variable it means the same number is visible to all the `MLINK` objects so every time a new `MLINK` is created it can increment `next_label`.

Most of the functions that manipulate the multiply linked list have to be recursive as the only way to reach all the items in a list is to start at the first one and then search all the `mnext` links and at each of them, search each of their `mnext` links, and so on (obviously starting at the last one and searching all `mprev` links works too). In order to prevent recursive functions searching `MLINKs` multiple times (as there may be several paths to some `MLINKs`) it is necessary to mark them as having been traversed already so that if they are met again, that branch of the recursion can be stopped. This is the purpose of the `tag` variable and `mark()`, `unmark()` and `istagged()` functions. The first time a `MLINK` is traversed it is marked so that if the recursive function finds that link again, it is stopped by referring to the `istagged()` function. However, having used these `tags` in a recursive function, they must be reset (`unmarked`) before they can be used by another recursive function. This requires the use of another marker, `rtag`, to ensure that all of the `MLINKs` are visited and have their `tags` reset. An extra complication is that the functions that check for mutual reachability, as described in section 5.2.9, require the use of recursive search functions `search_left_for(MLINK*)` and `search_right_for(MLINK*)` many times *during* another recursive procedure. Hence the need for variables `tag2` and `rtag2` and functions `iscollapsible_reset()` and `iscollapsible_set()` which use `rtag2` to reset `tag2`.

This set of classes is significantly more complex than set for the doubly linked list. An added complication is the use of ‘virtual’ functions. Although all the graph manipulation is done using the `MLINK` objects, there are occasions when a graph manipulation process such as insertion relies on knowledge of the actual listed object (in this case a `GLINE`). Obviously the process of inserting is done by manipulation of `MLINKs` but the position in which to insert the new object is determined with reference to the `GLINE` data. This can be achieved using ‘virtual functions’ which appear to be referring to `MLINK` objects, hence can be used in the general multiply linked list functions, but actually call a function

belonging to class `GLINE`.

This is the case with the test functions `istotherightof()`, `istotheleftof()` and `iscollapsable()`. These tests all call the appropriate `GLINE` function, described previously.

The `isunhitchable()` function determines from the connectivity if a node is removable from the graph, as described in section 5.2.5.

Class `MLLIST_BASE` is the holder for `first` and `last` objects in the graph.

Function `reset_tags()` uses functions `set(MLINK*)` and `reset(MLINK*)` to clear the tags of all the items in the list. It is used after any of the recursive functions have been used.

Function `insert(MLINK*a,MLINK*b)` inserts `MLINK a` at a location described by `MLINK b`. i.e one of the search functions will create `b` to say where `a` should be put in the graph. This function then performs that insertion and updates all the surrounding links. There are two types of insertion as described in section 5.2.4 and this function will perform them both.

Node removal described in section 5.2.5, is performed with function `unhitch(MLINK*)`. As this is not always possible, it returns a value to say whether it occurred or not. Function `tidylink(MLINK*)` is called immediately afterwards to ensure that the nodes formerly pointed to by the removed node do not have any duplication of `mprev` or `mnext` links.

Function `search(MLINK* test_link,MLINK* test_results)` performs the search to determine the correct location for `test_link`. This location is returned in `test_results` and is passed to function `insert()`, described above. This function performs both types of searching as described in section 5.2.4. Type 1 searching is performed by calling function `recursive_test()` with the test link, the results storing link, and the `first` link as a start point. If it does not find any suitable locations then type 2 searching is performed using function `recursive_toright()` with the test link, results link and `first` link, followed by `recursive_toleft` with the test link, results link and `last` link. Being recursive, each of these functions will call itself with for each of the `mnext` links to continue the search (`mprev` links if searching to the left) unless the links are marked as having been already visited.

The `islinear()`, `linearlength()` and `linearlengthcheck()` functions are used to see if the graph has collapsed to a single line and to extract the longest linear section if it has not, as described in section 5.2.9.

The template wrapper class `MLLIST` is used in the same way as class `DLLIST` was to

‘hide’ the use of `MLINKs` from the outer program, which only deals with `GLINEs`. A multiply linked list of `GLINEs` is therefore created with `MLLIST<GLINE> list`.

The concept of iteration, introduced with the `DLLIST` also applies to multiply linked lists and is templated in the same way, but is more complex as there is often a choice of which link to proceed to. Thus `MLLIST<GLINE> iter(list)` would create a list iterator for the `MLLIST` created above. Function calls `iter.first()` and `iter.last()` would set the pointer to the first and last items in the list respectively and function call `iter()` would return the currently pointed to `GLINE`

The process of iterating through the graph as used during node shuffling (section 5.2.8) and collapse searching (section 5.2.9) is done randomly. There is no deterministic way of ensuring that each possible pathway is searched exactly once, as the processes of shuffling and collapsing, by their very nature, change the shape of the graph. So each time the graph is iterated through on the next shuffling or collapsing pass, it has to be treated as a new graph. The only way, therefore, to select a particular branch where there is a choice, is to pick a random one. This is done using functions `rand_to_left()` and `rand_to_right()` which return the next item to the right or left, making a random decision if there is a choice.

When collapsing the graph (section 5.2.9), the functions `collapse_search_right()` and `collapse_search_left()` are used. These functions respectively call the recursive functions `csr(MLINK*)` and `csr(MLINK*)`. These functions search the graph from the left or right, respectively, using the `iscollapsable()` test function for the `GLINE` objects (via the ‘virtual’ function system, described above).

As with `DLLIST` the class `MLLIST_BASE_ITER` is wrapped in a template class `MLLIST` to provide an interface that does not refer to `MLINK` objects.

### *Multiply Linked List Debugging Functions*

As discussed at the end of chapter 5 the testing and debugging of such a complex set of functions as described above presented a considerable challenge. A number of functions were included in the multiply linked list class set for just this purpose.

By the very nature of such an entity, there are only two points of entry into the list given by the class `MLLIST_BASE` pointers, `first` and `last`, although theoretically only one is necessary. If there are any inconsistencies within the list, they can be very difficult to trace as the relevant information to do with the list structure can only be accessed via the list itself. If that list structure is inconsistent then this task becomes highly problematic.

For example, a damaged list may appear differently depending on whether it is accessed via the `first` or `last` link.

The possible errors can be broadly divided into two types: list consistency errors, in which the graph structure does not follow the connectivity rules described in chapter 5 and pointer errors which actually cause the program to crash whilst running.

Pointer errors occur when a `mprev` or `mnext` value is in some way invalid. This most often occurs when an object is deleted and the links that used to point to that object are not fully removed. This will usually cause the program to crash *next* time an attempt is made to follow that link. i.e. at some other location in the program than the one that made the mistake. These problems have to be tracked using debugging software as this is the only way to catch the bad pointer without the program crashing and then allow access to all the program data to try and find the part that caused the bad pointer. As program execution cannot be reversed, this often requires rerunning the program and stopping it some time before the problem would occur and then ‘single-stepping’ through the code, attempting to spot the point at which the bad pointer was created - a time consuming and tedious business!

List consistency errors are where the program does not crash due to bad pointers but generates an invalid graph structure<sup>2</sup>. Typical examples of this include:

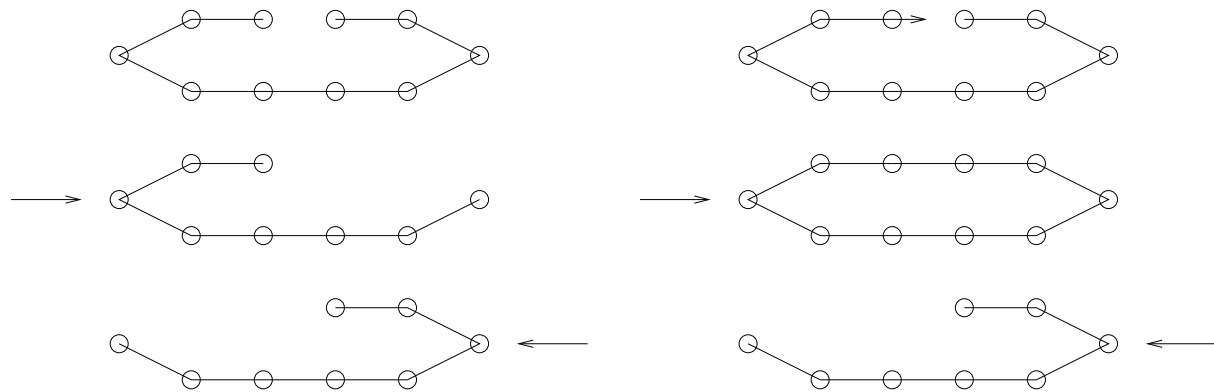
- Duplicate links - where `mprev` or `mnext` contain more than one reference to the same `MLINK` object
- Dangling links - where a link has either no `mnext` entries or no `mprev` entries. Only the `first` and `last` links are allowed this.
- Mismatched links - where `MLINK a` points to `MLINK b` but not vice versa.
- Confused links - where `MLINK a` contains `MLINK b` in both the `mprev` list *and* `mnext` list. This would imply that node `a` was to the left of and to the right of node `b` and can be caused if the mutual reachability test of section 5.2.9 is not performed properly.

All these problems are caused by program implementation errors, but a set of functions to assist in tracking them down were developed.

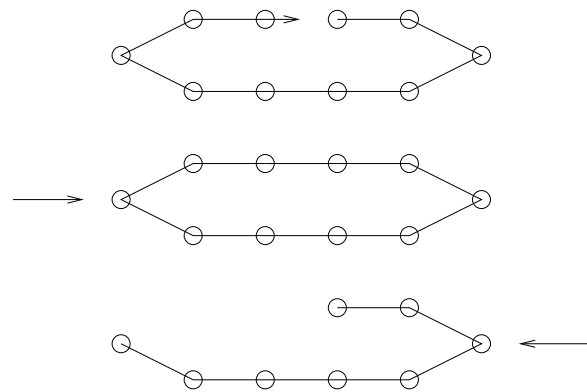
These functions were `printl()` and `printr()` from class `MLLIST` and from class `MLLIST_BASE`, the functions `integrity_left(MLINK*)` and `integrity_right(MLINK*)`.

---

<sup>2</sup>Although this can cause bad pointers, the root cause is the invalid graph structure



**Figure B.1:** *Missing link problem*



**Figure B.2:** *Mismatched link problem*

The `integrity_left(MLINK*)` and `integrity_right(MLINK*)` functions perform a very thorough check of all the links in the multiply linked list, traversing the list in a left (via `mprev`) or right (via `mnext`) direction respectively. An undamaged list will appear identical regardless of whether it is accessed via the `first` or `last` link, but a list that contains dangling or unmatched links will appear very different. Hence it is necessary to perform all such checks from both ends. These situations are illustrated in figures B.1 and B.2.

The following checks are made on each link by the integrity checking functions before recursively calling the same function for all the `mnext` links or `mprev` links as appropriate. In this case the normal recursive ‘tagging’ procedure, described previously, is not performed and the recursion is allowed to traverse links multiple times, if that is what the list branching causes. This is because the checking functions must not, in any way, effect the elements in the list. Use of the ‘tagging’ variable, to limit the recursion, would prevent these checks being performed during any other recursive process. In practice it was the recursive processes that often caused the problem!

- Link must have at least one `mprev` and at least one `mnext` (except the `first` or `last` link)
- The link must not appear in its own list of `mprevs` or `mnexts`
- There must be no duplication in the list of `mprevs` or `mnexts`
- Every link in `mprev` must have a link in `mnext` forward to this link.

- Every link in `mnext` must have a link in `mprev` back to this link.
- No link must appear in the `mnext` and `mprev` lists.

If any of these tests fail then a short report is printed of which link caused the problem and to which other links it is connected. In order to find the exact point in the program that was causing the problem, it was usually necessary to run these checks after every single manipulation of the list, such as an insertion or removal. As these occur very frequently in the processing of the graph (for example, the shuffling process of section 5.2.8 will remove and re-insert virtually all the nodes in the graph and is performed many times) this testing imposed a considerable performance penalty, often slowing execution by at least an order of magnitude. However, they were most useful in that if the integrity check functions were successful in both directions, then it could be concluded that the list was sound and if they were not successful it was usually possible to easily isolate the section of code that was causing the problem.

The printing functions, `printl()` and `printr()`, traversed the list from right to left or left to right respectively, and for every node printed out its label and the labels of all the nodes in the `mprev` and `mnext` lists. It also printed the x and y averages from the `GLINE` data (which is why it is a `MLLIST` function rather than only a `MLINK` function). Obviously these functions could be used in conjunction with the integrity test functions and were needed in left traversing and right traversing forms to highlight missing and mismatched links. However, these functions did make use of the recursive ‘tagging’ system to ensure that only one record was printed out for each node and therefore they could not be used during any other recursive procedure.

These functions were also used to supply the connectivity data to Matlab software in order to generate the plots of the graph structure (used liberally throughout chapter 5) - hence the need for the x and y average data. The Matlab routine used the `printr()` data to plot the ‘next’ links and the `printl()` data to plot the ‘prev’ links. Thus mismatched and missing links would show up very clearly indeed.

### B.2.8 `graphdef` - the graph manipulation object

Although most of the node manipulation functionality is given by the multiply linked list class set, the processing was invoked from the `GRAPH` class.

The file `graphdef.hpp` contains the definitions and function prototypes for the graph handling objects.

```
class GRAPH {
private:
    MLLIST<GLINE> surf;    // The multiply linked list of GLINEs
    int xs,ys;           // Original image dimensions
public:
    GRAPH(DLLIST<LINE>&li, IMAGE *im);
    int collapse();
    void random_shuffle();
    int simplify();
    GLINE* extractbest();
    void big_shuffle();
};
```

Class `GRAPH` only contains the class `MLLIST` object which actually contains the `GLINE` data.

The constructor function fills the `MLLIST` object with the list of image edges provided by the `merge_tag_set()` function from class `LEVEL`. This creation is as described in section 5.2.3 and used the search and insert functions of class `MLLIST_BASE`.

Graph shuffling (section 5.2.8) is performed by function `random_shuffle()` which makes a single left and single right pass through the graph, making a random choice at each branch. This function makes extensive use of the remove, search and insert functions of class `MLLIST_BASE` and the traverse functions of class `MLLIST_ITER`. Function `big_shuffle()` performs this single shuffle several times.

Collapsing of the graph is performed by function `collapse()`. This calls alternately and repeatedly the `collapse_search_left()` and `collapse_search_right()` functions of class `MLLIST_ITER` until no further collapsing is possible.

Function `simplify()` is the master function which cycles through, shuffling and collapsing the graph and relaxing the collapse conditions, as described in section 5.2.9 until the graph is linear or a set number of iterations has been performed. In this circumstance the `extractbest()` function is used to extract the longest linear section from the graph to be passed to the surface generation routines.

### B.2.9 surfdefs - surface generation and display

This class generates, contains and manipulates the three dimensional surface data. File `surfdefs.hpp` contains the necessary definitions and function prototypes.

```
class SURFACE {
```

```
private:
    int nlines;          // number of projected edges in surface
    int xs,ys;          // original size of camera image
    VECTOR **R;         // mesh of surface points
    VECTOR **N;         // mesh of surface normals
    void fillup(GRAPH&,GLINE*,int);
public:
    SURFACE(SURFACE *face1,SURFACE *face2);
    void display();
    void display_one_surface();
    void display(SURFACE*,SURFACE*);
    void find_normals();
    void normalise();
    void trim();
    float overlap(SURFACE*,SURFACE*,int,int,float*);
};
```

The surface data object contains a mesh of three dimensional points and surface normals stored in VECTOR arrays R and N.

The mesh is initially filled by function `fillup()` which carries out the tasks described in section 6.1. It is supplied with the ordered list of GLINE objects representing projected edges which are converted into 3D coordinates and the mesh is then trimmed with reference to the surface curvature. The surface normals necessary for both curvature measurement and surface display are determined with function `find_normals()` and the surface is trimmed using the function `trim()`.

The constructor function `SURFACE(SURFACE *face1,SURFACE *face2)` attempts to generate a combined surface from the two supplied surfaces `face1` and `face2`. It uses the function `overlap()` to perform the minimum squared error search described in section 6.3.

The surfaces are displayed using the OpenGL rendering library created by Silicon Graphics. In this case, as the program was developed on an SGI O2 computer, this library is hardware accelerated so gives astonishing rendering speeds, although the library is available in software form on many other platforms.

The `display_one_surface()` function creates a window for and renders one single surface. It is called either by `display()` in the case of a single face view (one camera only) or by `display(SURFACE*,SURFACE*)` in the case of a combined surface in order to display both single surfaces and the combined surface. Before being displayed the surfaces

are normalised in space to be centered on (0,0,0) and have a maximum distance from the origin of 1, in the x-y plane. This is to fit the surface display into the window.

When displayed in the window, the surface can be tilted and rotated with the mouse and at any time the window image can be output as a PostScript file.

### B.2.10 facescan - the main program

The main program `facescan.cpp` decodes the command line arguments and calls the rest of the routines as appropriate. It is shown, in outline, below:

```
void inifileread(char* inifilename);
SURFACE* process(char *filename);
void main(int argc,char* argv[]) {

    [ ..... ]

    if (TWOCAMERAS) {
        inifileread(inifilename);
        facel=process(filename1);

        inifileread(inifilename);
        facer=process(filename2);
    }
    else {
        inifileread(inifilename);
        facel=process(filename);
        facel->display();
    }

    if (TWOCAMERAS) {
        SURFACE face(facel,facer);
        face.display(facel,facer);
    }

    [ ..... ]

}
```

The function `inifileread()` reads in a file containing the parameters that can be adjusted (such as the `CamAng` and the image edge overlap thresholds).

The function `process()` takes the name of an image file and generates the face surface from it using the classes and routines previous described. In summary the steps are as follows:

- Take the input file and load it into an **IMAGE**
- Create an appropriate set of Gaussian **FILTERS**
- Using these filters create a sequence of Gaussian filtered **IMAGES**
- Subtract consecutive images from each other to create Difference of Gaussian filtered **IMAGES**
- For each D.O.G **IMAGE** find the set of zero crossing **LINES** and store them in a **LEVEL** using a **DLLIST**
- Take the set of **LEVELS** stored in a **DLLIST** and merge appropriate **LINES** from the zero crossing line sets to create a set of image edge **LINES**, also stored in a **DLLIST**
- Feed this **DLLIST** of image edge **LINES** into a **GRAPH** where the **LINES** are stored in **GLINES** and the **GLINES** are stored in the **MLLIST**.
- Simplify the graph by collapsing it, during which process **GLINES** are combined and thus may end up containing several **LINES**
- Take the final linear list of **GLINES** and feed them into a **SURFACE** where a mesh of 3D points will be created. Trim the mesh as appropriate and display it.

In the two camera case, this entire process is performed on both images to produce two surfaces (`face1` and `face2` in the code fragment, above) and then a new surface, `face`, is created by combining these two surfaces together, before displaying all three surfaces (left, right and combined).

### B.2.11 Numerical Output

At any stage, any of the data stored in any of the objects can be output to a file and analysed elsewhere - normally with Matlab. In particular this was used to generate most of the example figures throughout this dissertation and to analyse the reference surfaces in chapter 6.

### B.2.12 Technical details

#### *Operating Systems*

Development of the software first began on a MS-DOS 6 platform. Work moved to MS-Windows 3.1 when access to graphics display facilities and an easier-to-use debugging environment were required. When MS-Windows NT 3.51 became available this was used as it provided a much more resilient programming environment<sup>3</sup>. This also provided access to a software version of the OpenGL system. The final software was developed using IRIX 6.3 (a UNIX variant, proprietary to SGI), a highly resilient programming environment with, in this case, access to a hardware OpenGL system.

#### *Computing Hardware*

The Microsoft operating systems were all used on a Pentium 90MHz system. The IRIX operating systems were developed for SGI computer systems, in this case an SGI O2 R10000.

#### *Compilers*

The Watcom C/C++ compilers were used on the Microsoft operating systems and proprietary Silicon Graphics C/C++ compilers were used on the SGI O2.

#### *Other Products*

Matlab 4 and Matlab 5 (from The Mathworks, Inc) were used extensively to both develop some of the techniques (mainly the filtering) and to display the output from the various software modules - such as the zero crossing lines, the image edges and the graph structure.

#### *Execution Time*

Execution time is dependent on the size and content of the supplied camera images. However, the typical time to perform all the processing of the image and display one view of the final surface varies (on the SGI O2) from under 5 seconds for the single reduced size image (207x272 pixels) used to create many of the figures in chapter 5 to about 40 seconds for a two camera view using larger size images (512x512 pixels). These times

---

<sup>3</sup>read as 'more crash-proof'

include the output of files containing the zero crossing lines for each filtered image, the set of image edges and the graph structure at each stage of collapsing.

# Bibliography

---

- [1] Joseph J. Atick, Paul A. Griffin, and A. Norman Redlich. Face recognition from live video for real-world applications - now. *Advanced Imaging*, 10(5):58–62, 1995.
- [2] Nicholas Ayache and Francis Lustman. Trinocular stereo vision for robotics. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13(1):73–85, 1991.
- [3] G. Bhatia, M.W. Vannier, K.E. Smith, P.K. Commean, J. Riolo, and V.L. Young. Quantification of facial surface change using a structured light scanner. *Plastic and Reconstructive Surgery*, 94(6):768–774, 1994.
- [4] OpenGL Architecture Review Board, editor. *OpenGL Reference Manual*. Addison-Wesley Publishing Company, 1994.
- [5] Vicki Bruce, Anne Coombes, and Robin Richards. Describing the shapes of faces using surface primitives. *Image and Vision Computing*, 11(6):353–363, 1993.
- [6] Roberto Brunelli and Daniele Falavigna. Person identification using multiple cues. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 17(10):955–966, 1995.
- [7] Roberto Brunelli and Tomaso Poggio. Face recognition: features versus templates. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(10):1042–1052, 1993.
- [8] K. Bush and O. Antonyshyn. 3-Dimensional facial anthropometry using a laser-surface scanner - validation of the technique. *Plastic and Reconstructive Surgery*, 98(2):226–235, 1996.
- [9] John Canny. A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 8(6):679–698, 1986.
- [10] Peter Champ. Reverse engineering in industrial applications using laser stripe triangulation. In *Colloquium: 3D Imaging and Analysis of Depth/Range Images*, pages 4/1–4/4. IEE Electronics Division, March 1994.

- 
- [11] Shoude Chang, Marc Rioux, and Jacques Domey. Face recognition with range images and intensity images. *Optical Engineering*, 36(4):1106–1112, 1997.
- [12] Rama Chellappa, Charles L. Wilson, and Saad Sirohey. Human and machine recognition of faces: a survey. *The Proceedings of the IEEE*, 83(5):705–740, 1995.
- [13] James O. Coplien. *Advanced C++: Programming styles and idioms*. Addison-Wesley Publishing Company, 1992.
- [14] I. Craw, H. Ellis, and J. Lishman. Automatic extraction of face features. *Pattern Recognition Letters*, 5:183–187, 1987.
- [15] I.D. Faux and M.J. Pratt. *Computational Geometry for Design and Manufacture*. Ellis Horwood Limited, 1979.
- [16] James Foley, Andries van Dam, Steven Feiner, and John Hughes. *Computer Graphics: principles and practice*. Addison-Wesley Publishing Company, 1990.
- [17] A. Jay Goldstein, Leon D. Harmon, and Ann B. Lesk. Identification of human faces. *Proceedings of the IEEE*, 59(5):748–760, 1971.
- [18] Gaile G. Gordon. Face recognition based on depth maps and surface curvature. *The Proceedings of SPIE*, 1570:234–247, 1991.
- [19] Gaile G. Gordon and Luc Vincent. Application of morphology to feature extraction for face recognition. *The Proceedings of SPIE*, 1658:151–164, 1992.
- [20] Gaile Gibson Gordon. *Face recognition from depth and curvature*. PhD thesis, Division of Applied Sciences, Harvard University, September 1991.
- [21] A. Gregory and R.T. Lipczynski. The scanning and three dimensional reconstruction of facial features. In *Colloquium: 3D Imaging and Analysis of Depth/Range Images*, pages 10/1–10/3. IEE Electronics Division, March 1994.
- [22] Darrell R. Hougen and Narendra Ahuja. Shape from appearance: a statistical approach to surface shape estimation. In *Computer Vision - ECCV'96*, volume 1064 of *Lecture Notes in Computer Science*, pages 127–136, 1996.
- [23] Cyberware Inc. URL=<http://www.cyberware.com/>. HTML Electronic Documentation.
- [24] M.S. Kamel, H.C. Shen, A.K.C. Wong, T.M. Hong, and R.I. Campeanu. Face recognition using perspective invariant features. *Pattern Recognition Letters*, 15:877–883, 1994.

- 
- [25] Andreas Lanitis, Chris J. Taylor, and Tim F. Cootes. Automatic face identification system using flexible appearance models. *Image and Vision Computing*, 13(5):393–401, 1995.
- [26] Andreas Lanitis, Chris J. Taylor, and Tim F. Cootes. Automatic interpretation and coding of face images using flexible models. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19(7):743–756, 1997.
- [27] Peter Lindsey and Andrew Blake. Real-time tracking of surfaces with structured light. In *Colloquium: 3D Imaging and Analysis of Depth/Range Images*, pages 5/1–5/4. IEE Electronics Division, March 1994.
- [28] 3D Scanners Ltd. URL=<http://www.3dscanners.com/>. HTML Electronic Documentation.
- [29] David Marr. *Vision*. W.H. Freeman and Company, 1982.
- [30] J.P. McDonald, R. Lambert, and R.J. Fryer. 3D measurement using stereo scene coding. In *Colloquium: 3D Imaging and Analysis of Depth/Range Images*, pages 3/1–3/4. IEE Electronics Division, March 1994.
- [31] Benjamin Miller. Vital signs of identity. *IEEE Spectrum*, 31(2):22–30, 1994.
- [32] Laurent Najman, Régis Vaillant, and Étienne Pernot. From face sideview to identification. In *Image Processing : Theory and Applications*, pages 299–302. Elsevier Science Publishers, 1993.
- [33] Osamu Nakamura, Shailendra Mathur, and Toshi Minami. Identification of human faces based on isodensity maps. *Pattern Recognition*, 24(3):263–272, 1991.
- [34] Joseph H. Nurre and Ernest L. Hall. Encoded moiré inspection based on a computer solid model. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 14(12):1214–1218, 1992.
- [35] George Orwell. *Nineteen Eighty-Four*. Secker and Warburg, 1949.
- [36] Stephen B. Pollard, John Parrill, and John E.W. Mayhew. Recovering partial 3D wire frames descriptions from stereo data. *Image and Vision Computing*, 9(1):58–65, 1991.
- [37] Marc Proesmans, Luc J. Van Gool, and André J. Oosterlinck. Active acquisition of 3D shape for moving objects. In *Proceedings of the 1996 IEEE International Conference on Image Processing*, volume 3, pages 647–650. IEEE Signal Processing Society, 1996.
- [38] O. Rioul and M. Vetterli. Wavelets and Signal Processing. *IEEE Signal Processing Magazine*, pages 14–38, October 1991.

- 
- [39] Marc Rioux. Colour 3-D electronic imaging of the surface of the human body. *Optics and Lasers in Engineering*, 28(2):119–135, 1997.
- [40] Graham Robertson and Ian Craw. Testing face recognition systems. In *Proceedings of the British Machine Vision Conference*, 1993.
- [41] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing Company, 2nd edition, 1991.
- [42] C. W. Therrien. *Decision, Estimation and Classification*. Wiley, 1989.
- [43] Matthew Turk and Alex Pentland. Eigenfaces for recognition. *Journal of Cognitive Neuroscience*, 3(1):71–86, 1991.
- [44] Colin W. Urquhart. *The active stereo probe: The design and implementation of an active videometrics system*. PhD thesis, The Department of Computing Science, The University of Glasgow, June 1997.
- [45] Dominique Valentin, Hervé Abdi, Alice O’Toole, and Garrison W. Cottrell. Connectionist models of face processing: a survey. *Pattern Recognition*, 27(9):1209–1230, 1994.
- [46] Jiang Yu Zheng. Acquiring 3-D models from sequences of contours. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(2):163–178, 1994.