Interface & support hardware for CMOS image sensors using minimum hardware and low bandwidth radio transmission

Andrew A Murray

Doctor of Philosophy

DEPARTMENT *of* ELECTRICAL ENGINEERING
The University of Edinburgh

September 1997

Abstract

Interface & support hardware for CMOS image sensors using minimum hardware and low bandwidth radio transmission

Andrew A. Murray

September 1997

This work investigates the interface between a video sensor and a low bandwidth radio transmitter. In the context of a low-cost low-power radio video link, it outlines a hardware minimal solution.

To solve the bandwidth conflict between the low power radio links and even a modest image sequence quality, a broad range of digital coding techniques are evaluated. Aspects of the coding methods, other than the compression ratios they offer and the ability to implement them using minimal hardware, are considered. Particular emphasis is placed on how vulnerable they leave the coded data to corruption through transmission errors.

Through software simulation, implementations of the most promising compression technique (colour quantisation with error diffusion) is further investigated. Particular emphasis is placed on implementation of the software algorithms using architectures close to those of the simplest hardware implementations.

Colour quantisation with error diffusion is pursued further in the hardware implementation of two algorithms in the form of a field-programmable gate-array (FPGA). The successful implementation of the architecture demonstrates its suitability to hardware implementation. Results from the FPGA offer subjective analysis of the algorithms output at higher frame rate.

A framework that was developed to allow comprehensive subjective testing of image processing algorithms is described, and results, although statistically insignificant, are given.

In evaluating the importance of colour quantisation with error diffusion, amongst other compression and coding techniques, this work concludes that where hardware is at a premium and strict viewing requirements can be met, there are applications where it can be applied profitably, offering results comparable with much more complicated solutions.

Declaration

This work, presented for the degree of Doctor of Philosophy, conforms to the University of Edinburgh's current research degree regulations. It has been composed by me and unless indicated otherwise, the work presented here is original and my own.

Where material has been drawn from the work of others it is duly acknowledged in the text, and the appropriate reference details listed at the end of this work.

Acknowledgements

My thanks go to:

David Renshaw, my first supervisor, without whose continuous support I would never have finished this work. Thanks also to Peter Denyer, both my initial second supervisor and also my industrial supervisor, and Brian Flynn who kindly took over the role of second supervisor.

VLSI VISION Limited for sponsoring me for the first three years, for giving me office space during the majority of my work and for providing me with printing facilities.

EPSRC for funding.

At VISION: Roger Montieth (PC Card driver), Ed Duncan (FPGA & imputer), Martin Turner (imputer), Robert Stevenson (PCB layout), Stewart Smith, JED Hurwitz and Graham Townsend for their continued support, especially for putting up with me during the writing up process.

My contemporaries in the Department of Electrical Engineering: Geoff Jackson, Lynne Munro, Ann Duncan and Andy Myles and also Mike Smart, Drew Holmes, Robin Woodburn and Emma Braithwaite for keeping me sane throughout the years.

Vida the librarian at the Robertson Library and Dot Laing from Electrical Engineering Stores.

My parents Barbara and Stuart Murray for 'hosting' and acquiring many of the subjects used in the subjective tests and also for proof reading.

All who kindly 'volunteered' to rank images for the subjective tests.

Sarah Davis for proof-reading.

Kirsty Mackinnon for her support during some long PhD years.

Victoria Ball for proof reading and all her help over the last few months.

My brother Malcolm for proof-reading beyond the call of duty, advice on the subjective test strategy and much appreciated help both in the logistics executing the tests and in the printing of this thesis.

Finally, thanks to my two examiners, Peter Ivey and John Hannah, for their time, constructive comments and suggestions.

Table of Contents

Abstract .... *i*
Declaration .... *ii*
Acknowledgements .... *iii*
Table of Contents .... *iv*
List of Figures .... *ix*
List of Tables .... *x*

- chapter one **Introduction** -

Chapter outline 1.1
What is a radio video link? 1.2
A definition 1.2
History of video link applications 1.3
More economic realisation 1.4
General architecture 1.4
Thesis objective 1.6
Cheaper radio options 1.7
Communication requirements of the radio video link system 1.8
Radio link technologies 1.9
Modulation techniques 1.10
Data format 1.11
Possible UK frequency bands 1.12
Summary 1.13
Integration 1.14
Integrating the video transmitter 1.14
RF circuit integration problems 1.16
Summary 1.17
Conclusions & thesis structure 1.18
Thesis Structure 1.19

- chapter two **Compression and coding** -

Coding theory 2.2

Compression coding 2.2

Error protection coding 2.5

Coding for ease of reception 2.6

Bit-timing recovery 2.7

Cryptic coding 2.8

Requirements of the radio video link 2.8

Review of error protection techniques 2.9

Systematic error detection and/or correction coding 2.9

Parity codes 2.9

Hamming codes 2.10

Convolution codes 2.11

Redundancy in the coded source data 2.11

Incomplete removal of redundancy 2.11

Over-description 2.12

Techniques of damage limitation 2.12

Transmission of frequency domain data 2.13

Use of fixed-length codewords 2.13

Re-arrangement of spatial data 2.14

Summary 2.14

Review of image compression techniques 2.14

Entropy coding techniques 2.15

Huffman and Shannon-Fano codes 2.15

Ziv-Lempel coding 2.16

Arithmetic coding 2.17

Suitability of entropy coding 2.17

Predictive coding techniques 2.17

Run-length encoding 2.18

Differential predictive coding 2.19

Suitability of predictive techniques 2.19

Transform coding techniques 2.20

Suitability of transform coding 2.21

Sub-sampling techniques 2.22

Frequency sub-sampling 2.22

Colour quantisation 2.23

Vector quantisation 2.24

Conclusions 2.24

- chapter three **Algorithm Design and Evaluation** -

The coding approach 3.2

Colour space quantisation with error diffusion 3.2
Spatial dithering background 3.4
Particular dithering applications 3.5
Alternative dithering algorithms 3.6
Error diffusion 3.8
History of error diffusion 3.9
Error diffusion mechanics 3.9
The importance of viewing conditions 3.10
Summary 3.10
Considerations for minimal hardware 3.10
Algorithm simulation environment 3.11
Design overview 3.12
The coding functions 3.14
Analytical functions 3.15
Tailoring of the algorithm 3.15
Quantiser design 3.16
Implementation options 3.17
Adaptive quantisation 3.18
Relevance to the error diffuser in a video transmitter 3.20
Increased dynamic range problem 3.22
Summary 3.24
Diffusion filter design 3.25
Processing order 3.25
Filter size 3.27
Filter weights 3.29
Summary 3.29
The 'simple' filter 3.30
Improvement on 'simple' 3.38
The 'perturb' filter 3.39
The 'safe perturb 1' filter 3.44
The 'safe perturb 2' filter 3.51
Summary 3.53
Conclusions 3.54

- chapter four **Hardware Implementation** -

Introduction 4.1
Implementation technology 4.2
Processor implementation 4.3
Processor architecture 4.3
Processor logic design 4.7
Prescalers 4.7

Diffusers 4.8
Quantisers 4.8
Rescalers 4.9
The 'simple' processor 4.10
The 'safe perturb 1' processor 4.10
Test system architecture 4.11
Summary 4.15
The slave FPGA system architecture 4.15
The stand alone FPGA system architecture 4.16
FPGA design philosophy 4.17
Internal FPGA architecture 4.18
The processors 4.19
Resolution 4.20
The control logic 4.21
Overall control 4.22
Low-level processor control 4.23
Imputer RAM address generation 4.23
Auxiliary parameter setup 4.24
FIFO management 4.24
Simulation results 4.24
Comments on the tools 4.24
Extracted layout timing limit information 4.24
Version control 4.25
Problems exposed by back-annotation 4.25
Hardware tests 4.25
Test system 4.26
Test procedures 4.27
Initial proof of test system integrity 4.27
Checking algorithm implementation 4.27
Live video input and display 4.28
Results 4.28
'Safe perturb 1' failure 4.28
Perceived display linearity 4.29
Processor speed 4.30
Images 4.30
Conclusions 4.35

- chapter five **Subjective Testing** -

Assessing image quality 5.1
Subjective measures of image quality 5.2
Simple objective meters 5.3

The human visual system 5.3
Objective meters based on models of the human visual system 5.4
Spatial and temporal filtering 5.5
Multi-channel processing 5.5
Correction for non-linearity 5.5
Context masking 5.5
Using objective meters with error diffusion 5.7
Hypotheses 5.8
Experiment design 5.9
Processed vs. unprocessed 5.9
Algorithm comparisons 5.10
Still image sorts 5.10
Source images 5.10
Measures employed to reduce false results 5.11
Test software 5.12
Results 5.14
Algorithm comparison tests 5.14
Still image sorting tests 5.15
Processed vs. un-processed 5.17
Discussion & conclusions 5.18

- chapter six **Discussion and Conclusions** -

Critical review and future work 6.2
Dismissed options for diffusion 6.2
Hardware minimisation 6.3
Alternative architectures 6.4
Pixel level error manipulation 6.4
Context dependent pixels 6.6
Conclusions 6.6

References R.1
Appendix one: Simulation software source code A1.1
Appendix two: PINK FPGA schematics A2.1
Appendix three: Imputer FPGA firmware A3.1
Appendix four: Subjective test software A4.1

List of Figures

Figure 1.1 classic analogue radio video link architecture 1.5

Figure 1.2 constituent parts of a radio video transmitter that employs compression 1.15

Figure 1.3 the video transmitter hardware with an integrated imager, ADC and dedicated coder. 1.18

Figure 2.1 the position of the coding hardware within the architecture of the video transmitter half of the radio video link. 2.1

Figure 3.1 examples of false contouring 3.3

Figure 3.2 examples of manual shading 3.5

Figure 3.3 examples of common dithering techniques 3.6

Figure 3.4 a flow chart representation of the main simulation loop 3.12

Figure 3.5 data flow within the simulation software. 3.13

Figure 3.6 the DOS graphics display. 3.14

Figure 3.7 general quantiser with diffuser architecture 3.16

Figure 3.8 schematic implementations of quantisers using architectures classically associated with analogue to digital conversion 3.17

Figure 3.9 a comparison of uniform and adaptive quantisation 3.19

Figure 3.10 the position of the pre-scaler operation 3.24

Figure 3.11 un-quantised pixels 3.30

Figure 3.12 a schematic of the 'simple' diffusion filter 3.31

Figure 3.13 examples of the 'simple' filter output (lena) 3.33

Figure 3.14 examples of the 'simple' filter output (salesman frame 0) 3.34

Figure 3.15 comparison of simple and floyd-steinberg (salesman frame 0) 3.36

Figure 3.16 a diagram of the 'perturb' diffusion filter 3.39

Figure 3.17 comparison of the results of the 'simple' and 'perturb' diffusion filters (lena) 3.41

Figure 3.18 comparison of the results of the 'simple' and 'perturb' diffusion filters (salesman frame 0) 3.42

Figure 3.19 sections of diffusion pattern from lena reduced to 2 bpp 3.43

Figure 3.20 comparison of the results of the 'perturb' and 'safe perturb 1' diffusion filters (salesman frame 0) 3.46

Figure 3.21 comparison of the results of the 'perturb' and 'safe perturb 1' diffusion filters (greyscale ramp) 3.47

Figure 3.22 comparison of the results of the 'safe perturb 1' and 'simple' diffusion filters (greyscale ramp). 3.49

Figure 3.23 diffusion patterns produced when reducing lena to 2 bpp 3.50

Figure 3.24 comparison of the results of the 'safe perturb 2' and 'safe perturb 1' diffusion filters (greyscale ramp) 3.52

Figure 4.1 data flow within the error diffusion algorithms 4.4
Figure 4.2 architecture of a pipelined processor 4.5
Figure 4.3 hardware implementation of the prescaler 4.8
Figure 4.4 the 'simple' error diffusion processor pipeline. 4.10
Figure 4.5 the 'safe perturb 1' error diffusion processor pipeline 4.11
Figure 4.6 the proposed radio video link architecture 4.12
Figure 4.7 the imputer architecture 4.13
Figure 4.8 the imputer-slave PINK2 FPGA system architecture 4.16
Figure 4.9 the stand-alone FPGA system architecture. 4.17
Figure 4.10 high level internal FPGA architecture 4.19
Figure 4.11 data flow through the pipelined image processors 4.20
Figure 4.12 schematic representation of the FPGA control logic and address generator 4.21
Figure 4.13 the PINK FPGA test set-up. 4.26
Figure 4.14 correction of the perceived linearity of the test system monitor 4.30
Figure 4.15 a comparison of the output of the 'simple' and 'truncate' processors 4.32
Figure 4.16 a horizontal grey ramp processed using the 'simple' processor 4.33
Figure 4.17 comparison of the output of the hardware 'simple' processor with that of its software counterpart 4.34
Figure 5.1 the three test sequences 5.11
Figure 5.2 the window of the subjective test application during display of a set of still images 5.12
Figure 5.3 the window of the subjective test application during display of a moving image sequence 5.13
Figure 5.4 results of the 3 bpp still image sorting test 5.16
Figure 5.5 results of the 4 bpp still image sorting test 5.16

List of Tables

Table 5.1 results of the algorithm comparison tests at 2 bpp 5.14
Table 5.2 results of the algorithm comparison tests at 3 bpp 5.14
Table 5.3 results of the algorithm comparison tests at 4 bpp 5.14
Table 5.4 results from the processed vs. Unprocessed sequence comparisons 5.17

chapter one

Introduction

This thesis investigates obstacles to the implementation of a low-cost radio video link. It considers the interface and support hardware between a CMOS image sensor and a low bandwidth radio transmitter. Successful implementation of this will enable the production of radio video link with significantly lower production and operating costs than any currently available.

Implementation is brought nearer through the development of digital coding hardware, designed to marry a minimal moving-image specification to the likely bandwidths available from a low-cost and low-bandwidth radio link. The result is a hardware minimal solution suitable for integration with the image sensor.

Chapter outline

This chapter sets the work in context in terms of both the technologies involved and potential avenues for realising these aims. It begins by defining the term 'radio video link'. A consideration of current implementations reveals two opportunities for improvement:

1. the use of a low bandwidth/low power radio link.
2. further integration of the system.

Integration of the RF transmitter is shown to be problematic and yield little actual benefit. Instead, the design of a coder which can be integrated on the same die as the image sensor is the most pragmatic route to achieving these ends.

What is a radio video link?

A definition

A video link is a means by which images can be transmitted, permitting the observation of a scene by a distant viewer. This could also be a literal definition of the word 'television'. However, television relays both scenes that are remote either in space or time, whereas video links tend to deal only with live pictures (i.e. the linking of spatially remote places). Where live images are not required, it is normally more cost-effective to record them local to the camera - except under circumstances where the camera is inaccessible or access undesirable (e.g. in a hazardous environment).

A further distinction from television is that video links need not conform to television standards. Such standards (PAL, NTSC etc.) govern high level factors of the images (such as frame rate and image resolution) as well as the actual electrical format of the video signal itself. Conformance to standards allows the simple use of other equipment compatible with that standard. In some situations, however, problems such as limited bandwidth have forced the use of non-standard image formats. This necessitates the use of specialised, and possibly more complicated, image capture and display hardware. The freedom from conformance to standards allows tailoring of the video signal to meet the needs of the applications and the abilities of the available technologies. In breaking away from traditional video standards, there has been an emergence of many incompatible systems. Some standardisation in video conferencing has been brought about through the widespread adoption of the common intermediate format (CIF) and smaller 'quarter-CIF' (QCIF) format. These were devised in parallel with the H.261 video coding standard [*CCITT 90*].

Use of a radio link to make the connection between camera and observation site gives a radio video link. Although linked in terms of communication, the camera and observer in a radio video link are not mechanically tethered. The freedom from physical connection allows the use of links in situations where they would otherwise be impractical (say, where one may move relative to the other, or one is in a sealed environment). It also allows temporary installations to be set up quickly, and offers more freedom when systems need to adjust to suit changing positional needs. There are, however, also disadvantages. The initial cost of a radio linked system tends to be higher than a cabled one (although in some situations the reduced cabling costs of a radio system can offset the increased initial hardware costs - up to half of the installation costs of a close circuit television system can be in its cabling). Another disadvantage is the expense of transmission bandwidth when using a radio link. Unlike a cabled connection, where there is no real restriction on the bandwidth of baseband video, there is a firm relationship between the cost of a radio link and the signal bandwidth it supports. Other potential problems include radio interference, and the complications of radio transmission regulations and licensing.

History of video link applications

Vision is one of the most highly developed of the human senses. Research has indicated that non-verbal communication (facial expression, bodily posture and gesturing) constitutes a large part of communication during face-to-face encounters [*ATKINSON et al 87*]. This implies that when communicating through verbal means alone, such as by telephone or mobile radio, we are somewhat disadvantaged and that communication using a video link could be beneficial. Indeed video links could be put to profitable use wherever the visual observation of a remote site would be advantageous.

Why, when visual communication using video links is technically possible, do we so often make do with poorer means of communication? The reasons are manifold. As with many communication technologies, its widespread benefits cannot be realised without a significant installed base of compatible systems. Why buy a video phone when no-one else has one? There are also problems with the social acceptance of the technology: a commonly expressed fear is that without the visual anonymity of voice-only communication, people consider they would feel more vulnerable in some situations. These are, however, only generalisations and exceptions do occur.

Applications exist in consumer electronics, such as door-entry phones, baby monitors and surveillance, where installed base and the social acceptance problems do not apply.

Historically, the expense of the equipment involved has limited their employment to commercial 'high return' applications such as surveillance in the security industry and to relay pictures from 'outside broadcasts' or long-distance interviews in television production. With the advent of the camcorder, however, video equipment is becoming available a at more reasonable cost. Together with developments in the telecommunication industry, this has led to more frequent use of video links in the form of video phones and video-conferencing facilities. Although some low cost video-conferencing systems have recently become available[1], the cost of purchasing and operating these systems remains prohibitively expensive for many applications, restricting them to being the toys of the rich consumer and the tools of the wealthy business person.

More economic realisation

One of the major remaining obstacles to the widespread adoption of video links must be a lack of available equipment at a low enough cost for the consumer to perceive that it is worthwhile. With a view to realising a radio video link more economically, common implementations are introduced below and possible areas for cost reduction explored. The basic architecture of a radio video link is first examined.

General architecture

The basic architecture of a radio video link is shown schematically in Figure 1.1 below. With image data flowing from left to right, each block in the schematic represents a distinct processing task. These tasks are: sensing of the image

[1] These include: Creative Labs Sharevision 3000, BT Relate 2000 Videophone, Vivitar Motion Picture Phone, and Creative Labs Webcam.

information, control of the sensor array to produce a single, time-varying video signal, radio frequency (RF) modulation, RF demodulation, and image display.

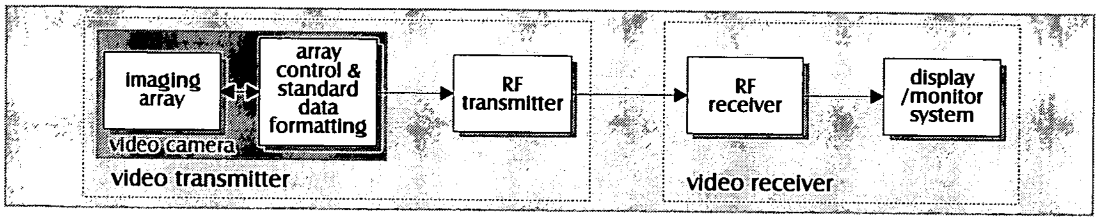


*Figure 1.1 A block diagram of the **classic analogue radio video link architecture**. The darker background highlights the portion of the transmitter end of the link that is commonly implemented using a video camera.*

Although this architecture is common to nearly all systems, it can be implemented in many different ways. Conventionally, implementations have concentrated on transmitting video in the format of a full television-standard signal (e.g. NTSC or PAL). This is achieved using a standard video camera as the image source and transmitting its analogue output signal using a high bandwidth radio transmitter (baseband video signal bandwidth is approx. 4 MHz). Such systems produce high quality video and because they use a standard video format they are convenient for use with other standard pieces of video equipment. However, the high camera and radio costs and the licensing requirements of these systems make them expensive. In addition, the high bandwidth they use limits the density with which they can be operated without interference.

Tailoring features of the video format (such as image resolution and update rate) to suit the needs of a particular application limits the general application of the system and the convenience with which it can be used with standard pieces of video equipment. However, if the tailoring process reduces the bandwidth required to transmit the signal, then savings can be made both in radio hardware costs and the amount of power required for the transmission. In addition to the crude reduction of bandwidth through lowering image resolution and update rate, compression techniques can be used to lower the raw image bandwidth, although at some cost in image processing. The often sophisticated techniques of lossless compression may prove to be prohibitively expensive, however, it may be possible to employ lossy algorithms to achieve savings in bandwidth without necessarily losing significant perceived image quality.

The use of a specialised video format with low bandwidth is an approach shared by some of the video phone systems based around modem connections that have appeared recently (see [1]). These systems use either dedicated hardware or are designed to interface with a personal computer (PC). They use specialised (and often proprietary) video formats to achieve the required image quality over the relatively low bandwidth of the modem connections (typically between 9600 and 33000 baud). As these systems achieve the low bandwidth through the use of sophisticated compression algorithms executed on digital signal processors or microprocessors their systems costs are high. To remain low cost, the implementation of a radio video link using low bandwidth would have to achieve the lower bandwidth through hardware-minimal means, rather than employing complex compression algorithms.

A second approach that is likely to bring cost savings is the integration of the radio video link hardware. Integrated circuits (ICs) specifically designed for an application can often be tailored to make them cheaper than using a combination of standard parts. In addition to the production of dedicated ICs, it is likely that the costs of manufacture could also be lowered by implementing many of the functional parts that would normally be fabricated on separate ICs (if not separate PCBs) on the same die. The use of application specific ICs to achieve cost savings must, however, be justified by high volumes as the costs of the design and prototyping process are significant.

Two areas that could offer reductions in price, size and power consumption are: the use of lower power, lower bandwidth radio links (through systems tailoring and compression/coding) and the further integration of the system by combining parts on a single IC.

Thesis objective

This thesis tackles the problem of implementing a radio video link at low cost. It continues previous work that resulted in the development of a entirely integrated monochrome video camera [*RENSHAW & DENYER et al 90*]. It concentrates on the cost reduction of the sensor end of the link, with the aim of designing an integrated solution. The work centres around the definition of the interface between sensor and transmitter that is required in order to deliver a radio video link using a much lower bandwidth than is common. The ultimate aim is to arrive at a simple solution that is both low power and can be integrated on the same die as the video sensor.

Unlike the problem of integrating the video camera, which had well defined goals, the design of the sensor/transmitter interface requires identification of the exact role of the interface.

The work concentrates on investigating the integrated, tailored approach, outlined above, attempting to achieve the cost savings both through integration of the circuits and the use of a low bandwidth and low power radio link. A detailed introduction to these two problems is given below.

Cheaper radio options

Employing a low power and low bandwidth radio link was cited above as a means of decreasing the cost of the radio video link hardware. The reduced initial costs can be achieved through a combination of using very simple RF circuits, the use of low bandwidth components and components that need not handle large amounts of power. Low power transmission also implies low power consumption and thus low running costs. All these advantages are in keeping with the size, power and economic aims of this project. Low power and low bandwidth can also be considered a more 'responsible' solution; bandwidth is a scarce resource that is under increasing pressure and low power reduces the potential for causing interference to other users of the same part of the electromagnetic spectrum.

Cost and power consumption are not the only considerations, however, when designing or choosing a radio link. Many other factors distinguish radio links and have a bearing on their suitability to particular applications. Unfortunately, low power and low bandwidth have consequences on some of these factors, especially some features of link performance. Low power transmission is inherently more susceptible to radio interference. Low bandwidth offers low rates of data transmission.

Below, the likely communication requirements of the radio video link are identified. Through the introduction of some of the other important factors involved in selecting a link technology, and a review of a few of the low power radio options available in the UK, the consequences of going down the low power and bandwidth route are explored. The requirements of the radio video link and the abilities of the cheaper link options are compared, emphasising the implications on the remainder of the video link system.

Communication requirements of the radio video link system

In the radio video link application, the basic requirement of the radio link is to communicate image data from the camera to the observation site. Other uses could include communication from the observation site to control the camera. Assuming a system in which the transmitter end of the link autonomously sends image data to the observation site, a one-way (or *simplex*) link is all that is required.

The bandwidth of the raw image sequence is dependent on its spatial resolution, colour depth and frame rate. These factors are dependent on the particular needs of a link.

The primary applications driving this research are low-cost video links for consumer markets (such as the toy market). An ideal image quality would be something similar to a PAL video signal. The image would be something of the order of 512x512 pixels, and would be updated 25 times per second. Unfortunately this equates to over 6 million pixels per second, a rate unsustainable by the sort of link considered here. Due to this high implied data rate, it is unlikely that consumer links will be high-fidelity for some time to come.

For the purposes of this investigation, a benchmark minimum image-quality specification has been set. The frame rate minimum is set at 10 frames per second (fps), as lip-sync (the ability to correlate the movement of someone's lips with the sound of their speech) is generally considered to be impossible to achieve below 10 to 15 fps. The image size minimum has been set at 64x64 pixels. This is much lower than ideal, but sufficient to convey a fairly detailed face or a simple scene. The pixels specification is 6 bits per pixel (bpp), monochrome. The use of a monochrome image ties in with the intended image sensor, and 6 bpp gives a greyscale resolution that approximates the limit of the human visual system viewing a CRT [*RUSS 95*]. This specification implies a raw data rate of 240 kbit/s, which is roughly the capacity of 3.75 standard ISDN2 lines (64 kbit/s each), but is two orders of magnitude less than that of raw broadcast-quality digital video [*POYNTON 96*].

Image quality will also be affected by any errors introduced in the transmission/reception process. Many meters are available for the objective measurement of distortion (such as bit-error rate). However, as correlation between these criteria and perceived image quality is low, no strict specification was defined.

Like bandwidth, the range of the system is dependent on the particular application, however it is unlikely that it would be less than 100m. If such a short link was necessary, a cable solution would probably be more economical. For the purposes of the investigation, a minimum range of 100m will be set.

Radio link technologies

A large number of features distinguish radio links. These include their frequency of operation, the RF bandwidth they use, the modulation techniques employed, the output power of the transmitters, the component technologies used and particular details of the RF circuits employed. These features affect the link in terms of its range, the available data bandwidth, the likely levels of distortion and the costs of both implementation and operation. These factors therefore have a direct impact on which applications different radio links are suitable for.

In practice, the options open to the link designer are not as free as the long list of seemingly independent factors above might suggest. This is due to restrictions imposed by the regulation of radio transmission. Organised use of the electromagnetic spectrum is required to minimise interference, allowing successful simultaneous radio communication by numerous parties. The majority of this organisation is imposed through government legislation which divides the spectrum into discrete frequency bands and stipulates factors such as who can transmit in each band, what types of data can be transmitted, which modulation techniques can be employed and what limits there are on transmitted power (e.g. [*RA 89; RA 92*]).

With the advent of the mobile phone and other internationally widespread radio systems, significant advances towards standardisation have been made across many radio bands in past years. However, government regulation in many frequency bands still varies from country to country. Different regulations make the development costs of systems in some bands more significant as the application of the technology (and thus the potential volume) is more limited.

The choice of radio link hardware is thus dependent on a combination of the features required of the link (range, bandwidth etc.) and what it is permitted to use under the local government regulations.

Before a brief summary of regulations of some of the UK radio bands that may be suitable for the radio video link, two important fundamental aspects of the radio link (the modulation technique and whether data is transmitted in analogue or digital form) are discussed below.

Modulation techniques

The modulation[2] techniques used in radio communication are classified according both to the type of carrier signal used and the method by which the carrier is modified by the data. The carrier can either be a continuous signal (commonly a sine wave) or a train of pulses. Modulation can be achieved by modifying the amplitude of the carrier (amplitude modulation, AM), its phase (phase modulation, PM) or its instantaneous frequency (frequency modulation, FM). In practice, many modulators employ a combination of these basic techniques.

[2] Modulation is the process by which the signal that is to be transmitted is used to modify a high frequency 'carrier' signal in order that it can be communicated at radio frequency. This is the essence of radio transmission.

In its simplest form, continuous wave AM is the least complex modulation technique to implement. It is, however, wasteful of both power and bandwidth. Single side-band can be used to halve the bandwidth (and thus the transmitted power) and wasted power can be reduced further by suppressing the carrier signal, but these techniques require more complicated circuits. Wideband FM offers comparable power efficiency to single side-band AM yet uses relatively simple circuits. It also exhibits improved noise rejection as its demodulation is not dependent on the amplitude of the received signal. Pulsed carrier schemes offer even better noise suppression, however, they incur more circuit complexity[*SMITH 86; KRAUSS 80*].

As in many mobile applications [*SMITH 86*], FM is probably most suited to the radio video link due to the relatively simple circuits required, its power efficiency and its resilience to noise.

Data format

Most modulation techniques permit the transmission of either analogue or digital data, although many implementations of the techniques are more suited to one in particular. Consideration as to which format should be used in the transmission of the image information of the video link is given below.

Nearly all physical phenomenon that we want to measure are themselves analogue (in that they are continuously variable over some range). An analogue representation is thus a natural way to describe them. Pixel information is no exception. Conversion to a digital representation requires hardware and introduces sampling errors or 'quantisation' noise. Despite this, two key advantages of transmitting data in a digital form make it more suitable for many applications. The first advantage is the inherent noise immunity that digital data has over analogue. This is important in the radio video link application as the low power of the link will make it susceptible to interference (and thus distortion). The second, and more compelling, reason in the case of the radio video link is that the vast majority of the techniques available for both improving noise immunity and offering data compression are digital coding techniques - thus requiring that the data be in a digital form both prior to and after transmission (for coding and decoding). In addition to the sheer proliferation of digital coding techniques, they are also convenient in this application as the logic required to implement them can easily be fabricated using the same standard CMOS technology as the image sensor.

A further advantage of digital transmission in terms of the desire for a low power radio video link system is that very efficient power amplifiers can be fabricated for digital transmission [*SMITH 86*].

Possible UK frequency bands

Transmission of digital data using continuous wave FM has been shown to be the most suitable solution for the radio video link in terms of technology. Attention now switches to a practical solution. For a system to be implemented there must exist a radio band that allows the transmission of image data using continuous wave FM at a convenient bandwidth. Possible frequency bands available under UK legislation are considered below.

Obvious candidates are the license exempt bands. These radio bands are specifically designed for the transmission of low power, low bandwidth data. They are 'license exempt' in that although circuit designs require to be licensed, individual radio units need not. MPT1336 and MPT1340 are two such bands in the UK [*RA 89; RA; 92*]. Their relaxed regulation allows the use of inexpensive RF components. However, the same degree of regulation also makes users of these bands prone interference from other systems.

Unfortunately the bandwidths permitted for telemetry[3] within the current license exempt bands is typically around 10 KHz. Using simple FM modulation, this translates to a data bandwidth of approx. 10 Kbits/s - far below the 250 Kbits/s implied by the radio video link specification outlined earlier. Another factor that

[3] 'Telemetry' (as defined in Radiocommunications Agency regulations) is the use of telecommunication for automatically indicating or recording measurements at distance from the measuring instrument [*RA 87*].

prohibits their use is that the license exempt bands are not intended for continuous transmission.

A telemetry band at 2.4 GHz has recently been opened up in the UK, intended for spread-spectrum communication [*PICKHOLTZ et al 82; TSUI & CLARKSON 94*]. This band does not suffer from the bandwidth limitation of the license exempt bands (it will support data rates in excess of I Mbit/s). However, the complexity of spread-spectrum transmission makes both the initial costs and running costs of the hardware expensive.

It is believed that other radio bands are being released for telemetry in the UK. Existing pressure on the spectrum means, however, that the bandwidth of any such new bands will be unlikely to be more than 100 - 150 KHz. (This is with the exception of bands at around 2.3 - 5 GHz - however transmission at these frequencies suffers interference from microwave ovens, problems of multi-path transmission and fading.) If bands of around 125 KHz bandwidth are opened up they would offer transmitted data rates of approx. 125 Kbits/s using FM modulation. This is still lower than the 250 Kbits/s of the radio video link specification, however only by a factor of around 2:1.

Summary

There are advantages to the radio video link application in using a low power, low bandwidth radio link, namely: savings in initial component costs and in the running costs. However, there is a conflict between the desire for low bandwidth and the desire to communicate a relatively high bandwidth signal, such as is represented by the minimum video link image specification outlined earlier.

If a low bandwidth link is to be used, then reduction in the bandwidth of the image data will be necessary to address this mismatch. This can be achieved either by reduction in the minimum image specification or by employing compression techniques to remove some of the redundancy in the image data. As the image specification is already tight, it was decided to use compression rather than lower the specification further. Many techniques exist specifically for the systematic removal of redundancy, producing 'compressed' data sequences. The degree of compression required by the mismatch between image specification and likely available bandwidth (around 2:1) is relatively low and could be achieved by many common compression techniques. If the compression is to be realised without compromising the low power and cost design goals of the system, a compression technique that is very cheap to implement will have to be found.

Integration

Integration is a common method of reducing costs used in the electronics industry. In high volume, the production costs of integrated circuits (ICs) are far lower than their discrete counterparts[4]. Power consumption also tends to be less as integrated circuits have lower internal circuit drive requirements. Systems built from ICs also tend to be smaller and lighter. Design and prototyping are the only areas of high cost, hence the requirement for high volume if unit cost benefits are to be realised.

Possibilities for realising cost, size and power benefits through integration of the constituent parts of the transmitter end of the radio video link are considered below.

Integrating the video transmitter

Conventionally, the hardware at the transmission end of a radio video link is constructed from a number of ICs and many discrete components. These normally populate more than one PCB which are often housed in separate enclosures. One reason for this style of construction is the marrying of quite separate technologies (video and radio), where the various components are typically constructed by different manufacturers.

As more specialised systems develop this situation is changing (e.g. [*STERN et al 95*]). Several of the system components of the video transmitter have already been

[4] Ever-increasing integration on a single die does not guarantee costs savings. With large circuit areas die yield can become a problem [*SZE 88*].

integrated further. The entire circuitry of a video sensor has been integrated onto a single CMOS IC [*RENSHAW & DENYER et al 90; MENDIS et al 93; ACKLAND & DICKINSON 96*] so there is no longer a need to use a separate standard CCD imager and support ICs. Integrated digital signal coders for transmission, including a complete spread spectrum transmitter [*CHIEN et al 94*], have also been demonstrated.

In a conventional implementation of a video transmitter, each of the constituent parts of the system (shown earlier in Figure 1.1) would be realised using one or more separate integrated circuits often on different PCBs. Adding of a digital compression stage to the system leads to a sensor-transmitter architecture, such as that shown in Figure 1.2. Again, using conventional components this would be implemented using separate devices for each functional block shown.

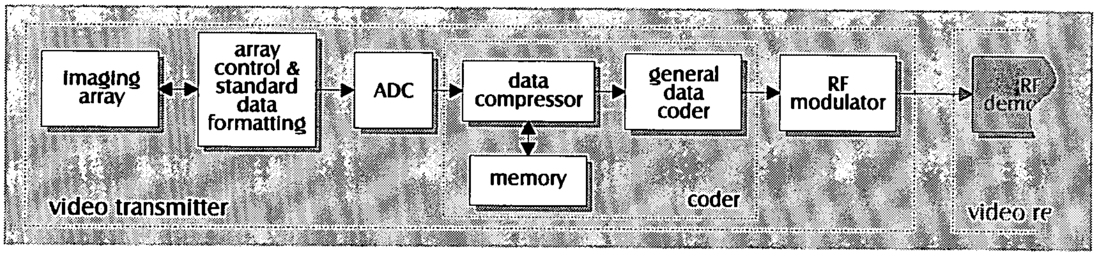


Figure 1.2 *A block diagram of the* ***constituent parts of a radio video transmitter that employs compression*** *(an expansion of the transmitter side of Figure 1.1, the start of the receiver is shown shaded). This architecture differs from that in Figure 1.1 in that the RF transmitter has been divided into a general channel coder and separate modulator, and in the addition of a digital data compression engine. Addition of the compression engine would typically require the addition of the ADC to digitise the analogue video signal and also provision of a significant amount of memory for use during the compression operation. Again, each of the parts of this system would conventionally be implemented using at least one separate IC, possibly on different PCBs.*

The ultimate integrated solution would be to construct the whole of this sensor-transmitter system on one IC. Integration of an entire system permits the tuning of its components to meet only the needs of that particular system. Component interfaces that would normally be generalised for use in various systems can be pared down to leave only that required for the individual application. For example, in the radio video transmitter the picture information need never exist as a standard composite video signal - saving both on signal formatting and decoding hardware.

Creating a one-chip video transmitter would require integration of a version of the existing CMOS imager with a specialised data coder and an RF modulator. Manufacture of a data coder on the die of the imager should not pose a problem as the imager is fabricated on a standard CMOS. Interfacing of the coder to the data stream of the camera will require inclusion of an on-chip analogue-to-digital converter (ADC). The manufacture of ADCs with sufficient performance has already been demonstrated on a standard CMOS process [*CHEN et al 90; ACKLAND & DICKINSON 96*]. Problems stemming from the inclusion of a digital coder on the same die as the sensor and ADC are only likely to occur if the coder is either particularly large or power hungry. Large coder size could lead to a die size that may imply yield problems. High power consumption could lead to problems of cross-talk through power supplies, or problems of high sensor dark current due to heat generated by the coder (a component of imager dark current is dependent on device temperature). Integration of the RF modulator is likely to be more problematic than the coder. Two areas in particular complicate its integration with the rest of the system. These are, firstly, the differing fabrication needs of some of the likely components involved and potential noise problems between the power output stage, and secondly, the delicate image sensor array. These two areas are expanded upon below.

RF circuit integration problems

If the whole of a circuit is to be integrated on the same die then it must be possible to produce all the components of the circuit using the same fabrication process (and produce them at the required quality). Unfortunately, some components such as those that operate at very high speeds, those that require a very low noise environment and those whose operation relies on unusual electrical effects[5], have radically different construction needs. Some are impossible to create on particular standard fabrication processes, others are difficult to produce reliably. Conflicts can sometimes be overcome through the re-design of circuits so that they only use components that are available on a single fabrication process. Another alternative is to permit fabrication of unusual structures by adding processing steps to an otherwise normal fabrication process. Any movement away from a normal

[5] That is electric effects *unusual* to normal IC fabrication processes.

fabrication process (such as standard CMOS), however, incurs initial set-up costs and makes manufacture of the design less portable between fabrication facilities.

Some of the components used in the simplest RF transmitter circuits make them prone to these problems. Simple transmitters typically exploit the properties of components that cannot be fabricated using a standard CMOS process - such as SAW resonators and very high-speed transistors [*NIETROJ 90*]. Although the adaptation of these circuits may be possible (through the construction of non-standard components using what would normally be considered to be parasitics of a standard process) it is outwith the scope of this thesis.

Integration of high speed transistors with logic can be achieved using a fabrication process such as BiCMOS. There are typically only a few high speed transistors used in the simple transmitter circuits, however, the economic advantage of integrating them with the logic through the use of a BiCMOS process is questionable. BiCMOS processes tend to be significantly more expensive than CMOS.

The second fundamental problem in integrating the radio transmitter along with the rest of the transmitter end of the radio video link is that there is an inherent conflict between the needs of the imager and the job of the radio transmitter. One is a delicate sensor and the other an intentional radiator of energy. Although ultimately they are concerned with quite separate parts of the electromagnetic spectrum (visible light and VHF radio) a degree of cross-talk between the underlying electronics is inevitable. This problem could be alleviated to an extent through the maximum physical separation of the two circuits on the die, the isolation of their power supplies and possibly keeping the final power output stage of the transmitter off the chip. These measures, however, may not solve this problem completely.

Summary

Integration can offer benefits in terms of manufacturing costs, power consumption and system size. However, integration of all circuits is not simple and problems such as cross-talk can result.

The practicality of integrating the simple RF circuits considered for this application (e.g. [*NIETROJ 90*]) using current fabrication processes is questionable, and the cost benefits doubtful. Integration of the image sensor, ADC and data coder that represent the remainder of the transmitter end of a radio video link is however, eminently possible. The hardware architecture of a video transmitter with such an integrated imager/ADC/coder is shown in Figure 1.3.

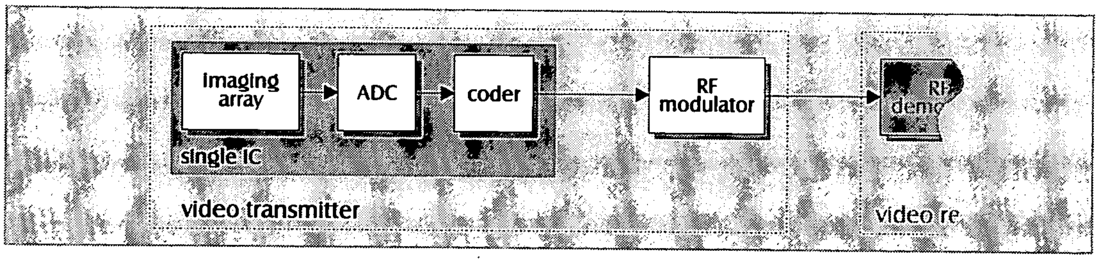


Figure 1.3 *Schematic representation of* ***the video transmitter hardware with an integrated imager, ADC and dedicated coder.***

Conclusions & thesis structure

To realise a radio video link at lower cost, this thesis investigates the practicality of a low power, low bandwidth, integrated implementation. It concentrates on identifying the necessary features of a coder required to marry the minimum image specification to a radio link with between half and a quarter of the necessary bandwidth to transmit the raw signal, and with the design of such a coder. Development of the coder was chosen as, after the success of the integrated video sensor, it is the next step towards a completely integrated radio video transmitter. Successful integration of the coder was deemed to be more likely and bring about more cost savings than attempts to integrate the radio transmitter. In offering a successful implementation of a particular hardware-minimal coder solution, the investigation shows that a minimal hardware approach is viable. In addition to application in a radio video link, the work on image data coding is equally applicable to any application where image data is to be reliably communicated using minimal hardware.

Thesis Structure

The remainder of the thesis begins with chapter two which gives an introduction to the general subject of data coding prior to transmission. Particular reference is made to compression techniques and considerations of the vulnerability of data to transmission errors. This chapter ends with a review of existing compression and coding techniques. The most promising of the techniques presented in the review is then further investigated in chapter three. The detailed hardware implications of its approach are explored and particular implementations tested through software simulation.

Chapter four documents the implementation of two of these algorithms in the form of a field programmable gate-array. The implementation serves to prove the validity of the hardware minimal approach and allows evaluation of the output of the algorithms at higher frame rates than were permitted by the software simulation.

More in-depth subjective analysis of the algorithms is considered in chapter five. This takes the form of a programme of subjective tests. The work is concluded in chapter six. The success of the hardware minimal interface is evaluated and suggestions are made for future work (including some radical alternatives to the approaches adopted here).

chapter two

Compression and Coding

In chapter one coding was outlined as a potential solution to the mismatch between the raw data bandwidth required by the video link and the low capacities of cheaper radio links. The coding hardware was shown as a discrete block positioned in the digital data path between sensor and transmitter, as shown in Figure 2.1.

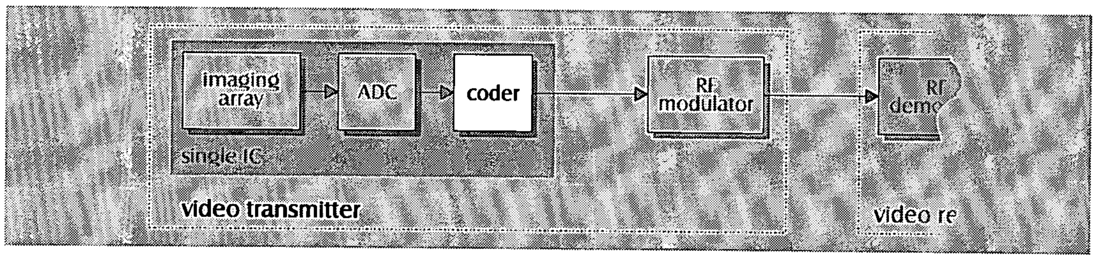


Figure 2.1 ***The position of the coding hardware within the architecture of the video transmitter half of the radio video link.***

Compression is not the only purpose of pre-transmission coding techniques. This chapter introduces all the considerations when preparing a data stream for transmission.

The chapter begins by introducing pre-transmission coding, outlining its basic purposes. The importance of these objectives in the case of the radio video link is considered. Existing error protection and image compression techniques are reviewed with a view to selecting suitable candidates for use in the radio video link. The modest compression requirements of the application mean that the implied hardware costs of each compression approach and the error vulnerability of the code it produces are the main criteria for selection. Conclusions are drawn as to which particular coding techniques may be suitable for the radio video link application.

Coding theory

Techniques of data coding are often employed before data are transmitted. This section introduces the purpose of these techniques. As such, a fairly abstract view of data communication is taken, with different aspects of coding explained with reference to a general communication system that is concerned with the transmission of messages.

The introduction is arranged according to the objectives that pre-transmission coding can address. These are: transmission efficiency (compression), communication reliability, simplicity of reception and privacy. Some coding techniques address many of these issues, others concentrate on one in particular. All the issues, and thus the techniques, are concerned with the description of the original data set (the source data) during transmission and to some extent are therefore interdependent.

Compression coding

The efficiency with which messages can be transmitted (transmission efficiency) is directly dependent on the amount of data that needs to be transmitted in order to communicate the message, given the code that is used to describe it. The object of compression is to reduce the amount of data that is required to describe the message by using a code that describes it efficiently - thus permitting communication of the message at lower cost. In addition to its use in improving transmission efficiency, compression is often used to improve storage efficiency. This application is not considered here.

With most types of data, the form of coding that is generally used to represent it involves much redundancy[1]. In addition to the efficiency of its coded size, factors such as the ease of the coding and decoding operations and the manipulation of the coded data have also to be considered. It is only when the costs of communication or storage of the coded form of data become significant that particularly efficient coding schemes tend to be considered.

Image data is a good example of a type of data that is commonly coded with much redundancy. Its standard raw format is a raster scan of picture elements (or pixels). Each element is either described to the capabilities of the display device, or with reference to a limited 'palette' of colours. Although this raw form is convenient in many applications (as it relates directly to the hardware architectures of many imaging and display technologies) it tends to lead to much redundancy. The use of colour 'paletting' is a form of compression. It is typically employed to reduce the amount of memory required in display hardware.

Many compression algorithms exist that are general to all types of data, requiring no prior knowledge of the data that is to be compressed. Some of these rely on exploiting forms of redundancy that are common to most types of data and encode the data on a casual basis, others can achieve higher coding efficiency by optimising the form of the code to suit the particular data of each message. The latter, 'adaptive' schemes are more expensive to implement, however, as they must analyse the data set prior to encoding.

In contrast to this general approach, there exist many compression techniques that are designed to exploit characteristic mathematical structures that are found in particular types of data. Such algorithms can achieve high compression ratios without necessarily employing the amount of data analysis used by the general adaptive schemes described above. Their success, however, is limited to use on data sets that exhibit the characteristic structure that they have been designed to exploit.

[1] In this context the term *redundancy* is used to describe features of the coded message that bear no *new* information in relation to communication of the message.

With the increased use of digital images, a combination of the high costs of storing and transmitting digital images in their raw format, the needs of applications (such as video conferencing) and the availability of digital coding hardware (e.g. DSPs), much research has been carried out into coding techniques particular to image data that offer compression. As a result, there now exist many specialised image compression algorithms. Some are further specialised by being tailored to image data from sources with particular characteristic features (used in the compression of data from sources such as weather satellites).

In common with all compression techniques, image compression techniques are based both on alternative methods of coding data and on techniques which irreversibly discard some of the source information. Techniques that achieve compression only through the use of coding processes that are fully reversible are termed *error-free* or *lossless*, those that do not preserve the original image data perfectly are termed *lossy*. This is an important distinction as, although lossy techniques offer much higher compression ratios, they are not always suitable for use in applications where images need to be carefully scrutinised or relied upon for legal evidence.

The benefits of compression do not come without cost. Both compression and decompression of data take time, they employ hardware and use power. In addition, where the techniques used are not fully reversible, a loss in perceived image quality may be incurred. Unlike the other costs, the measurement of losses in perceived image quality is non-trivial, by definition it is not a quantity that can be directly measured by some objective means. This subject is investigated further in chapter five.

Selection and design of compression algorithms for a specific imaging application therefore involves consideration of which type of algorithm might suit the nature of the image source, what degree of compression the application requires, the resources available to perform the compression and whether any losses in image fidelity can be tolerated.

Error protection coding

The inherent noise in any practical transmission channel means that in any transmission system there will be a certain level of errors in the communication of data. The vulnerability of a coded message to corruption through errors in its communication is therefore an important consideration.

Transmission errors, by definition, affect the integrity of the coded message that is communicated. How these errors in the coded form of the message affect the decoded message is dependent on the coding scheme used. Thus it is the coding scheme that controls the ultimate effect of transmission errors. This property is an important aspect of its design.

There are three distinct approaches to lessening the effect of transmission errors. The first is to code so as to allow the presence of errors to be detected, the second is to code so as to allow correction of a level of errors and the third is neither to attempt detection or correction of errors, but to code the message so as to minimise the effect that any errors have in the *perception* of the decoded message. The first two approaches involve dedicated coding, and are general to all types of data as they are concerned merely with the integrity of the coded message and not with its content. The third approach can also be implemented using a dedicated coding stage. However, as most coding schemes affect the vulnerability of the messages they describe, the third method can be achieved as a consequence of a coding method whose primary aim is not for error protection (or through minor modification of such a coding stage). As some understanding of the effect of different errors in the content of the message must be understood before attempting to minimise the effect of errors in its coded form, the third approach is dependent on the particular data type.

Features of the application may determine which protection approaches are suitable. These include the tolerance of the application to a level of errors or missing data and the ability of the receiver to request the re-transmission of corrupted data.

In general there is a conflict between the aims of coding for compression and those of error protection. Where compression centres around the removal of redundant data, error protection is concerned with the use of redundancy for protection. The increased significance that each element of a compressed data set has in conveying the original message often makes it more vulnerable to transmission errors than the message was in its raw form. Achieving a high compression ratio at the expense of making the coded data exceptionally vulnerable to corruption may be pointless if the characteristics of the transmission channel dictate a subsequent level of coding for error protection that much of the transmission efficiency gain is lost. Compression is often considered in isolation from susceptibility of the data to errors. However, the interdependence between compression and error protection means that where data is to be both compressed and transmitted the vulnerability of the data to transmission errors must be considered at the compression stage if a balance between these two goals is to be achieved.

Although the ultimate aims of compression and protection may conflict there are some techniques used in compression that produce data that is more resilient to the effects of errors. These include transforms of data into alternative data spaces in order to facilitate quantisation. Described in this alternative form, each element of the original message is dependent to a small extent on many of the elements of the coded message. The effect of any error in the transmission of the coded message is thus distributed thinly over a large part of the decoded message [*PRATT 69*].

It is important to note that any error protection scheme can only offer a finite level of protection, and therefore that there will always be a trade-off between the level of protection and the cost of the scheme. The costs of dedicated coding to provide error protection come both in increased computation (hardware, power and time) and reduction in storage and transmission efficiencies.

Coding for ease of reception

The ease with which a transmitted message is received, and thus the complexity of the necessary receiver circuit, is affected by certain factors of the data coding used. As discussed above, the use of any compression and/or error protection coding has consequences in terms of the hardware necessary for decoding. Aside from these direct implications, there are other factors of the data sequence that affect the simplicity of the receiver. In particular, aspects of the code govern the methods available to the receiver to recover the necessary timing information required to sample the data sequence in order to recover the data successfully. This factor is considered below.

Bit-timing recovery

When communicating with a remote device there is no common *system clock* with which the timing of individual bits in the data stream can be established. The remote device must decide when it should sample the received signal in order to reliably recover the 'bits' of the data sequence using features of the signal itself.

There are two systems commonly employed to govern sample timing. They differ in complexity of the synchronisation with the data sequence and are called *synchronous* and *asynchronous* modes of transmission. With asynchronous transmission individual transmitted characters are preceded by start symbols and followed by stop symbols. The receiver assumes a symbol transmission frequency and, using a local clock, times sampling points with reference to an edge within the start symbol. As all samples are made with reference to the start symbol, errors in the accuracy of the timing limit this mode of transmission to relatively short symbol sequences (e.g. eight). In synchronous reception the point at which the received signal is sampled is controlled by a local oscillator which is kept synchronised to the frequency and phase of the data sequence itself. This is accomplished through the use of either digital or analogue phase-locked loops (PLLs) to synchronise to edges within the data sequence. Dedicated characters are often used at the start of data sequences in order to ensure that word/character boundaries are correctly interpreted.

Asynchronous transmission is simpler to implement as it does not require the same degree of synchronisation. The requirement for frequent synchronising symbols limits the efficiency with which it can be used to transmit data. This mode tends to be used primarily in situations where the data is transmitted at random intervals. In application where the data rate is more predictable or communication at a high bandwidth is required the efficiency of synchronous transmission is generally favoured [*HALSALL 88*].

Although coding for ease of reception has cost implications in terms of transmission efficiency, it is generally independent of other compression and error protection considerations.

Cryptic coding

The use of obscure or complicated coding can offer the data a degree of privacy by making it difficult for a third party to understand the transmitted message without some prior knowledge of the code. This use of coding is generally termed encryption and is often associated with the world of espionage. With the increase in electronic communications its use as a method of combating fraud is becoming common (in areas such as mobile telephone billing and financial transactions).

As system security is not a high priority for the radio video link application, the encryption aspects of coding will not be considered further.

Requirements of the radio video link

Given the mismatch between the bandwidth implied by the minimum specification and that of likely cheap radio links, compression is probably the most important of the four aspects of coding outlined above for the radio video link. Both the tolerance of the coded data to errors and the ease with which the data can be received are also important and will therefore also have to be considered.

The drive for a hardware-minimal radio video system is likely to be the main restricting factor in the choice of coding techniques. In addition to favouring techniques that are computationally simple, the minimal hardware approach also favours those that use the least memory.

Ease of reception will require some dedicated coding prior to transmission. Synchronous transmission is suggested by the nature of the communication (continuous transmission of a relatively high-bandwidth stream of data). The addition of sync and framing characters will be required immediately prior to transmission and need not significantly affect transmission efficiency. The needs of compression and error protection can therefore be considered independently from the ease of reception coding needs.

The radio video link is an application concerned with the transmission of images to be viewed by human observers. As such, a degree of data fidelity loss through the use of lossy compression techniques can be tolerated. As image details are unlikely to be scrutinised, the application may well afford the use of techniques that lead to a perceivable degree of image degradation.

Before compression techniques are reviewed, techniques for reducing the vulnerability of the coded messages to transmission errors are now considered.

Review of error protection techniques

It was stated earlier that the types of data coding used to describe a message determine how it is affected by errors in its transmission. This section considers various approaches to intentionally reducing the vulnerability of a coded message to corruption by transmission errors.

The techniques are divided into three groups: those that protect data through the systematic inclusion of redundancy, those that offer protection through more casual use of redundancy, and techniques that limit the damage caused by any one error.

Systematic error detection and/or correction coding

The effect of randomly occurring transmission errors can be tackled by employing redundancy in the transmitted form of the message. Careful use of redundancy can reduce the significance of individual symbols, thus making the potential loss of any symbol less critical. Techniques for the systematic inclusion of redundancy to allow a certain level of error detection and/or correction are considered below.

Parity codes

Parity coding guarantees that all blocks of transmitted data fulfil some statistical criteria (typically an even number of 1s in a word of binary code) [*HAMMING 80; YOUNG*

94]. Checking for the same condition in the received data allows the detection of some transmission errors. Among methods of error detection and correction codes, parity is by far the most commonly used.

The simplest parity code ensures an odd or even number of 1s in a block of binary code, coding all blocks independently. This allows the detection of any *odd* number of 'bit errors' within each block, however the check is fooled by any even number of errors in a block. The efficiency of the protection this code offers can be traded against transmission efficiency by varying the size of the coded block. Parity coding is simple to implement either by limiting the set of possible codewords to a subset that satisfy the parity condition, or more normally, the calculation and insertion of parity 'bits' just prior to transmission.

The simple parity check described above can be extended by involving each data bit in more than one parity check. The extension to involve each bit in two checks leads to what are known as rectangular and triangular parity codes (due to the conceptual way in which the data bits are arranged for checking). The inclusion of each data bit in two parity checks, offers the ability of detecting a higher level of errors (it requires four errors in a specific pattern of bit locations to completely fool a rectangular parity error detection check) and allows individual errors to be corrected as their position can be pinpointed.

Cyclic codes are a special subset of parity codes. They are intended to detect the presence of any error in a large amount of data. The most common implementation [WILLIAMS 93] is designed for relatively fast hardware execution. Unfortunately, the detection that an error has occurred in a large set of data is of limited use in the radio video link application. As its communication is one-way, it cannot request re-transmission, and as the error is not pinpointed no attempt can be made even to disguise it.

Hamming codes

Another approach to coding that permits error correction is the use of Hamming codes [HAMMING 80]. These are algebraic self-correcting codes, again aimed primarily at combating the effects of random errors. Use of hamming codes is simple to implement both in terms of coder and decoder, however, the penalties in transmission efficiency are high (use of a Hamming single-bit code to protect individual ASCII characters drops transmission efficiency by 36% [*HALSALL 88*]).

Convolution codes

Another method that employs redundancy to protect data against noise is the use of convolution codes. The use of long enough code symbols allows a convolver to be successfully used as a decoder even when much of the symbol has been corrupted by interference. The encoder transmits the same fixed long sequence for every '1' in the data set, and the logical opposite of that sequence for every '0'. Convolving this transmitted sequence at the receiver with the same long fixed sequence produces large positive outputs for the '1's and large negative outputs for the '0's. Noise added in the transmission channel that is sufficiently random in nature (i.e. uncorrelated to the data source) produces no net output from the correlator.

This type of coding is typically employed where there is no other option as the penalties in either transmission time or bandwidth are high. However, sufficiently long code symbols allow the recovery of data from signals that are significantly below the noise floor (such as transmissions from space probes). A second beneficial property of convolution codes is the implicit bit-timing that is built into the coded sequence.

Redundancy in the coded source data

As an alternative to the systematic use of redundancy, other techniques can be used to include it. These techniques are based on the 'over-description' of the message and although they are less efficient than systematic methods they are less expensive to implement.

Incomplete removal of redundancy

Any coding method that leaves some redundancy within the coded data, such that parts of the coded image have some independence from each other, offers each of these parts some protection from errors in the other. Such redundancy can be achieved through many different means - often dependent on the other coding methods used. Examples include: algebraic compression methods that limit the scope of their search for redundancy, schemes that compress images using the independent compression of sub-images, and those that compress the images in a sequence independently. Schemes with adaptive codebooks often periodically re-transmit entries in the codebook that have not changed (in case their previous transmission had been corrupted) - this is another example.

Like systematic inclusion of redundancy these more casual schemes incur cost penalties in transmission efficiency. A possible benefit, however, is that if the redundancy is kept from the original source data by making a compression coder less efficient that coder can often be implemented more economically.

Over-description

Sending more source data than is necessary to communicate the message is another form of redundancy that can offer a tolerance to errors. In terms of image communication this can relate to the use of higher resolution or frame rate than is absolutely necessary for the application. When errors are encountered this approach relies on the recipient of the data being able to distinguish between the signal and the noise.

If over-descriptive source data is already available then it can be obtained without cost. The coding and transmission costs, however, increase linearly with the amount of redundancy as the redundant data must be compressed without reference to the 'minimal' data or the redundancy is lost. An advantage of such a scheme is that if the error rate during transmission is low then the communicated image will be more detailed. This is especially attractive in a system where the detail of the 'minimal' image specification is low.

Techniques of damage limitation

The final class of error protection techniques considered here is concerned with minimising the effect of transmission errors, aimed both at the effects of errors in single transmitted symbols or the situation where several neighbouring symbols are lost (*burst errors*).

Transmission of frequency domain data

In terms of the effects of errors on reconstructed images, transmission of image data in the frequency domain is often considered more error tolerant than in the spatial domain [PRATT 69]. The reason for this is that the effect of each individual transmission error is not concentrated on one small area. Rather it is spread over a large number of pixels, each pixel being formed from a weighted sum of many frequency domain terms. This type of error is generally considered less objectionable.

When using compression techniques that employ transformations to the frequency domain this form of damage limitation comes as a handy bi-product of the compression process. To include the transformations (two transformations are required, the second to transform back to the spatial domain at the receiving end) purely for the reasons of damage limitation is expensive in terms of hardware. In the case of the radio video link under consideration, this cost is prohibitive.

Use of fixed-length codewords

Any code that uses variable-length codewords to describe a data set leaves the data vulnerable to transmission errors if that code does not also allow for their detection and correction. The problem with variable length codes is that the codeword boundaries (implicit with fixed length symbols) must be inferred from the data during reception. If a transmission error causes misinterpretation of received data such that a codeword is mistaken for one of a different length, all codeword boundaries are then lost until the system is actively re-synchronised.

Unless error protection techniques are employed in the video link the use of coding techniques that result in the transmission of variable length codewords should be avoided.

Re-arrangement of spatial data

Another technique with the aim of spreading the effects of errors, is the re-arrangement of spatial image data prior to its transmission. This technique can spread the effect of burst errors amongst pixels in a wide area rather than amongst those in a close group. This prevents any one area of the image from being completely destroyed. This technique can be extended into the time domain by mixing the data from several frames before transmission (e.g. MPEG, see [*ARAVIND et al 93*]).

The costs of these techniques are increased memory requirements and a short time delay in transmission while data is buffered (increased latency). It is unlikely that they can offer sufficient benefit for the cost in memory to be worthwhile in the case of the radio video link.

Summary

Coding techniques exclusively for error protection are generally expensive in terms of transmission efficiency. The few that are not are instead expensive in terms of the computation and memory required.

For the video link, this means that image compression and coding to protect the data from errors can only be considered in isolation *if* the compression coding can save enough bandwidth that the system can afford the loss in efficiency caused by a separate error protection coder. Otherwise the two aspects will have to be dealt with together - achieving a balance in the one coder between the transmission efficiency and the susceptibility of the code it produces to corruption.

Review of image compression techniques

This section reviews current image compression techniques, with a view to their use in the radio video link application. The techniques are organised into groups according to the primary method they use to achieve compression. As, in practice, many commonly implemented compression algorithms are hybrids that employ more than one coding technique, these groups may seem a little artificial. However, the segregation serves to group the individual techniques into categories that have similar implications in terms of the radio link application.

There are two fundamental approaches that yield compression: code data more efficiently, or throw some of it away. All compression techniques are based on these two approaches, although many different coding and reduction algorithms are employed. There are three techniques of efficient coding that are currently prevalent. These are entropy coding, predictive coding and transform coding. The review is split into four groups: techniques that use these three types of coding and a section on schemes that discard data.

Entropy coding techniques

By examining the statistics of a data set it is possible to devise efficient coding schemes particular to it. These techniques are based on the *entropy*[2] of the data set or data source as defined by Shannon [*SHANNON & WEAVER 63*] and are not particular to image data.

Huffman and Shannon-Fano codes

The basis for all entropy codes is that symbols in a data set generally occur with unequal frequency. In Huffman [*HUFFMAN 52*] and Shannon-Fano coding [*HAMMING 80*] short output symbols are assigned to frequently occurring input symbols and long output symbols to rare input symbols. If the frequencies with which the input symbols occur are sufficiently unequal then translation of the data to such a variable length code leads to a more efficient description.

The process of coding involves three steps: ranking of input symbol probabilities, assigning output symbols to the input symbols (devising the code) then actually encoding the data. In Huffman coding input symbols are organised into a binary tree structure based on their probabilities. Output symbols are then allocated

---

[2] *Entropy* is the quantity used in information theory as a measure of information content (cf. entropy in thermodynamics).

according to the positions of the input symbols on the tree. Shannon-Fano coding is identical except in the way that the symbols are allocated.

Huffman encoding always produces an optimal solution to the problem of mapping the input symbols to the possible output symbols (the way symbols are allocated in Shannon-Fano coding means that it is often slightly sub-optimal) [*GAILLY 95*]. For the coded output message to be truly optimal, however, the product of all output symbol probabilities and their length should be equal. The discrete steps in binary output symbol lengths mean that this is rarely the case.

Use of a code with variable length symbols incurs a cost which is often overlooked For variable length symbols to be interpreted without ambiguity, all symbols must be unique in that the start of a each must not be able to be mistaken for the whole of a shorter code (i.e. if a single '1' is defined as the smallest code symbol, all other symbols must start with '0', precluding a single '0' from itself being a symbol (unless only two symbols are required)). Fixed length codes, or *block codes*, do not suffer from this problem as their symbol boundaries are implicit.

An overhead of all adaptive coding systems is the communication to the decoder of the code itself. At low message lengths this can become a significant proportion of the entire transmission. The process of communicating Huffman codebooks is often reduced by transmitting the output symbol lengths alone. Given assumptions about the way the coder created the code, the decoder can then rebuild the entire set of output symbols.

Ziv-Lempel coding

Another approach to lossless coding was proposed by Ziv and Lempel and involves a process of building up a dictionary of frequent symbol strings such that the input symbol sequence can be described by reference to part of the dictionary wherever possible. The best known implementation is probably that devised by Welsh, known as LZW [*WELSH 84*].

Unlike Huffman where the encoding process is hugely recursive, Ziv-Lempel algorithms execute the coding process whilst analysing the symbol and inter-symbol probabilities of the input sequence. As such, the memory requirements are much lower, and importantly this is achieved without significant cost in compression efficiency [*ZIV & LEMPEL 77*]. In addition, through management of the dictionary size, the scheme lends itself to balancing efficiency against resources.

Arithmetic coding

The restriction on efficiency of discrete output symbol length suffered by Huffman and Ziv-Lempel coders is overcome in 'arithmetic coding'. Arithmetic coding tackles the problem of representing an entire input set of symbols using an interval of real numbers between 0 and 1 [*GONZALEZ & WOODS 92; ARAVIND et al 93*]. As additional input symbols are encoded the size of the interval is reduced according to the probability of that symbol occurring. Arithmetic coders can be implemented using assumed fixed sets of input symbol probabilities or more generally by analysing symbol probabilities prior to encoding or adapting them during the encoding process itself.

Suitability of entropy coding

Entropy coding techniques typically offer image compression at ratios of around 2:1, although, as they are lossless, the ratio is heavily dependent on the image content. The costs involved, however, are high. Gathering and sorting of the statistics is expensive in terms both of computation and memory, and when image sizes are small, the costs of any codebook communication should not be overlooked. A further problem of all these codes is their use of variable -length codewords. As mentioned earlier such coded data is particularly vulnerable to transmission errors - the corruption of a single bit of a codeword can lose the synchronisation of all of the following codeword boundaries.

Predictive coding techniques

Predictive coding is an extension of the general statistical approach taken in entropy coding that can be used profitably when coding the data of a Markov process[3] (such as natural image data) [*JAIN 81*]. Predictive codes achieve compression by exploiting likely structure within the data set. If that structure can be accurately predicted, the level of uncertainty in the source data is effectively lower.

Whereas statistical methods analyse the data set itself, predictive coding relies on statistical characteristics of the source itself. Such an approach can be successful if the information source is stochastic (its symbols occur according to probabilities). Thus the coding scheme can be based on the probabilities of symbol occurrence from the source rather than the actual symbol frequencies in each particular data set. This approach is closer to that described by Shannon and leads to encoders that are specific to the data source rather than any data set from it.

Predictive coders do not require the expense of gathering probability statistics on-the-fly, but their successful use is restricted to data that display the characteristic features of the source assumed during the system design. In image compression, predictive compression is normally designed to exploit redundancy between neighbouring pixels.

Run-length encoding

Run-length encoding (RLE) could be considered one of the simplest predictive encoders [*HAMMING 80*]. It is used to encode series of data, making the assumption that the next data symbol will be the same as the last. When it is not, the system encodes how long the last assumption remained true, and which symbol came next. Run-length coding can successfully be used in image processing, especially when communicating images as a series of bit-planes. It has very low implementation costs, but with natural image data it generally achieves a fairly low degree of compression (less than 2:1).

[3] *Markov Processes* are a subset of stochastic processes whose symbol probabilities are affected by previously chosen symbols.

Differential predictive coding

A more complicated but generally more successful approach with image data is taken in differential predictive coding [*JAIN 81*]. Here only the errors in the output of the predictor are transmitted to the receiver. By limiting the data used to make the predictions to that which has already been encoded, an identical predictor at the decoder can be used to reconstruct the original data set using that error data alone. This system is generally referred to as DPCM (differential pulse code modulation - c.f. PCM).

In image compression, interpolation-between and/or extrapolation-from the values of neighbouring pixels are generally used to make the predictions. Successful compression is achieved when predictions are accurate enough that the error data has lower dynamic range than the source pixel data and can thus be encoded using less bits per pixel. A fixed size of error term is normally used and if any errors are too large to be coded then the complete pixel is sent (preceded by an escape sequence to prevent the pixel from being misinterpreted as error data). The size of the error term is typically set to allow a compression ratio of the order of 2:1 if compression is successful.

Suitability of predictive techniques

Predictive coding is attractive in that it can be achieved using relatively low hardware costs. A small two-dimensional predictor can be implemented using a few additions, binary divisions and enough memory to store just over one line of pixels.

Problems can occur, however, with the use of predictive coding on image sequences (as in the video link application). Where prediction accuracy is not sufficiently high and the implementation is to be lossless, then either the communication system must be built to cope with the possible increase in data rate caused by the transmission of the escape codes, or frames must be dropped. In addition, the receiver must be able either to cope with a variable frame rate or to buffer data in order to re-display an old frame when a frame has been dropped. Predictive schemes could alternatively be implemented as a lossy compressor, guaranteeing a compression ratio of say 2:1. This approach would require careful management of the situations where prediction error data had to be quantised, both so that the visibility of the quantisation errors is minimised and that they do not propagate further into the image via the predictor.

Successful predictive coding achieves compression by reducing inter-pixel redundancy. Unfortunately, if compression has been successful, each pixel reconstruction then relies on data from many previous pixels' reconstructions. This open-loop nature makes the compressed data vulnerable to even very low error rates and therefore unsuitable for transmission via a realistic channel unless the transmission is preceded by some error protection coding.

A partial solution to this problem is to limit the dependence of predictions to within a local area of image data. This restricts the effect of any transmission errors to within the area of data in which they occur. The protection of data through such isolation comes at a cost in achievable compression ratio, however, as, when blocks are encoded independently, inter-area redundancies can no longer be removed.

Transform coding techniques

A radical alternative to standard coding techniques is to describe images (or sub-images) as transformations of other images. This approach, known as transform coding, uses transformations or *mappings* consisting of translations, rotations, scalings and 'warping functions' to describe the route from one pixel array to that desired. Although the transform of complete images has advantages in terms of distortion [*SEITZ & LANG 90*], images are often described using a number of combined smaller transformed arrays (or *sub-blocks*) to reduce the amount of computation (both during compression and decompression). Where source images are dynamically partitioned into sub-arrays, information about how to recombine the them spatially must also be transmitted.

Transform coding itself is not generally aimed at achieving compression. The strength of the alternative codes lies in their ability to present data in ways in which it can be simply quantised in order to approximate images efficiently. As such they are employed almost exclusively in lossy approaches.

The most prevalent transform coding technique employed is the discrete cosine transform (DCT) that transforms from spatial to frequency domains and vice versa. This transform is at the heart of many hybrid compression techniques such as the JPEG image compression standard. The wavelet transform [*PRESS 92*] is another that transforms data into an orthogonal space. It decomposes the source into a series of sinc-like functions, which when quantised and re-transformed are intended to lead to less objectionable errors than if the DCT had been used.

Compression methods that employ other forms of transform coding include: block transform coding which uses a codebook of general array primitives and affine transforms to encode images (that are first divided into 'sub-blocks'), motion compensation which uses parts of historic images as the array primitives (thus achieving compression by re-using the old data) and fractal compression which uses contractive transforms which when applied iteratively will tend towards a stable image [*JACQUIN 92*]. Fractal compression is particularly attractive due to its independence from resolution and aspect ratio and the fast decoding speeds it offers. Although fractal techniques can offer compression ratios in excess of 100:1 the coding process is highly non-linear and very computationally expensive and, as such, cannot be performed in real-time [*FOX 94*].

Suitability of transform coding

As mentioned previously, data transformed into orthogonal spaces that lend themselves to the inconspicuous use of quantisation tends also to be well protected from the effects of random transmission errors especially when used with a low compression ratio [*PRATT 69*]. This favours the use of such transforms when transmitting the compressed data. The use of block-based and motion compensation transforms can make data more prone to corruption especially when used at high compression ratios. Fractal coding suffers similarly, however its iterative nature has the advantage that the effects of transmission errors die away over time [*HURD 92*].

Unfortunately all transform coding techniques are computational intensive. For example, the 2-dimensional DCT transformation of an 8x8 pixel block requires 1024 multiplication and addition operations [*YATES & IVEY 95*]. In addition to the large amount of computation, most require buffering of the input image and significant memory during execution. This complexity means that they cannot be employed whilst satisfying the hardware minimal requirement of the radio video link..

Sub-sampling techniques

Probably the least elegant technical approach to lossy image compression is the discarding of some of the source data to obtain a smaller but cruder representation of the message.

Discarding data can be performed mathematically by quantising or truncating components of the data set. Quantisation tends to be more common than truncation, as it can be implemented whilst preserving the dynamic range of the original data set, however both techniques are used. In image compression, sub-sampling can theoretically be performed on spatial, colour, temporal and frequency components of the data, however output spatial resolution and frame rate are often fixed (as is the case in this application) leaving only the components of colour and frequency as candidates.

Quantising can be performed blindly on the data, or with some consideration of its effect on the perception of the image. The ideal approach would be to remove the data that has least importance in conveying the image to human observers. Design of such techniques can be made on an *ad hoc* basis and their evaluation based purely on subjective viewing of images. A more scientific approach to both the design and evaluation of all lossy image compression techniques can be taken with some understanding of the human visual system.

Frequency sub-sampling

A common form of image compression is to transform image into the frequency domain then either quantise data coefficients or simply discard small ones. Although computation costs can be reduced by processing images as a series of smaller sub-images, the transforms involved (such as the DCT that was already ruled out above) are too expensive for use in the radio video link.

Colour quantisation

Colour quantisation is a form of compression that is used in even the most basic of digital image formats. Indeed it is used in all digital image formats other than so called 'true colour'. This proliferation is for two reasons: the enormous amount of redundancy in the colour information of digital images (especially simple graphics) and the compatibility of displaying data that is compressed in colour format using the hardware architecture of all but the most expensive digital image display systems[4].

Opinions differ as to the ultimate colour resolution of the human visual system [*STOFFEL & MORELAND 81; BLINN 92*], however, it is generally considered that under normal conditions a grey scale resolution of 6 bpp or 7 bpp (i.e. 64 or 128 grey levels) is sufficient to represent a smooth grey colour space on a cathode-ray tube (CRT) display [*RUSS 95*]. The standard greyscale resolution used in greyscale image storage and manipulation is 8 bpp. This is probably due to its convenience when using computer platforms that operate with 8, 16 and 32-bit words. In most applications there is therefore potential for at least a reduction in colour resolution by an eighth without a perceivable loss in image quality.

With quantisation, errors in the absolute value of individual pixels will always occur. The most obvious and objectionable problem occurs, however, when there are too few quantisation levels in an area of the colour space where a smooth gradient in the source image needs to be represented. The result of this is that discrete steps between the areas of different palette colours become visible, these are often misinterpreted as contours in the image, this is the problem of *false contouring.*

The use of non-uniform quantisation steps and adaptive quantisation steps that attempt to optimise themselves for the current image data can serve to alleviate this problem, but there is always potential for it to occur if the number of quantisation steps is severely limited.

[4] This architecture is based around the use of a hardware 'frame buffer' to store complete display images. Instead of storing the colour of each pixel to the resolution of the ouput device, frame buffer size is reduced by storing references to a subset of the display device gamut in the form of a look-up table (or *palette*) [*HECKBERT 82*].

An alternative method of tackling the problem of false contouring is the use of dithering (or halftoning) techniques. These techniques simulate the appearance of the missing intermediate shades by dithering the available shades and exploiting spatial integrating properties of the human visual system.

The drawbacks of halftoning techniques come both in their implementation costs and in the high frequency noise they add. (The satisfaction of strict viewing conditions can alleviate the patterning problems of the high frequency noise - indeed a high enough resolution can make it imperceptible.)

One particular dithering technique stands out as being suitable for use in this application. This is both because of the ease of its hardware implementation in and its effectiveness at solving the false contouring problem. This technique is Error Diffusion, and was first presented by Floyd & Steinberg in 1975 [*FLOYD & STEINBERG 75*]. It is arguably the simplest of the many dithering techniques that exist and, amongst those which are adaptive, it is the most popular [*ULICHNEY 88*].

Vector quantisation

An alternative to quantisation of the scalar value of individual pixels is the quantisation of groups of pixels - a process referred to as vector quantisation (VQ) [*NASRABADI & KING 88; COSMAN 93*]. Images are broken down in sets of sub-images (*input vectors*) which are then represented by the closest *reproduction vectors* from a code book. Decoding vector quantised images is relatively simple as it is a lookup operation; regenerating the image from its coded form and the original codebook. Coding is more computationally intensive as it involves the task of searching through the codebook for the best fit vector. This task is often reduced through the use of incomplete searches. A common implementation of this is the use of carefully arranged codebooks that use a tree-like structure to organise similar vectors.

Conclusions

Three aspects of coding are relevant to the radio video link: compression, vulnerability to errors and coding for ease of reception.

Coding methods dedicated to error protection are expensive in terms of either implementation cost or transmission efficiency. Their implementation in the radio video link application conflicts with the aim of producing a sensor-transmitter interface with a minimum of hardware.

A compression technique was sought which could satisfy the following criteria. It had to achieve the desired compression ratio (c. 2:1), require only minimal hardware and yet not produce compressed code so vulnerable to errors that a dedicated error protection stage is required within the coder.

Colour space quantisation with error diffusion satisfies these requirements. The use of error diffusion in image compression is novel, as it was originally developed to allow the display of continuous tone images on a binary output device.

chapter three

Algorithm Design and Evaluation

Chapter two has shown that colour quantisation with error diffusion is the data coding approach most suited to the radio video link application. This chapter explores the hardware implementation of algorithms based on this approach. After examining the techniques involved, the general hardware architectures that are required to implement algorithms are considered. Implementations of the algorithms are then tailored in an attempt to reach an optimum balance between compression and the costs in image quality, hardware complexity and increased vulnerability of the data to transmission errors.

The evaluation of proposed hardware implementations is achieved through a combination of objective and subjective evaluations of software simulations. A structured simulation environment has been developed to facilitate software implementation of the hardware-based algorithms. Within the software environment, image processing can be performed on sample images using the software algorithms. Processed image display and the output of some simple metrics allow a degree of both subjective and objective evaluation.

In order to put the work into context some background information about the history of halftoning is presented (with particular reference to error diffusion). General factors of algorithms that are likely to affect implementation costs are emphasised and the software simulation environment introduced before details of particular algorithms are considered. Conclusions are drawn as to which algorithms are most suitable for use in the coder of the video link.

The coding approach

Colour space quantisation with error diffusion is deemed suitable for the radio video link as it offers modest compression with minimal hardware. This is achievable through two key factors: the data can be processed in an order close to the raster scan order of the input (thus minimising buffering) and can be executed using relatively simple computation. Unfortunately, quantisation with error diffusion suffers from the visibility of the error diffusion patterning. There is a trade off between the complexity of the diffusion scheme and the sophistication (thus visibility) of the masking.

The problems of using colour space quantisation alone are discussed below. The use of spatial dithering is then introduced as a solution to these problems. The motivation behind selecting error diffusion (one of many dithering techniques) for use in the radio video link is then explained. A brief history of the development of error diffusion algorithms is given and the details of error diffusion processing introduced.

Colour space quantisation with error diffusion

Colour space quantisation is the use of a cruder description for the colour of each image pixel. Although the technique can be applied to any image data, only greyscale data is considered here as this is the colour space of the video transmitter application.

Simple independent quantisation of the grey level of each pixel in an image yields compression. By treating pixels independently, it exploits few of the redundancies that can be found in image data. There is, however, one key advantage in this lack of sophistication in that after compression, pixels remain independent in their description. This limits the extent of damage caused by errors in communication of the image data.

The disadvantages of quantisation become apparent when the colour space resolution is so low that it is possible to distinguish between adjacent shades. This can lead to some image detail being lost and other detail being amplified. This problem is generally referred to as 'false contouring'. Areas of image which have slow, smooth changes in grey level (before quantisation) end up being represented by a series of flat regions that meet at obvious steps - the 'contours'.

False contouring is exhibited in Figure 3.1 below. At the lower resolutions edges or 'contours' appear in areas of nearly flat grey where there is insufficient greyscale resolution to represent smoothly transitions between perceptible shades.

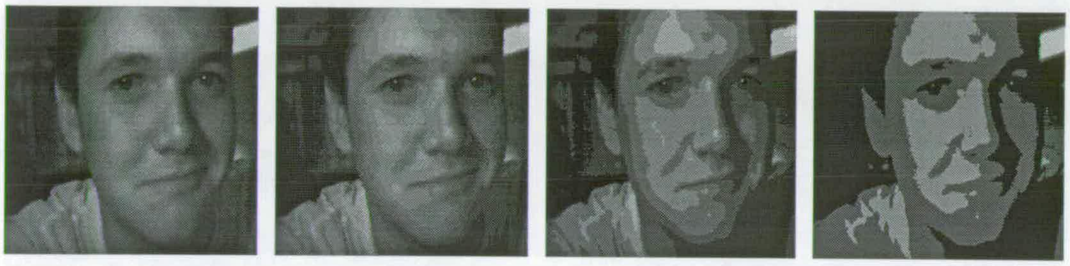


Figure 3.1 ***Examples of false contouring*** *- the greyscale resolution of the images decreases from left to right (6, 4, 3 & 2 bpp), increasingly obvious contouring is exhibited as a result.*

The perception of false contouring depends on a vast range of factors, including the vision of the observer, the distance between the observer and the display, the spatial resolution, contrast and size of the display, and the illumination of the display surroundings. In addition, the visibility of contouring can be masked by large amounts of detail in the image and the addition of a small amount of noise prior to quantisation [*GOODALL 51*],[*ROBERTS 62*].

There is therefore no single greyscale resolution at which contouring becomes apparent to all observers under all viewing conditions. Attempts have been made to determine the limits at which contouring *cannot* be distinguished. Research has shown that some greyscale images require sampling at more than 256 levels of grey if contouring is not to occur [*STOFFEL & MORELAND 81*]. However, as stated earlier in chapter two, it is generally accepted that, under most circumstances, human observers can only distinguish between around 2^6 or 2^7 shades of grey when viewed on a CRT [*RUSS 95*]. This maximum degree of greyscale resolution indicates that sampling, storing and display of greyscale pixel values using 7 bits is adequate to meet the needs of the human visual system. At this resolution it should not be possible for false contouring to occur as transitions between neighbouring greys should be imperceptible.

Achieving the compression ratio required to marry the minimum image specification with the capabilities of the cheaper radio links outlined in chapter one through quantisation alone requires quantisation to a maximum of 4 bits/pixel (bpp). This is unsatisfactory as false contouring is evident at this resolution under most viewing conditions. Error diffusion, a form of spatial dithering, is a technique that can be used to alleviate the problems of low resolution quantisation.

Spatial dithering background

The display of continuous tone images using insufficient colour resolution is not an uncommon problem. The communication of most continuous tone imagery has to overcome this barrier[1]. Consequently, the subject has been explored by numerous parties interested in image display including artists, illustrators and printers and has recently received attention from the image processing research world [*ULICHNEY 87*]. The solution is to use spatial patterns of the available shades to create the illusion of missing intermediate shades. Artists create these patterns manually, often combining clues about the texture of the subject into the shading patterns. Two examples of these manual techniques are shown in Figure 3.2.

In order to automate the creation of such patterns processing techniques (known as spatial dithering or halftoning algorithms) have been developed. Examples of the output of some of these techniques are shown later in Figure 3.3.

[1] Two exceptions are the use of photographs and dye sublimation printing.

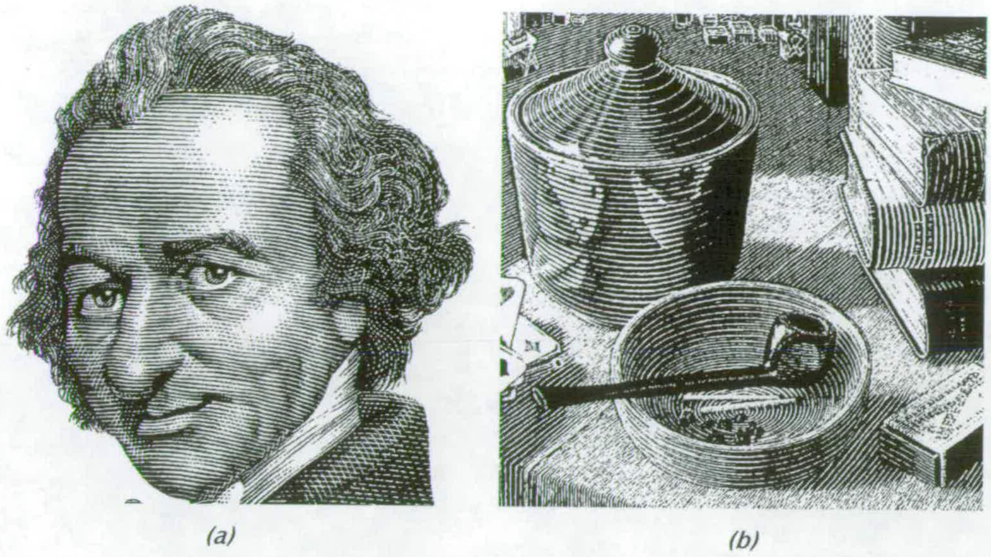


Figure 3.2 ***Examples of manual shading****, (a) an illustration of Thomas Paine [SOMMERS 95], and (b) a section of 'Still Life with a Street' [ESCHER 37].*

The success of these shading and dithering techniques relies on the spatial integrating properties of the human visual system. There are two main contributing factors. Firstly, the spatial resolution of the human eye is finite, if this resolution is sufficiently exceeded by the dither pattern[2] then results indistinguishable from a continuous tone image can be produced. Secondly, even when the resolution is low enough that individual pixels of the dither pattern can be resolved, features of the human visual system mean that a sensation of smooth colour can still be perceived [*MULLIGAN 93; CHAU 90*].

Particular dithering applications

Most dithering techniques are aimed at rendering continuous tone greyscale images using bi-level output devices (e.g. printing with black ink on white paper). However, the concept of dithering to produce intermediate shades is equally applicable to any situation where the colour resolution of the display device is less than that of the image to be displayed. As such, the concept of dithering, and many of its techniques, have been applied in other situations including the printing of full

[2] Based on the physiology of the human eye, there is a limit to the spatial resolution of the human visual system. This limit has been estimated to be at approximately 128 cycles per degree subtended [*SAKRISON 77*], although by 50 cycles per degree the response is almost zero [*CHAU 90*].

colour images using cyan, yellow, magenta and black ink (known as process colour) and attempts to display continuous tone imagery using modest colour palettes in computer displays [*HECKBERT 82*].

Use of dithering with colour space quantisation in the video link application is slightly unusual in that the output device itself is not the limiting factor. Rather, it is the intermediate form used to represent the image data that is limited in resolution (in order to lower its bandwidth). Yet the problem remains one of having insufficient colour space resolution to display continuous tone imagery.

Alternative dithering algorithms

A number of dithering algorithms have been developed. They differ both in the ways they can be implemented and in the style of pattern they produce. The techniques are generally classified according to two features: whether they operate on the data using fixed or adaptive techniques (fixed techniques correspond to ordered dithers), and whether the output pattern they produce is made up of dispersed or clustered dots. A brief introduction to the more common dithering algorithms is given below, more exhaustive surveys and detailed explanations can be found in [*ULICHNEY 87*] and [*JARVIS JUDICE & NINKE 76*].

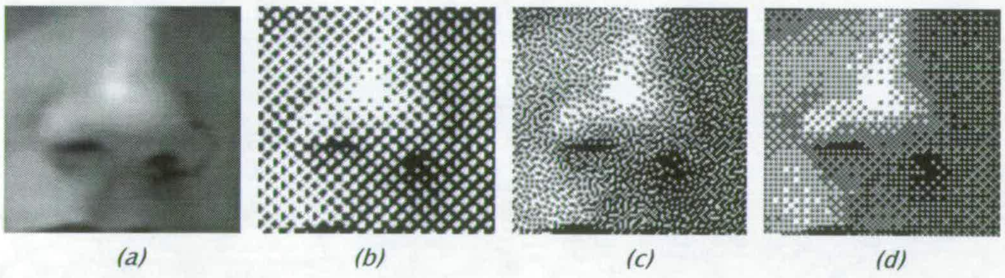


Figure 3.3 ***Examples of common dithering techniques:*** *The first image, (a), shows the 8 bpp original, the rest are sub-sampled and dithered versions that have been magnified to show the dither patterns clearly[3]. Image (b) was produced using a* ***clustered-dot ordered dither****, (c) an* ***irregular dispersed-dot dither*** *(Floyd-Steinberg error diffusion), and (d)* ***a dispersed-dot ordered dither****.*

[3] As this figure is intended to demonstrate dither patterns, a filter was used to blur the dither images. This is intended to compensate for the fact the images are shown at approx. 100 dpi but with the size of dither pattern used they would normally be printed at 600 dpi. Like the original, the blurred images have also been rendered by the rateriser of the printer driver (which uses a clustered-dot ordered dither with an output resolution of 600 dpi).

Figure 3.3 shows examples of the output of three of the most commonly used algorithms. Both (b) and (d) were produced by ordered dithering, a process also referred to as both *halftoning* and *screening*. All ordered dithers produce an inherently structured output. The pattern and scale of the structure is dependent on the size and pattern of the halftone cell used. Clustered-ordered dithering, (seen in Figure 3.3(b)), where the printed dots are clustered together, is the technique most commonly used in publishing. This prevalence is mainly due to features of the most commonly employed printing technologies (such as offset lithography and laser printing). The mechanics of the processes mean that printed dots below a certain size cannot be produced, however, above this fundamental size the size of each printed dot can normally be controlled with a high degree of precision. These capabilities lend themselves to the reproduction of clustered-dot output. The spatial resolution of the technologies (anywhere between 300 to 2400 dpi) means that at normal viewing distances the spatial frequency of the structured patterning produced by ordered dithers is sufficiently high not to be distracting.

Dispersed dithers, that produce output like the example shown in Figure 3.3(c) and (d), tend to be used with hard copy or display devices with low spatial resolutions, where pixels are non-overlapping and of fixed size (such as fax machines and the graphics displays used with computers). Irregular dithering techniques are designed to produce output with less obvious structure than ordered methods - often described as being able to simulate higher colour resolution without sacrificing spatial resolution [*CHEN 92*]. Although clustered irregular dithers have been developed [*KNUTH 87*], irregular dithers are much more commonly implemented as dispersed dithers as they tend to be employed where low spatial resolution allows dither patterning in the output to be resolved.

In addition to the different dither patterns that the various algorithms produce, differences in the way the image pixels need to be processed affect the architectures that can be used to implement the algorithms. The main factor that differentiates the possible implementations is whether an algorithm is ordered or irregular. In ordered dithers the processing of each pixel is essentially independent. This means that during execution of the algorithm any number of the pixels can be processed in parallel, thus allowing a trade off between the speed of processing and the amount of hardware employed. In contrast, irregular dithers are adaptive neighbourhood processes. This means that the behaviour at each pixel can be affected by the value of neighbouring pixels, thus making the algorithms inherently serial.

The factors that affect how suitable different forms of dithering are to a particular application therefore include the suitability of the type of dither pattern produced (both in terms of the capabilities of the output device, and the significance of the pattern to the observer) and also the complication of the processing and any differences in possible implementation architectures.

Error diffusion

Chapter two suggests that error diffusion (a form of irregular dispersed dithering) is the most suitable dithering technique to be used in the transmitter of the video link application in order to alleviate the problems of employing colour space quantisation.

The low spatial resolution of the video transmitter images and the use of a non-overlapping pixel graphics display both point to the adoption of a dispersed dither. Error diffusion was chosen as, amongst dithering algorithms, error diffusion is considered to offer good detail rendition [*STOFFEL & MORELAND 81*]. In most circumstances it produces dithering patterns with an attractively low-structure content [*ULICHNEY 88*]. It is not, however, without problems. The process causes an inherent spatial shift in image energy. Combined with the serial nature of common implementations this leads to slight movement of details such as edges. A second problem it shares in common with many neighbourhood dithers is that there is no lower limit to the frequency content of the patterning it produces [*STOFFEL & MORELAND 81*]. Despite these problems error diffusion is generally considered to give the best results at low spatial resolution and is the most popular neighbourhood dithering algorithm [*ULICHNEY 88*].

Although normally used with still images, the order of pixel processing in typical implementations of error diffusion is particularly suited to the raster scan order of video data. Applying the algorithm to image sequences introduces a temporal element to the dither patterning. Studies into the intentional use of dither patterns that change with time in the display of still images have shown promising results [*MULLIGAN 93*], however, it was not known exactly how the addition of a temporal element to an algorithm intended for spatial diffusion would affect the degree to which the dither patterning it produces would distract the observer. The appearance of high or low temporal frequency content in the patterning could either serve to mask it or make it more obvious.

History of error diffusion

Error diffusion was first proposed in 1975 as a method of rendering greyscale images using bi-level output devices [*FLOYD & STEINBERG 75*]. More generally, it can be regarded as a way of minimising the visible effects of any image data quantisation. The seminal work has also been attributed to Schroeder [*STOFFEL & MORELAND 81; SCHROEDER 69*].

Many alternative diffusion algorithms have been proposed since. Some have advocated the use of quite different diffusion schemes [*KNUTH 87*] and the use of alternative rasters [*WITTEN & NEAL 82; VELHO & GOMES 91*]. Alternative diffusion filters that trade off aspects of the diffusion patterning against each other and against computational expense have also been proposed (e.g. [*JARVIS JUDICE & NINKE 76*]).

Ulichney reported on the success of a perturbed version of Floyd & Steinberg's filter [*ULICHNEY 88*]. He describes the dither patterns produced as having the characteristics of 'blue noise'. Part of his work was later criticised by Bernard who suggested relaxing the 'dc constraint' normally imposed in error diffusion to improve the frequency content of the diffusion pattern [*BERNARD 92*]. An ordered dither with the same blue noise property has also been suggested [*MITSA 92*].

Error diffusion mechanics

Error diffusion is a process of error feedback that ensures that neighbouring quantisation errors tend to cancel. Individual quantisation errors are divided up and added to (or *diffused over*) a number of as-yet unquantised pixels, thus influencing future output in a way so as to cancel the current error. Errors are shared out amongst the as yet unquantised pixels according to a set of weights generally referred to as the *diffusion filter*. Error diffusion can be considered as a process that shifts quantisation noise from lower to higher spatial frequencies.

The importance of viewing conditions

As mentioned above, in common with all methods of dithering, the masking of the higher frequency noise produced by error diffusion and thus the 'illusion' of missing shades relies on assumptions about the frequency characteristics of the human visual system. The degree to which the illusion succeeds is dependent on viewing conditions. The most important of these are the perceived linearity of the display system and the perceived pixel size. Error energy is generally diffused assuming a linear data space, thus any non-linearity in the display mechanism affects the success of the error cancellation process. Perceived pixel size affects the visibility of the dither patterning, due to both the finite spatial resolution of the human visual system and the nature of its sensitivity to different spatial frequencies [*PEARSON 75*].

Summary

Error diffusion is suitable for use in the radio video link because, even at low spatial resolutions, it is successful at masking the problems of greyscale quantisation. The use of a halftoning algorithm for compression is novel as their normal application is in the rendering of still images for bi-level output. The ability of the video link to support the bandwidth of a modest greyscale, together with the fact that it transmits moving images, permits the use of less sophisticated diffusion algorithms.

Considerations for minimal hardware

The minimal hardware requirement of this project gives rise to two major areas of concern when evaluating the implementation of the algorithms. Firstly, the complexity of the computation and control logic required. And secondly, the amount of memory required both to implement the algorithm and to marry the resulting coder to the remainder of the system.

The complexity of the algorithm will obviously have a direct influence over the amount of computational hardware required. In particular the use of divisions and multiplications should be avoided (unless by powers of two). The exploitation of significant parallelism is unlikely to be of benefit unless processing data in parallel reduces the buffering needs of integrating the coder into the overall system.

The amount of memory that is required by the coder is dependent on two factors: any general data buffering required to manage the order and speed with which data is moved in and out of the coder and any requirements of the algorithm for temporary storage of partial results. The amount of general buffering at the input and output of the coder depends on how far the order of processing required by the coder differs from that of the raster sequence produced by the video sensor. Using a raster processing order for the diffusion processors would obviously minimise this requirement. The use of memory for partial results is dependent on the complexity of the algorithm. In terms of the error diffusion algorithms, the number of pixel cycles over which any partial result will need to be stored (and therefore the number that need to be stored over any one pixel cycle) will depend on the size of the diffusion filter.

A simple algorithm that processes data in a conventional raster scan order is therefore desirable.

Algorithm simulation environment

To meet the testing and evaluation needs of the design and evaluation process a software simulation environment has been developed. Within this environment coding schemes can be applied to live video input, standard test sequences or test still images. In addition to displaying both the source images and the results (for subjective comparison) a number of simple analytical tools have been included.

These tools both aid further subjective analysis of the results and offer some simple objective meters. The tools can be used to check that the implementation of the algorithm is bug free, that the algorithm performs as expected and to allow comparison of alternative algorithms. The tools comprise: differencing between source and processed images, squared differencing, differencing on blurred/softened images, calculation of grey level frequency histograms and calculation of mean grey level.

Live video input is achieved using a video sensor and a PCMCIA frame grabber. Test images can be read in from disk. To enable the direct comparison of algorithms, up to three types of processing and analysis can be performed on the same source with the results displayed simultaneously.

Design overview

To simulate the effect of the coding algorithms there are four main tasks required of the software:

- obtaining source images
- application of coding and analytical algorithms
- image display
- servicing user input

A flow chart of the main loop of the simulation software is shown in Figure 3.4. Whilst running, the software continually executes this loop.

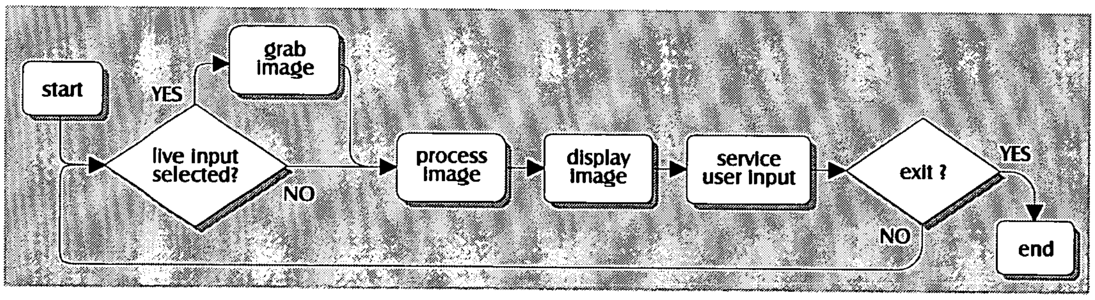


Figure 3.4 ***A flow chart representation of the main simulation loop***

The different image processing and display routines have been written and are called using a modular architecture that allows the simple inclusion of further processing functions and analytical tools as they are developed.

Image data is held in memory in a set of 2-dimensional arrays. This set of arrays (the frame store) is central to the flow of image data around the program. This flow is shown schematically in Figure 3.5. Access to the frame store is the only feature that the image capture, image processing and image display parts of the software share.

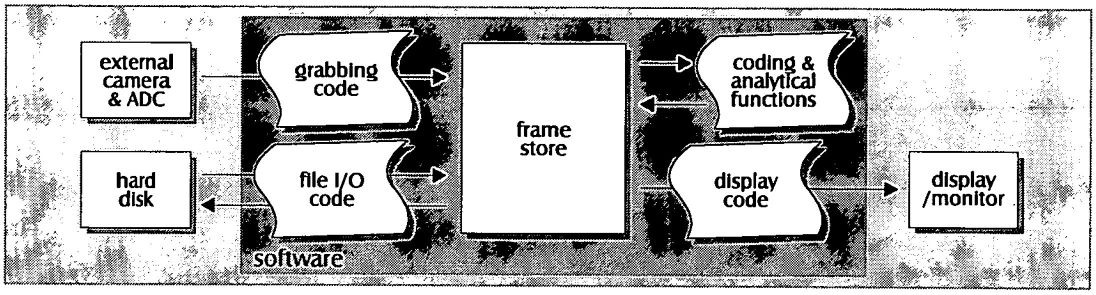


Figure 3.5 A schematic of the ***data flow within the simulation software.***

Source images are obtained either live from a video sensor (connected through a PC Card (PCMCIA) frame grabber) or loaded from file. All processing and analysis is then performed in the frame store and from there the images are transferred to display memory.

Up to three algorithms can run concurrently together with one analytical tool per algorithm. This triple replication explains the layout of the graphical display which is shown in Figure 3.6. Each of the vertical columns contains a processed output window, an analysis window and a status window (top to bottom), details of the currently selected image source and of file activity are given in the panel on the left.

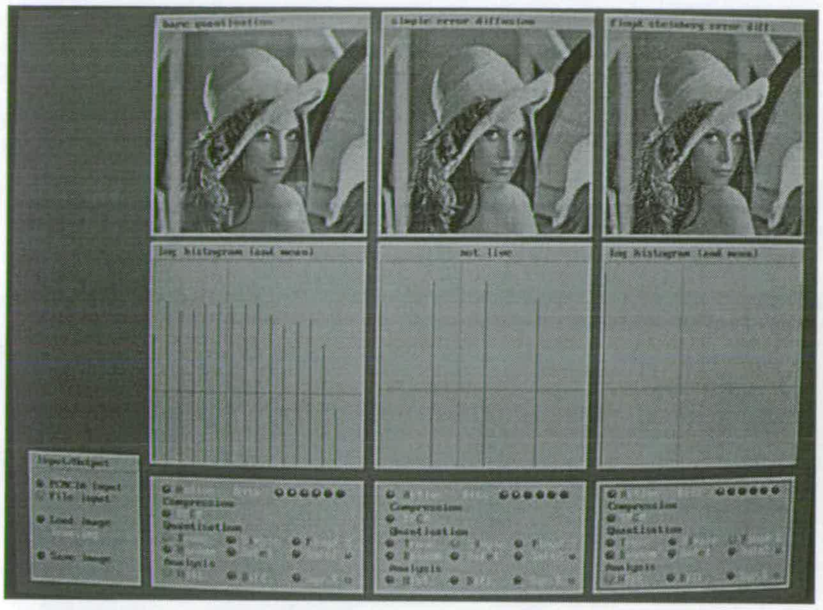


Figure 3.6 ***The DOS graphics display.***

Images can be saved to the hard disk either as binary files (the format used for input) or as ASCII files. Grey level frequency histograms can also be saved as ASCII files.

The coding functions

The various image coding algorithms are implemented in software as discrete functions. Each can be switched in and out as required. All of the coding functions access the image data using pointers that are moved through the image data in raster scan order, thus emulating the on-the-fly access that the hardware implementations would have. Limiting access in this way means that any buffering overheads of the algorithm are immediately obvious. On initiation, each coder function is passed pointers to the memory location of the first pixel of the source image and the memory location where the first pixel of the processed image should be written. Additional parameters (such as desired output pixel depth) are passed as required.

Although the information content of the image data is reduced during processing by the coding functions, the software stops short of arranging the data to produce truly compressed output. This step was seen as an unnecessary complication as the software was intended purely for evaluation of algorithms and not to be used for compression itself. Each coded frame occupies the same amount of memory as the original, however, the number of different grey levels always corresponds to the current resolution (in terms of bits per pixel).

Analytical functions

The analytical functions in the software include the ability to display the grey level histogram of a processed image, along with the image mean. This meter gives instant clarification of which grey levels are used in the output, confirming the output resolution, whether pre-scaling is being employed and with adaptive algorithms how the adaptation is changing. The display of the mean level is a convenient indicator of errors in algorithm implementations (as few of them should alter the image mean significantly). A log scale is used for the histogram display to cope with the wide dynamic range (0 to 65535 displayed using 256 steps). Another of the meters is the ability to generate a difference image (the pixel-by-pixel difference between source and processed images). This shows where the processing errors are most significant. Where errors are small enough, squared difference can also be employed. A smoothed difference image can be generated to display the difference between error diffusion output and the source without this being dominated by the pattern produced by the error diffusion pattern. This calculates the difference between versions of source and processed image that have been smoothed using a 5 by 5 smoothing filter.

Tailoring of the algorithm

The general architecture of quantisation with error diffusion is shown schematically in Figure 3.7. It consists of two functional blocks: the quantiser and the diffuser. Data enters the system at the diffuser, where past quantisation errors are added. It then passes out through the quantiser. A feedback path supplies the diffuser with the error from each quantisation.

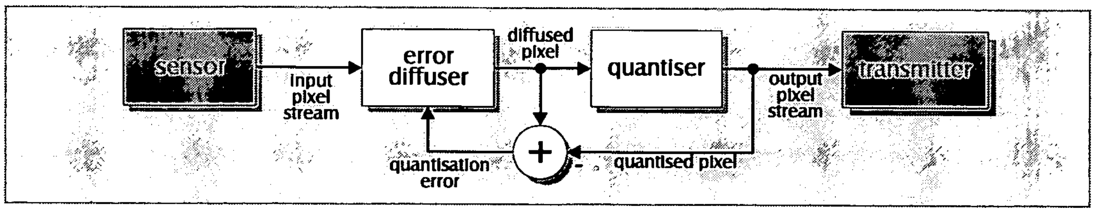


Figure 3.7 *A schematic of the* ***general quantiser with diffuser architecture****. The system blocks that would neighbour the diffusing quantiser are shown in grey.*

This general data flow is common to all implementations of error diffusion. The exact behaviour of the diffuser and quantiser are, however, dependent on details particular to the implementation. These can be altered to trade off issues of complexity, cost and performance.

As well as proving the overall validity of error diffusion as an approach, this section considers decisions about design flexibility. The flexibility issues can be broken into two: those concerning the quantiser and those relating to the filter used in the error diffuser. These are discussed in that order below.

Quantiser design

The job of a quantiser is to approximate an input signal, producing an output that describes the input more crudely but can either be represented more efficiently, or in the case of analogue-to-digital conversion, more robustly. Whether quantising to convert a signal from analogue to digital or simply to reduce the number of bits used to represent a digital one, the process is executed by comparing the input to a number of threshold levels and assigning it an output value depending on the results of these comparisons.

In the error diffusion system the quantiser is required to reduce the precision used to describe the greyscale of each pixel. The input to the quantiser is therefore a digital signal. Alternative quantiser implementations differ in terms of simple factors such as their speed of operation, their complexity, and also in more subtle features of operation such as preserving the dc component of the input signal. Before going on to consider their relevance to the error diffusion system, the merits of alternative implementations are introduced and discussed below.

Implementation options

Hardware architectures used for analogue-to-digital converters can also be used to implement digital quantisers. The most applicable of these architectures are *flash* and *single-slope* (for analogue-to-digital versions see [HOROWITZ & HILL 90]). Schematics of the computational hardware required to implement these architectures are shown in Figure 3.8.

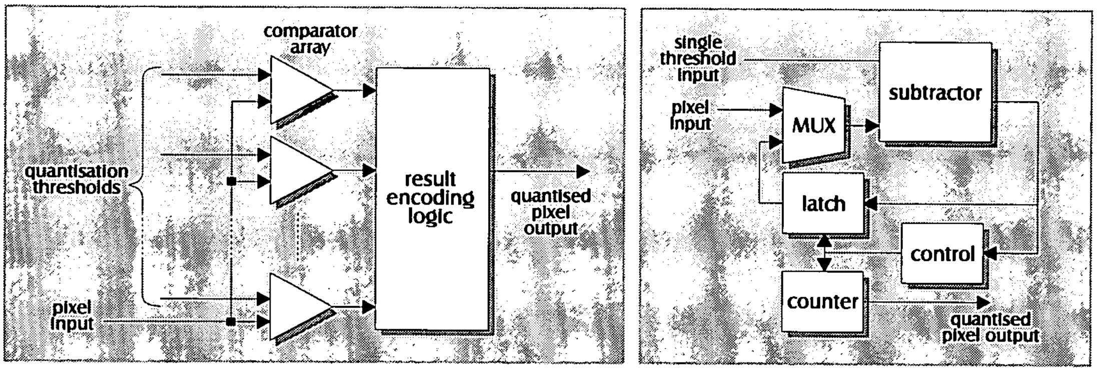


Figure 3.8 Schematic implementations of quantisers using architectures classically associated with analogue to digital conversion; (a) flash, and (b) single-slope.

The single-slope quantiser requires a full subtracter, the flash simply requires a set of comparators (which can be implemented using combinatorial logic equivalent to that used to generate the MSB of a hardware subtracter) and some simple combinatorial logic to combine the resulting comparisons.

Many of the pros and cons of these implementations stay true to their analogue counterparts. The flash architecture is expensive in terms of computational hardware (a quantiser with an output resolution of n bits requires (2^n-1) comparators), however the processing is simple to control and fast. It consists of one set of parallel comparisons followed by combination of the results. The single-slope converter uses less hardware (only one subtracter), however, the control logic is more complicated and, as the processing comprises a succession of subtraction operations, the conversion time is at best proportional to the output resolution.

In addition to these general architectures it is possible to achieve quantisation of a digital signal by far more hardware-minimal means. This approach uses a simple binary truncation operation where the output is simply the input signal after the least significant bits have been discarded. This operation can be realised without any hardware and thus incurs no computational delay.

The limitation of using binary truncation is that there can be no flexibility in the position of the quantisation thresholds. They are fixed and must be at regular binary intervals. In contrast, the thresholds of the flash converter are arbitrary and can be changed on-the-fly. The single-slope architecture can be programmed to use different steps for each subtraction (e.g. by presenting different thresholds to the subtracter using a multiplexer).

There are two key benefits of having flexibility in threshold position. The first is that the use of unevenly spaced thresholds can be used to combine the process of quantisation with the application of a scaling function (such as gamma correction) at no extra cost. The second is the use of a technique known as adaptive quantisation. This is only possible if the thresholds are programmable. A brief introduction to adaptive quantisation is given below.

Adaptive quantisation

Also referred to as 'tailored quantisation', adaptive quantisation is a variation on standard quantisation where the position of the quantisation thresholds are varied on the basis of some statistics of the current data set. Threshold positions are altered such that some error metric is reduced. Its application in greyscale quantisation generally centres around moving thresholds so that they are more closely spaced in the most densely used parts of the colour space, so reducing a metric such as MSE. The technique is particularly successful when the population density of the greyscale is far from flat. Quantisation of such a data set using fixed levels could result in many output 'bins' not being used and others containing disproportionately high numbers of image pixels. Examples of the quantisation of such an image (together with greyscale frequency histograms of the resulting images) are shown using both uniform and adaptive thresholds in Figure 3.9.

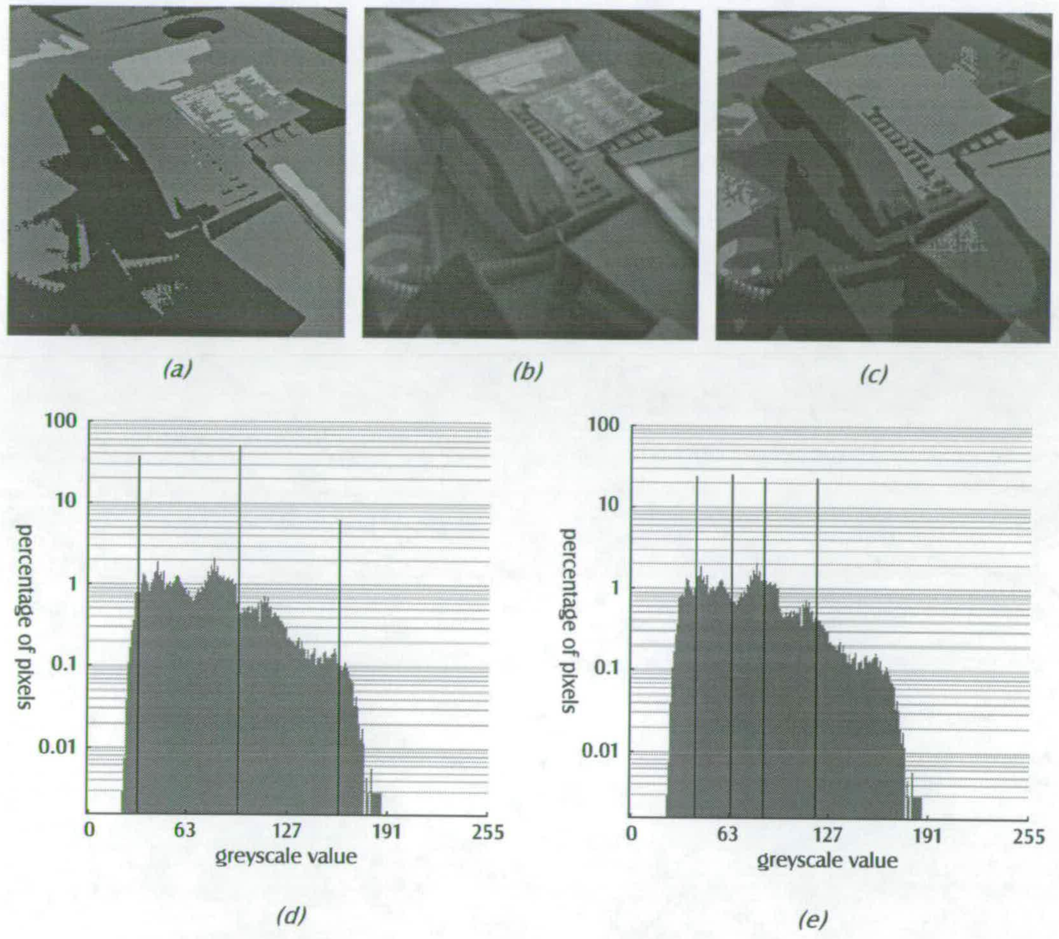


Figure 3.9 ***A comparison of uniform and adaptive quantisation****; the 8 bpp original image, (b), is shown in the centre of two versions that have been quantised to 2 bpp, the image to the left, (a), was created using uniform quantisation, the image on the right, (c), using an adaptive quantisation scheme. Below in (d) and (e) the sparse greyscale frequency histograms of the quantised images are shown. These are both superimposed on the much more detailed histogram of the original image (b). The histogram of (a) is shown in (d), and that of (c) in (e).*

A form of adaptive quantisation is implemented as part of the software simulation, the optimised results in Figure 3.9 come from this work. The algorithm used is essentially a one-dimensional implementation of Heckbert's median-cut algorithm [HECKBERT 82]. This represents only one example of optimisation, however, it shares the same three processing steps required of any algorithm: gathering statistics, calculating thresholds that, based on the statistics, minimise some error metric, then calculating representative output colours to be used when decoding the output. In the implemented algorithm this translates to gathering a full grey-level histogram, then, by making one pass through the histogram, defining threshold levels wherever the number of pixels between the last defined threshold and the current position within the histogram reach the average number of pixels per output code.

The representative output colour for each code is then calculated as the mean of all pixels represented by that code.

Relevance to the error diffuser in a video transmitter

In order to determine whether, in the context of the video transmitter application, the added functionality offered by the more complicated quantiser architectures justifies their hardware expense, it is necessary to consider the relevance of either applying a scaling function or using adaptive quantisation together with error diffusion.

The most likely scaling function that would be applied to the image data prior to quantisation is to correct for non-linearity. This is directly relevant to error diffusion, as its success relies on the perceived[4] linearity of the output data space (in order that the linear diffusion of pixel energy creates combinations of pixels that approximate the intended intermediate shade accurately). Non-linearities both in the sensor or display system can affect this. The ability to apply a non-linear scaling function prior to quantisation could be used to apply a combination of functions to correct for both display system non-linearity (often referred to as *gamma correction*) and to cancel non-linearities in the sensor or digitisation system. Two factors, however, make the benefits of such correction either expensive or doubtful.

The first is the near-linearity of the proposed image sensor and ADC combination. Without calibration of individual units, any function to correct this distortion would have little effect. Calibration would itself be an expensive step during production. It would mean that the correction function would have to be programmable, making it more expensive to implement.

The second factor is the dependence of perceived display linearity on many factors other than the linearity of the display itself[5]. Successful application of the correct

[4] It is important that it is the *perceived* linearity of the display luminance that is considered as human perception of brightness is non-linear - Weber's Law [*PEARSON 75*]

[5] Factors such as ambient light level and the shade of the surround of a display affect perceived linearity [*PEARSON 75*].

gamma function to make the display appear linear would therefore necessitate the careful control over viewing conditions (restricting the application). In addition to the practicalities of correcting for non-linearity, such a level of sophistication does not equate with the level of image fidelity implied by the remainder of the system (c.f. the image specification).

A stumbling block to the use of adaptive quantisation in the video transmitter application is its expense in terms both of computation and memory. Even with the simplest of error metrics, threshold optimisation is itself both non-trivial and requires a significant amount of memory. In the median-cut example compilation of the greylevel histogram for the 64 pixel square, 6 bpp minimum image specification image alone requires 256 Kbytes. At a cost in the precision of the optimisation, this memory requirement could be reduced by relaxing the accuracy of the statistics. However, even crude reduction both in sampling frequency (to say one in every 4 pixels) and grey-level resolution (to consider only the top 4 bits) still requires a significant amount of memory (16 Kbytes). A further overhead is that the palette of representative colours used to decode the quantiser output must be communicated to the receiver if the image data is to be interpreted correctly.

Iteration is one way to minimise the computation complexity of adaptive quantisation for use in application such as the video transmitter application. Instead of calculating an optimum set of thresholds for each frame, a small number of adjustments would be made per iteration, say the movement of one threshold every frame. This vastly reduces the size of the optimisation calculation per frame. An option for reducing memory costs is offered by simple statistical metrics (such as Heckberts) which permit gathering of statistics after quantisation where the data resolution is lower. Even with these compromises, however, some statistics still need to be gathered, some degree of threshold calculation has to be performed and at least one representative output code needs to be calculated and communicated to the decoder for each threshold move.

In terms of performance, Heckbert's median-cut optimisation works well when the image gamut is much smaller than the colour space. It falls down, however, in sparse areas of the colour space where input colours are lost which are deemed statistically insignificant but define image details that are important to object recognition (such as spectral reflections). This problem is exacerbated at the low output resolutions considered in this application where the level of colour use that defines statistical significance is high. The problem here is that the error metric does not represent all factors of what is regarded as optimum quantisation by a human observer, a failing recognised by Heckbert himself.

Error diffusion can go some way to solving this problem as it can approximate missing intermediate shades. However, the requirements of error diffusion further complicate the optimisation problem, as for it to be successful, it is not the distance from each input code to the nearest output code that matters, but some combination of the distances from each input code to both the nearest codes below and above. In addition, in an error diffusion system, the input to the quantiser is not the raw pixel stream, but the pixel after diffusion. Thus it is the statistics of this modified data set that are important, not simply those of the raw data. Definition of a metric that can be used in optimising thresholds to satisfy a human observer after error diffusion is a problem that requires further research.

In summary, the benefits of applying a non-linear scaling function prior to quantisation are slight, and the advantage of allowing adaptive quantisation is ruled out by the high cost of its implementation. In addition, without the definition of a more suitable error metric, the benefits of using adaptive quantisation with error diffusion at very low output resolution are questionable. In the context of an error diffusion system and the video transmitter application, the inflexibility of the binary truncation quantiser is therefore not an issue. In fact, there is a further benefit of the binary truncator in an error diffusion system in that the quantisation error term (that needs to be generated for feedback to the diffuser (see Figure 3.7)) is available at no cost. It is simply the least significant bits of the input - those that do not form part of the quantised output.

Increased dynamic range problem

During the software simulation, an additional complication of the behaviour expected of a quantiser in error diffusion was discovered. The problem is that if the raw pixel stream input to the diffusion system has a dynamic range that is a round power of 2 then the range of the input signal to the quantiser is not. This has ramifications upon the suitability of the binary truncating quantiser. Its fixed thresholds are only evenly spaced throughout the input range if that range is a round power of 2.

The difference between the dynamic ranges of the input pixel stream and the input to the quantiser is due to the summing action of the diffuser. The range of the input to the quantiser is in fact equal to that of the raw pixel input signal plus the magnitude of the largest possible error that is fed back from the quantiser. The magnitude of the largest possible error is determined by the spacing of the quantiser thresholds.

Given the ability to set arbitrary thresholds, the problem could be solved by increasing the dynamic range of the quantiser and spacing the thresholds accordingly. The use of arbitrary thresholds implies a cost in hardware implementation, however, as outlined above. The only way to allow the use of the binary truncator is to ensure that the input signal to the quantiser has a dynamic range that is a round power of 2. There are two ways this can be done: clip the input to the quantiser or apply digital attenuation to the input signal such that its resulting dynamic range plus that of the largest error is a round power of 2.

Clipping at the quantiser input is a relatively cheap option to implement in hardware as it translates to the logical OR-ing of all the bits of the diffuser output with a carry-flag from the diffusion sum. Use of a clip would, however, lead to a non-linear distortion as pixel energy is only lost in bright areas of images. Where the output greyscale resolution is relatively high (above 5 bpp) it may be hard to spot the distortion, as the loss is such a small component of the full dynamic range. As the resolution drops, however, it becomes increasingly significant.

The second option suggested above is to digitally apply attenuation to scale the input signal. Initially this sounds like an expensive option to implement as division is typically an expensive hardware operation. However, it turns out that all the divisions that are required can be performed using a binary shift and subtract operation. This solution can therefore be performed using a single subtracter. This means that the binary truncation quantiser can still be employed at lower cost than the more sophisticated quantisers.

As the scaling operation is performed before entering the actual error diffusion system it is referred to in the remainder of the thesis as *pre*-scaling. Figure 3.10 shows the position of the prescaler in the rest of the error diffusion system.

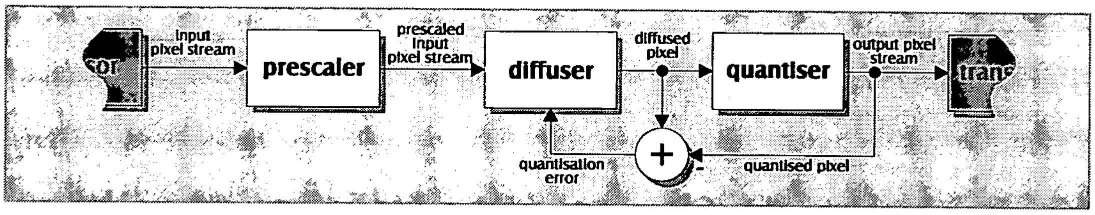


Figure 3.10 ***The position of the pre-scaler operation*** *with respect to the error diffuser. (If it were placed after the diffuser, error terms would pass through the scaler twice and would thus be reduced in significance.)*

The dependence of the increase in dynamic range upon the spacing of the quantisation thresholds has another implication on the use of adaptive quantisation with error diffusion. If such a system is to be implemented without the non-linear problem of clipping the quantiser input, either the quantiser will have to be designed to cope with an input of nearly twice the dynamic range as the raw pixel data or some limitations will have to be imposed on the spacing of the thresholds so as to guarantee a lower signal range.

Summary

The sophistication of the thresholds that a quantiser can use determines both the possible quantising action it can offer and the ways in which it can be implemented in hardware.

In selecting the type of quantiser to be used in the error diffusion algorithms, both the different behaviours the alternative architectures offer and the costs of their hardware implementations are considered.

After discounting the requirement for adaptive quantisation or application of a non-linear scaling function in the radio video link application, a binary truncation behaviour is chosen for the quantiser as it satisfies the requirements of the error diffusion algorithms and it allows both the quantisation operation and error-term calculation to be implemented without hardware expense.

The variable output resolution of the test system makes the inclusion of a variable prescaler stage necessary to maintain the dynamic range of the quantiser input constant at a round power of 2. In a fixed resolution application, set-up of the ADC could be used to ensure the correct input signal dynamic range instead.

Diffusion filter design

The diffusion filter is critical to the nature of the patterning that an error diffusion system produces. The measure of a filter's success is the degree to which it can produce a dither pattern that achieves the illusion of continuous tone output, under all input conditions, without masking image detail. As error diffusion tends to be used in conditions where the individual image pixels can be resolved it is particularly important that the diffusion patterning itself should have as little 'interesting' structure as possible, so as not to distract the observer from the detail of image.

Diffusion filters vary in the number of filter elements they contain (their size), the spatial arrangement of the elements (their shape), the way the quantisation errors are divided amongst the filter elements and in the order that the data is presented to the filter. These flexibilities relate to design decisions that trade off performance factors against each other and against the filter implementation costs. Processing order, filter size, filter shape and filter weight flexibilities are considered below.

Processing order

The fact that error diffusion is a neighbourhood process[6] (and therefore inherently sequential in nature) places constraints upon the shape of the diffusion filter. For a pixel location to be a valid candidate for a filter element it must be as yet unquantised. Thus, the raster order in which the data is processed partly defines the filter shape.

[6] In contrast to a *point* process, neighbouring pixels are not processed independently in a *neighbourhood* process.

Processing in conventional progressive raster order is generally favoured as this mimics the order in which image data is normally acquired, stored and otherwise manipulated (thus minimising buffering/re-ordering). The spatial polarity of the conventional raster does, however, have the unfortunate consequence of forcing a directional structure on the diffusion process. This has two results; a tendency for directional qualities to appear in the diffusion patterning produced and a shift of image detail in the directions that the raster progresses. This second phenomenon is often referred to as phase error [*STOFFEL & MORELAND 81*].

These directional problems can be alleviated through the use of alternative rasters. The drawback of using a non-standard processing order is incompatibility with any conventional raster components of the system. Re-ordering the pixels implies buffering. The further the alternative raster deviates from the conventional order used by other parts of the system, the higher the buffering overhead.

A common alternative processing order is the serpentine raster. The order that image lines are processed in a serpentine raster is the same as a conventional progressive raster. Within successive lines, however, pixel processing alternates between left-to-right and right-to-left. Serpentine raster is generally used to mask horizontal bias, but it suffers from the same vertical bias as the conventional raster.

The use of more radical rasters has also been investigated, in particular, the use of a special class of fractal functions known as 'space-filling curves' which have a localised pseudo-random behaviour [*WITTEN & NEAL 82; VELHO & GOMES 91*]. Space-filling curves can be used to remove most of the visibly directional quantities introduced by the raster, however the introduction of a random element to the processing generates low frequency noise in the diffusion output. Unlike diffusion with a more conventional raster, this noise cannot be 'tuned out' by altering the diffusion filter because the raster order lacks the deterministic qualities necessary.

Other attempts to escape from the restrictions of the raster order include the use of iterative diffusion functions [*MULLIGAN & AHUMADA 92*]. These typically make several passes over the input data set, allowing them to use arbitrary filter element positions. Iterative diffusion permits tuning of the filter to perform in a certain manner. The disadvantages are the increased amount of processing necessary and the very high memory overheads.

The implied memory overheads of alternative serial processing orders and iterative processing make these options unattractive for use in the radio video link application. Serial processing of the data in a conventional raster scan order carries no such overhead, however, there is a cost as it forces a directional bias on the diffusion ability of the filter.

Filter size

The size of a diffusion filter affects both the possible diffusion patterns it can generate and the amount of hardware that is required to implement it. As each filter element implies both computational hardware and error storage, the distance between elements (with respect to the order of the incoming data stream) determines the amount of memory required to store errors (from when they are generated to when they are diffused). Thus the two factors of size that influence the amount of hardware required are the number of filter elements and how they are arranged spatially. These factors are also largely responsible for affecting the possible diffusion patterns produced by the filter. The most important of these features are the success of the filter in minimising the area over which errors cancel and the degree to which the patterns they produce contain a potentially distracting structure.

In general, four element filters (such as that of Floyd and Steinberg) are used. However, larger filters have been proposed in attempts to alleviate patterning problems (in particular the 12 element filter of Jarvis *et al.* [*JARVIS JUDICE & NINKE 76*]). The larger the filter the more sophisticated the options for diffusion patterns, however, the further apart the elements the more the error energy is spread from its source. Energy being spread predominantly in one direction is undesirable, it leads to a phenomenon referred to as *phase shift* - where sharp details such as edges tend to move in one direction within the image [*STOFFEL & MORELAND 81*].

Extension of error diffusion from the normal 2-dimensional spatial diffusion into 3 dimensions is possible by including a temporal component. This can be achieved by using a 3-dimensional filter that propagates errors not only to adjacent pixels in the same frame, but also to pixels into future frames. Mulligan proposed the use of spatio-temporal diffusion filters for the display of static images using dynamic displays [*MULLIGAN 93*]. He reported advantages of increased greyscale resolution (through increased averaging) and improved perceptual segregation of picture and noise due to their separation in different temporal bands. Temporal dithering exploits the insensitivity of the human visual system to high frequency temporal patterns in much the same way that spatial dithering exploits high spatial frequency pattern insensitivity. Research has shown marked similarities between the spatial and temporal contrast sensitivity functions of the human visual system [*ROBSON 66; PEARSON 75*].

When used on image sequences there is an incidental temporal component to the diffusion pattern produced by even a 2 dimensional filter. This is caused by the accidental animation of the diffusion pattern by even the smallest image changes or by any noise in the image data. Unfortunately the controlled use of temporal diffusion through the use of a temporal filter requires buffering the errors from an entire frame of image data. This overhead is too large for the radio video link application, thus temporal filters have to be ruled out.

Very small diffusion filters are generally discounted because they are unable to produce sufficiently sophisticated patterning. Floyd and Steinberg themselves argued that a four element filter was the smallest that could be used to produce 'good' results. This conclusion is supported by Ulichney who reports the failings of smaller filters [*ULICHNEY 88*].

However, these findings may not be strictly applicable to the video transmitter application. In common with most published work on the subject, Floyd & Steinberg and Ulichney were concerned with the rendering of still images using bi-level output devices. In contrast, in the radio video link, the problem is of rendering moving images using a limited greyscale. The low hardware implications of small filters makes them attractive in the video transmitter application, thus a small filter should be used if the type of patterning it produces can be tolerated when used to reduce the moving data to a modest greyscale.

Filter weights

A further factor that affects both diffusion patterning and filter implementation costs is the fractions that are used to divide up errors amongst the filter elements. These fractions are generally termed the *weights*.

The complication of the hardware generation of the error fractions depends upon the choice of weights. One way of saving hardware is to restrict weight choice to powers of two, allowing all necessary multiplication to be performed using bit-wise shifts.

The introduction of a random element to diffusion processing can be used to break up unwanted structure. One that is relatively cheap to implement in hardware is the perturbation of the filter weights. Ulichney attributes the idea of using perturbed weight sets to Schreiber and the first demonstration of it to Woo [ULICHNEY 88]. Perturbation must be used carefully, however, as it can lead to low frequency noise from the perturbing signal becoming apparent in the filter output.

Summary

Several features of the diffusion filter can be altered to trade off different performance aspects and performance versus hardware implementation costs.

The desire for a hardware-minimal implementation of the radio video link favours the use of conventional raster scan processing, with as small a diffusion filter as possible. The memory cost of a temporal diffusion filter precludes its use. To simplify computation, where possible, filter weights should be kept to powers of two.

Using a conventional raster processing order imposes a severe horizontal and vertical bias on the action of the diffusion filter. In other applications where this bias is intolerable the horizontal component can be alleviated by using a serpentine raster without an excessive penalty in memory.

The use of a small diffusion filter is desirable in terms of patterning hysteresis and preservation of edges as well as its minimal hardware implications. The danger of a small filter is the limits this imposes on the patterning sophistication. Incorporation of a random element into the diffusion filter may alleviate problems of structured patterning at lower cost than increased filter size. Random perturbation should, however, be used carefully so as not to introduce too much low frequency noise to the patterning which can also be distracting. The proposed modest colour resolution of the radio video link should make filters smaller than those generally used for bi-level output acceptable.

Even though an explicit temporal filter is too expensive, an element of 'incidental' temporal diffusion should result from the proposed pipeline architecture used in any application where there are slight inter-frame image changes or even the presence of a small amount of noise in the image data. Whether the incidental temporal changes that occur serve to mask the diffusion patterning or distract the observer from the image data is considered later in chapter five.

Using the criteria defined above, four filters ('simple', 'perturb', 'safe perturb 1' and 'safe perturb 2') are considered below for use in the coder of the video transmitter.

The 'simple' filter

The first filter considered is the simplest possible to implement in hardware. The filter, referred to here as *simple*, consists of one fixed element and thus requires only one adder. As there is only one filter element there is no need to divide up each diffusion error thus there is no need for multiplication hardware to compute error components.

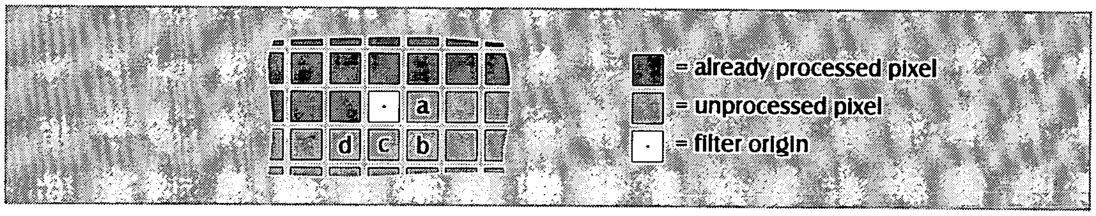


*Figure 3.11 The four **un-quantised pixels** adjacent to the filter origin in raster scan processing (labelled 'a' to 'd').*

The four pixels marked 'a' to 'd' in Figure 3.11 are the un-quantised pixels adjacent to the filter origin when using a raster scan. In terms of being candidates for error diffusion all four pixels are very similar, as they don't differ greatly in spatial distance from the origin. If used as the sole element of a diffusion filter all would lead to very similar and very directional diffusion patterns (the actual direction of the directional bias would be the only difference). They do, however, differ in their distance from the filter origin in terms of the raster scan pixel order. This affects the hardware implementation of the filter as the distance between the filter origin and an element defines over how many pixel cycles each quantisation error must be stored before it is diffused to that element. This distance therefore defines how much error storage memory is required. Pixel 'a' in Figure 3.11, the closest to the origin, was chosen for the *simple* filter to give the most minimal implementation possible. The *simple* filter is shown schematically in Figure 3.12.

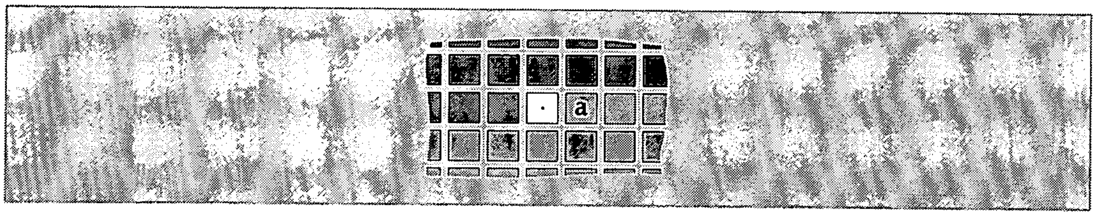


Figure 3.12 ***A schematic of the 'simple' diffusion filter.*** *'.' denotes the position of the pixel being quantised (the 'filter origin'), and 'a' is the destination for the whole of the corresponding quantisation error - i.e. the sole element of the diffusion filter.*

As well as being the cheapest to implement, this is arguably the crudest error diffusion filter possible. Ulichney includes single-element filters in a review of error-diffusion filters purely to point that they "fail in a big way" [*ULICHNEY 88*]. He was, however, considering them for use with bi-level output devices (i.e. 1 bpp), here they are being considered for up to 4 bpp. The action of the *simple* filter is similar to 'error-feedback' rounding techniques. These are used in high quality image processing when the high precision results of pixel manipulation are reduced in precision for recording and display. In contrast to the bi-level case, however, 'error-feedback' is concerned with conversion typically from 32 or 16 bpp to 16 or 8 [*JACK 93*]. Use of the simple filter for the modest compression required by the radio video link represents an application somewhere between these two examples.

The source code for the software implementation of the *simple* filter used in the simulation software is given in Listing 1. The function (`Quantise_Diffuse`) is designed to quantise to any output resolution lower than the input 8 bpp. The quantisation and error calculation are implemented using 'bit-wise' AND operations. This method facilitates variation of the output resolution through alteration of the two masks used in the AND operations. A trap to prevent pixel roll-over[7] is implemented by the IF statement.

```
void Quantise_Diffuse( byte *pSrc, byte *pDest, byte bits )
{
   dword pixel, end_of_frame=(dword)(FRAME_ROWS*FRAME_COLS);
   byte sum=0; /* temporary quantisation error storage */
   byte sum_mask = (byte)(0xff>>bits);
   byte pixel_mask = (byte)(0xff<<(8-bits));

   for(pixel=0; pixel<end_of_frame; pixel++){
      if (*pSrc < pixel_mask) sum = (byte)( *pSrc++ + (sum & sum_mask));
      else sum = *pSrc++;
      *pDest++ = (byte)(sum & pixel_mask);
   }
}
```

*Listing 1   function **Quantise_Diffuse()** from quantise.c (see appendix one for full listing).*

Two sets of example output from *simple* are shown in Figure 3.13 and Figure 3.14 below. The two source images (*lena* and *salesman*) display the effectiveness of the algorithm in areas of both high and low spatial frequency. With the exception of the feather, most of *lena* is low frequency. It contains large areas of smoothly changing colour such as the shoulder and the reflection in the mirror. The *salesman* image does not contain many areas of smooth colour, but has far more areas of high frequency content. Subjective examination of these still images indicates that if the output resolution is kept relatively high (3 or 4 bpp - the top halves of Figure 3.13 and Figure 3.14) the output of this filter is clearly better than that of a truncating quantiser and only marginally worse than the raw data itself. These resolutions represent compression ratios of 2:1 and 2.66:1. The images in the lower halves of the figures show the results of processing to the lowest integer colour resolutions (1 and 2 bpp). At these resolutions, the output of the simple filter still conveys more information about the detail of the image than the truncated images, however, the presence of the diffusion pattern becomes obvious. At 1 bpp a wood grain-like structure within the pattern is certainly apparent (and arguably quite objectionable). Processing to the output resolutions of 1 and 2 bpp represents compression ratios of 8:1 and 4:1, respectively.

[7] Roll-over is a term used to describe the corruption of a pixel value through either overflow or underflow during its manipulation (e.g. if a pixel, described using 8 bits, which initially has a value of 254, has 10 added to it the result will be 8 if rollover is not prevented).

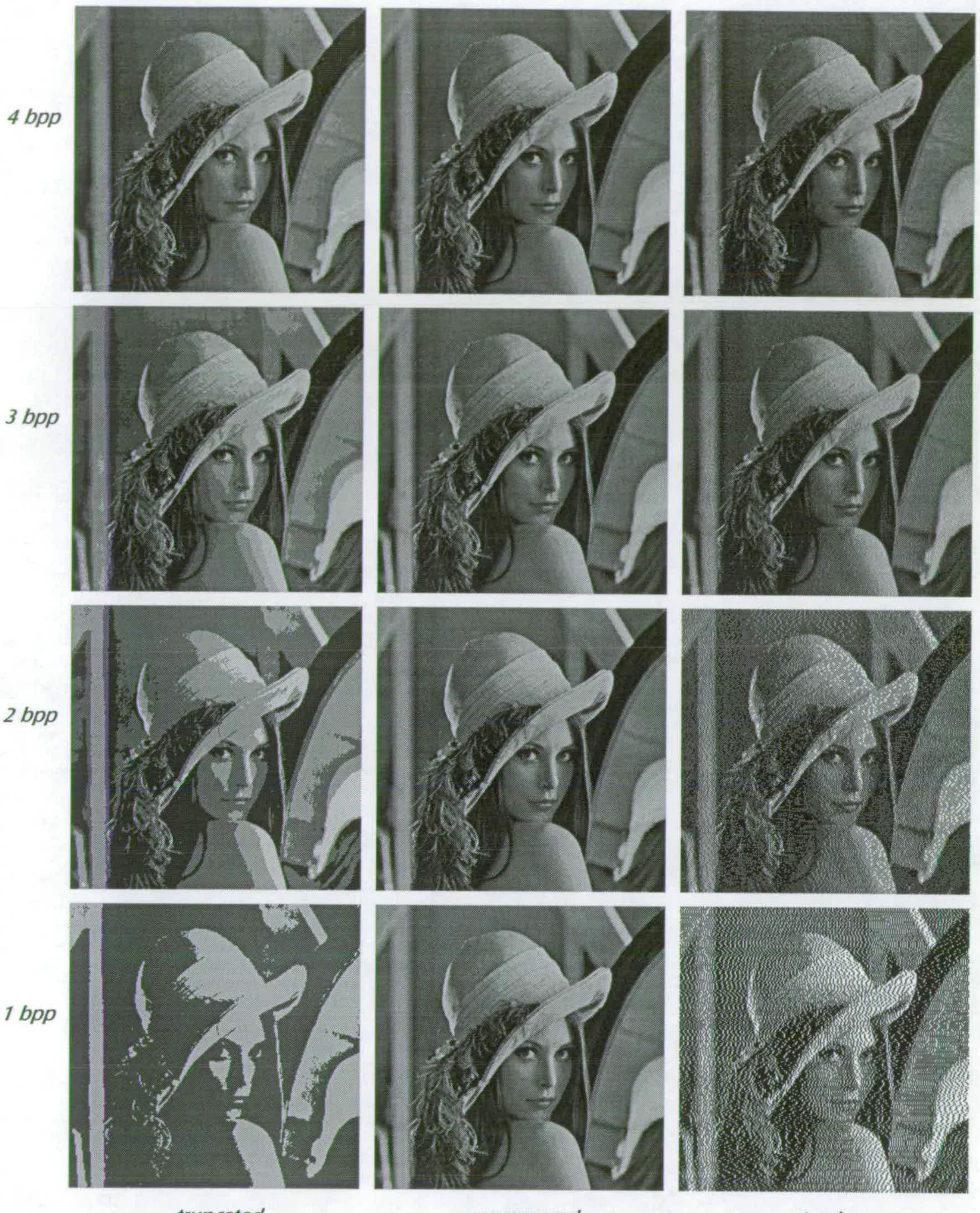


Figure 3.13 ***Examples of the 'simple' filter output*** *(lena), shown together with the un-processed image and the image after a truncating quantisation of the pixel values to the same pixel precision as the simple filter for comparison. The un-processed image (8 bpp) is shown in the centre of each horizontal triplet, the images on the right hand side have been processed with the 'simple' diffusion filter, and the left hand side have been quantised by truncation. The greyscale resolutions of the processed images range from 4 bpp in the top line to 1 bpp in the bottom line.*

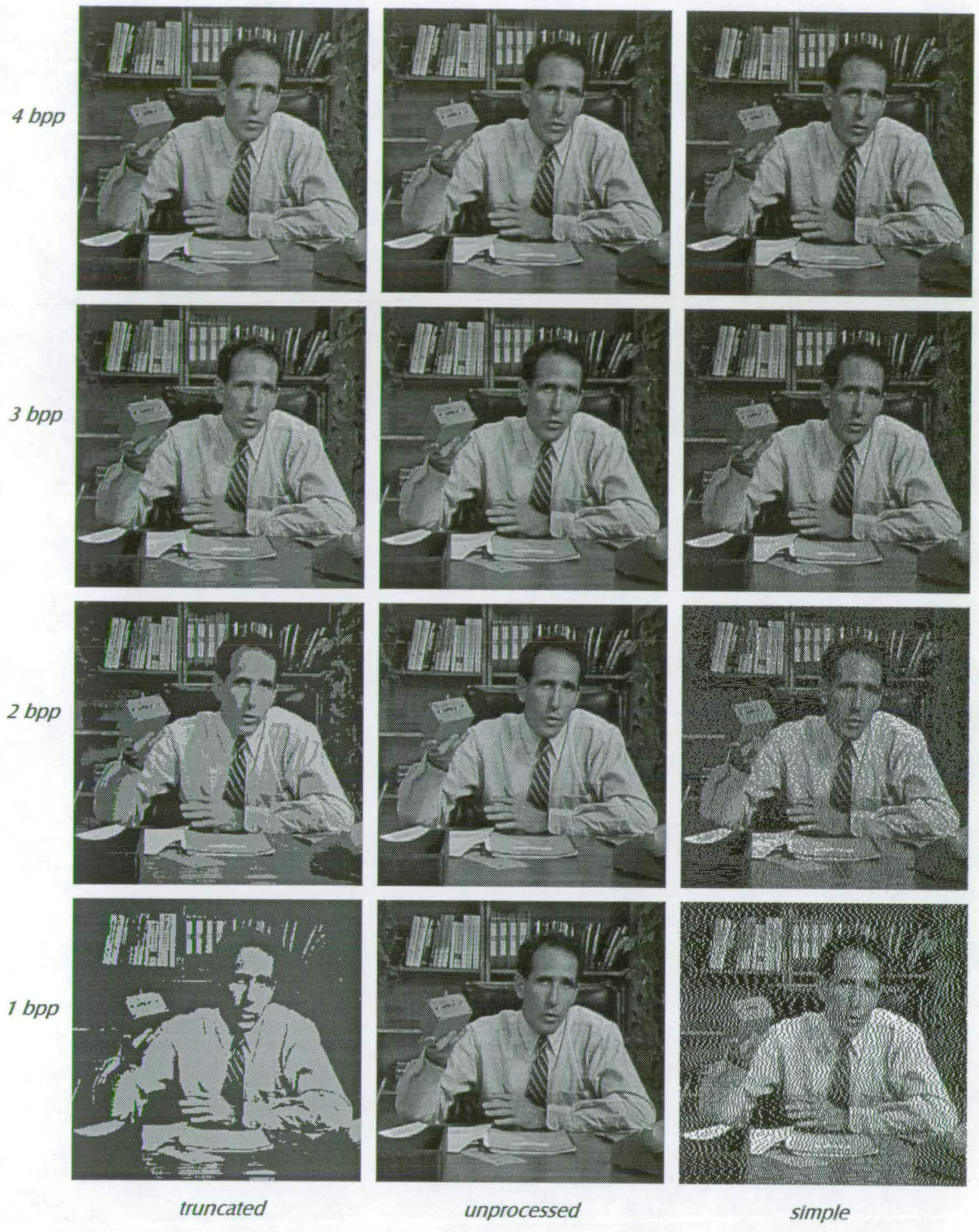


Figure 3.14 ***Examples of the 'simple' filter output*** *(salesman frame 0), shown together with the un-processed image and the image after a truncating quantisation of the pixel values to the same pixel precision as the simple filter for comparison. The un-processed image (8 bpp) is shown in the centre of each horizontal triplet, the images on the right hand side have been processed with the 'simple' diffusion filter, and the left hand side have been quantised by truncation. The greyscale resolutions of the processed images range from 4 bpp in the top line to 1 bpp in the bottom line.*

Both Figure 3.13 and Figure 3.14 compare static results from the *simple* filter and the truncating quantiser. These conditions do not directly reflect those of the video transmitter as its image data would be changing at a rate of at least 10 fps. Unfortunately, display of moving images is not possible here. The simulation software, however, permits the processing and display of both live video and short sequences at up to approx. 4 frames per second in addition to the processing and display of stills. For convenience, live video input was used for most of the moving image tests. This minimised the amount of disk space and disc access required. As expected the effect that the use of moving images had on the success/visibility of the diffusion pattern was largely dependent on image content and output resolution. In areas of images where there was much high frequency content animation of the diffusion pattern served to mask its presence, however, in flatter regions of images the pattern didn't change sufficiently frame to frame to hide its structure. The lower the resolution the less the animation masked the pattern.

The lack of sophistication in the pattern produced by *simple* is evident when compared with the performance of Floyd and Steinberg's filter at 1 bpp. Figure 3.15 offers such comparison plus comparisons at higher greyscale resolutions. The images diffused with Floyd & Steinberg's filter were produced using a variable resolution implementation of the filter (as given in Listing 2). In the 1 bpp image from *simple* the low frequency structure and directional hysteresis combine to create a wood-grain like pattern. The 1 bpp image processed by the *floyd-steinberg* filter shows little of this same type of structure except for some directional hysteresis in the darkest flat areas of the image such as the left-hand side of the chair and between the salesman's left arm and his body. As the output resolution is increased, however, the difference between the output of the two filters drops off rapidly. At 4 bpp these static results become difficult to distinguish.

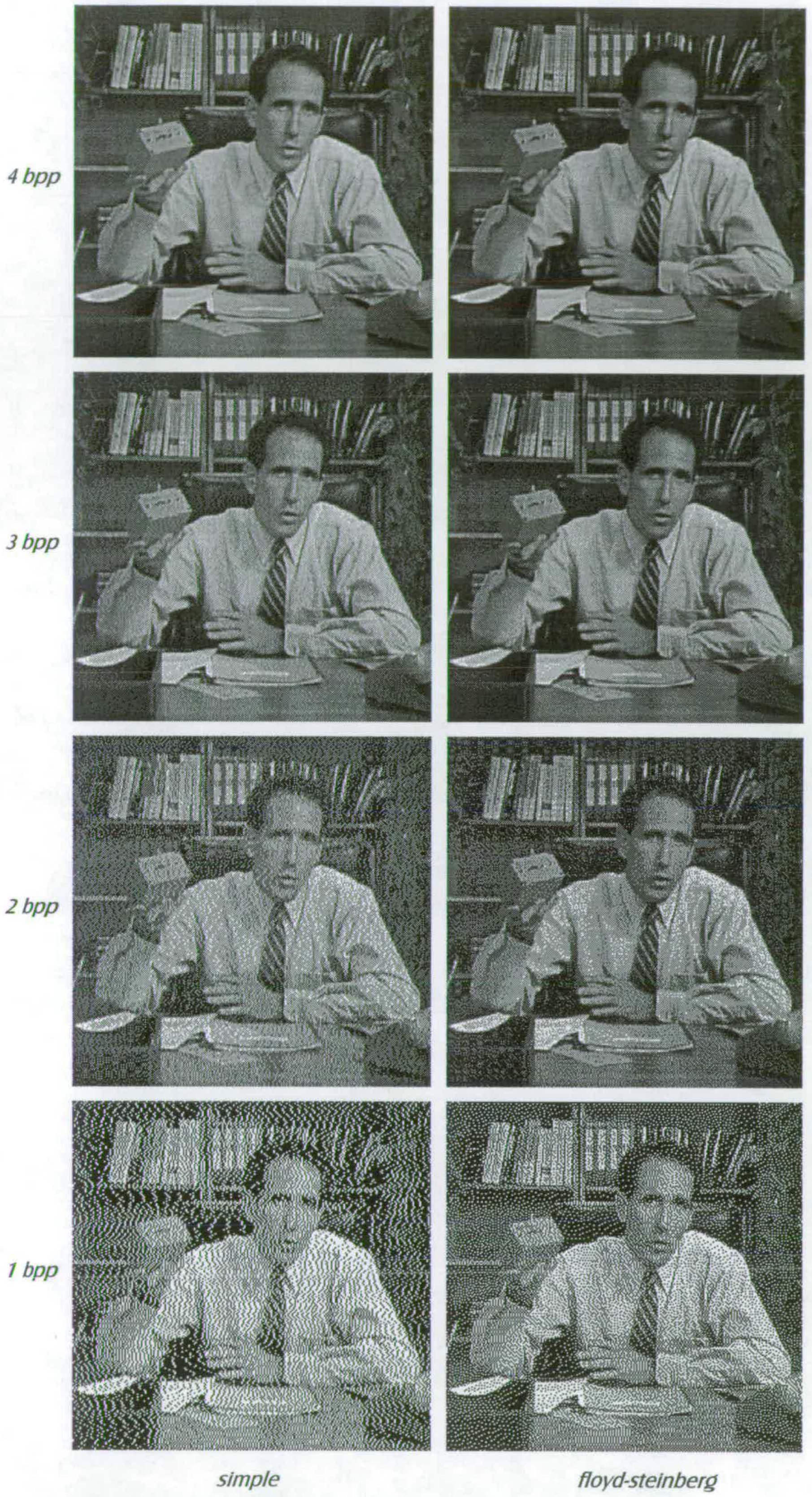


Figure 3.15 ***Comparison of simple and floyd-steinberg*** *(salesman frame 0). The images on the left hand side have been processed with the 'simple' diffusion filter, and those on the right hand side with a variable resolution implementation of Floyd & Steinberg's filter. The greyscale resolutions of the processed images range from 4 bpp in the top line to 1 bpp in the bottom line.*

Ulichney's dismissal of the *simple* filter when used at a 1 bpp and the success of it demonstrated here at 3 and 4 bpp shows that a minimal increase in greyscale resolution is significant to the acceptability of the filter output. This emphasises the difference between filter considerations for applications with bi-level output and those with even a modest greyscale.

```
int Quantise_FloydS( byte *pSrc, byte *pDest, byte bits )
{
    unsigned short fifo_index, fifo_len = frame_cols+1;
    short fifo[FRAME_COLS+1], threholds[64];
    long SpreadPixel, QuantError;
    byte QuantisedPixel, colours[64];
    dword pixel, end_of_frame = frame_rows*frame_cols;
    dword a_limit = end_of_frame - 1, b_limit = a_limit - frame_cols;
    byte num_thresholds=(byte)pow(2,bits), threshold;

    /* create the array of thresholds and corresponding array of colours */
    for(threshold=0; threshold<num_thresholds; threshold++) {
        thresholds[threshold] = (short)(255*(2*threshold-1))/(2*(num_thresholds-1));
        colours[threshold] = (threshold*255)/(num_thresholds-1);
    }

    for(fifo_index=0; fifo_index<fifo_len; fifo_index++)
        fifo[fifo_index]=0; /* intialise the spreading array */
    fifo_index=0; /* - probably not necessary as the fifo buffer is circular */

    for(pixel=0; pixel<end_of_frame; pixel++){                /* process image */
        /* calculate the spread pixel, quantised version and error */
        SpreadPixel = ( (((long)*pSrc++)<<4) + fifo[(fifo_index++)%fifo_len])>>4;
        threshold = num_thresholds-1;
        while (SpreadPixel<thresholds[threshold]) threshold--;
        QuantisedPixel = colours[threshold];
        QuantError = SpreadPixel - (long)QuantisedPixel;

        *pDest++ = QuantisedPixel; /* store the results (errors*16)*/
        if (pixel<a_limit)
            fifo[fifo_index%fifo_len] += QuantError*7; /* filter element A */
        if (pixel<b_limit){
            fifo[(fifo_index+frame_cols)%fifo_len] = QuantError; /* B */
            fifo[(fifo_index+frame_cols-1)%fifo_len] += QuantError*5; /* C */
            fifo[(fifo_index+frame_cols-2)%fifo_len] += QuantError*3; /* D */
        }
    }
    return TRUE;
}
```

*Listing 2 The function **Quantise_FloydS()** from quantise.c (see appendix one for full listing).*

The *simple* filter works, in that neighbouring pixel errors in the output it produces tend to cancel. At the lowest integer resolutions there is, however, a large amount of visible structure introduced. The visibility of this structure is sometimes masked by image movement, but not necessarily - in some cases it could be argued that it is actually enhanced. Representing one extreme of the cost/complexity trade-off, it is quite likely that *simple* does not offer the optimum balance for the video transmitter application. How much more complicated does a filter have to get in order to become sophisticated enough to produce significantly more attractive patterning? To answer this question three improvements on *simple* are explored below.

Improvement on 'simple'

A combination of three factors are responsible for the 'wood grain-like' structure in the output of the *simple* filter. These are that the filter is one-dimensional, that it is purely deterministic and that the data is being presented to it in a conventional raster scan order. Change of any one of these factors could potentially improve the resulting diffusion pattern. The cost of each option is considered below.

Keeping the raster processing order, the operation of the filter can only be made two-dimensional by expansion of the filter to include an element on another image line. The closest remaining candidates are pixels *b*, *c* and *d* in Figure 3.11. Allowing the filter to spread error energy in two directions should remove the severe directional bias that the single element filter suffers, however, the hardware cost of adding an element on another line is high. As mentioned above, increased filter size implies hardware costs in terms of error term calculation, diffusion and storage. Assuming that simple binary weights are used (e.g. ½ or ¼), the largest expense incurred in increasing the *simple* filter to two elements would be in error term storage, as errors would need to be stored for a complete image line.

Use of an unconventional raster order would alter the diffusion pattern of *simple*. Although the filter would still spread all errors to the next pixel, that pixel would no longer always be to the right of the filter origin. Processing pixels in the diffusion system using an order that differs from the rest of the video link system would require memory and control to re-order the data both as it entered the diffusion system and on its exit. The amount of memory and logic required to manage the re-ordering is dependent on how far the new processing order deviates from the conventional raster. The serpentine raster is probably the order that offers the lowest additional cost. Its use requires one line memory for each conversion and minimal logic (the largest part of which would be a line length counter). Use of the serpentine raster with *simple* would cause errors to be diffused to the left and right on alternate lines - one-dimensional diffusion with an alternating directional bias. Although this would help to alleviate the direction hysteresis, it would not stop the appearance of the vertical wood-grain pattern.

Introduction of a random element to perturb either the shape of the diffusion filter or the weights used can reduce the deterministic nature of the diffusion pattern produced. Although the introduction of noise can be used profitably to break up the deterministic patterns, care has to be taken that the low-frequency content introduced by the perturbing signal does not itself become distracting.

In terms of hardware cost, introduction of serpentine raster order is the most expensive of the three options. Its expense, together with an anticipation of little effect and reports of its limited success when used to alleviate inter-line structure in bi-level output diffusion [WITTEN & NEAL 82] meant that the use of a serpentine raster was not explored. The costs of either a perturbed single element filter or a purely deterministic two dimensional filter are approximately the same. Reports of the successful application of perturbed diffusion filters in *bi-level* quantisation [ULICHNEY 88] led to investigation of the perturbed filter route.

The 'perturb' filter

Like *simple,* the *perturb* filter spreads the whole of each quantisation error onto a single unquantised pixel. Instead of always spreading it onto the pixel to the right of the filter origin, however, this filter is equally likely to spread it onto the pixel below (i.e. either pixel *a* or *b* in Figure 3.16). The choice of which pixel to spread onto is made at random.

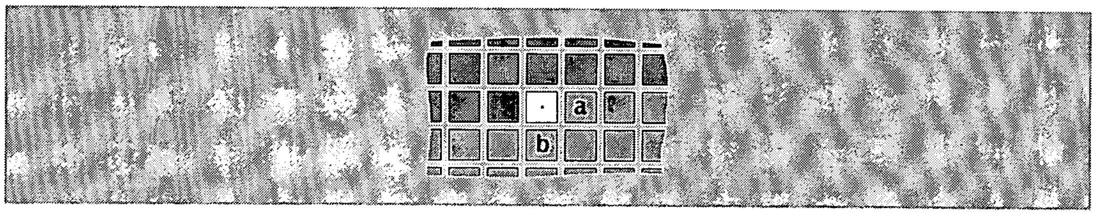


*Figure 3.16 A diagram of **the 'perturb' diffusion filter**. '.' denotes the position of the filter origin and 'a' and 'b' the two possible filter elements.*

When deciding on the location of the second element of the filter, three positions immediately adjacent to the filter origin remain as valid element candidates (pixel *b* in Figure 3.16 and the pixels to the immediate left and right of *b*). The distance between the three candidates and the origin is almost identical, thus in terms of implementation cost all three are roughly the same. Pixel *b* was chosen as it results in a filter that is balanced in its degree of horizontal and vertical diffusion.

The section of source code for the random decision between candidate elements in the *perturb* filter is given in Listing 3.

```
void Quantise_RandDiffuse( byte *pOriginal, byte *pDest, byte bits )
{
    dword pixel_index, end_of_frame=(dword)(frame_rows*frame_cols);
    dword most_of_frame=(end_of_frame-1), last_line=(end_of_frame-frame_cols);
    int candidate_below;
    byte error_mask=(byte)(0xff>>bits), pixel_mask=(byte)(0xff<<(8-bits)), dummy;
  byte *pRaster=pDest, *pSpreadee=pDest, error=0;

    Array_CopyFrame( pOriginal, pDest );

    for(pixel_index=0; pixel_index<most_of_frame; pixel_index++, pRaster++){
        error = (byte)(*pRaster & error_mask);
        *pRaster = (byte)(*pRaster & pixel_mask);

        /* choose the 'spreadee' from the two candidates */
        candidate_below = ((rand()>>6) & 0x01);
        if (candidate_below==FALSE) pSpreadee = (pRaster + 1);
        else {
            if (pixel_index<last_line) pSpreadee = (pRaster + frame_cols);
            else pSpreadee = &dummy;
        }

        if ( *pSpreadee < pixel_mask )
            *pSpreadee = (byte)(*pSpreadee + error);
    }
    *pRaster = (byte)(*pRaster & pixel_mask);
}
```

Listing 3 *The function **Quantise_RandDiffuse()** from quantise.c (see appendix one for full listing).*

In terms of hardware, implementation of *perturb* differs from that of *simple* in the addition of storage for a line of error data, provision for two additions per pixel and the addition of a pseudo-random number generator. The software implementation takes advantage of random access to the pixel data instead of using a buffer for the error data, and it uses a C library function (*rand*) rather than a discretely coded pseudo-random number generator.

Results of the *perturb* filter are shown in Figure 3.17 and Figure 3.18 alongside those of *simple*. A significant difference can be seen between the dithering patterns at all resolutions below 4 bpp.

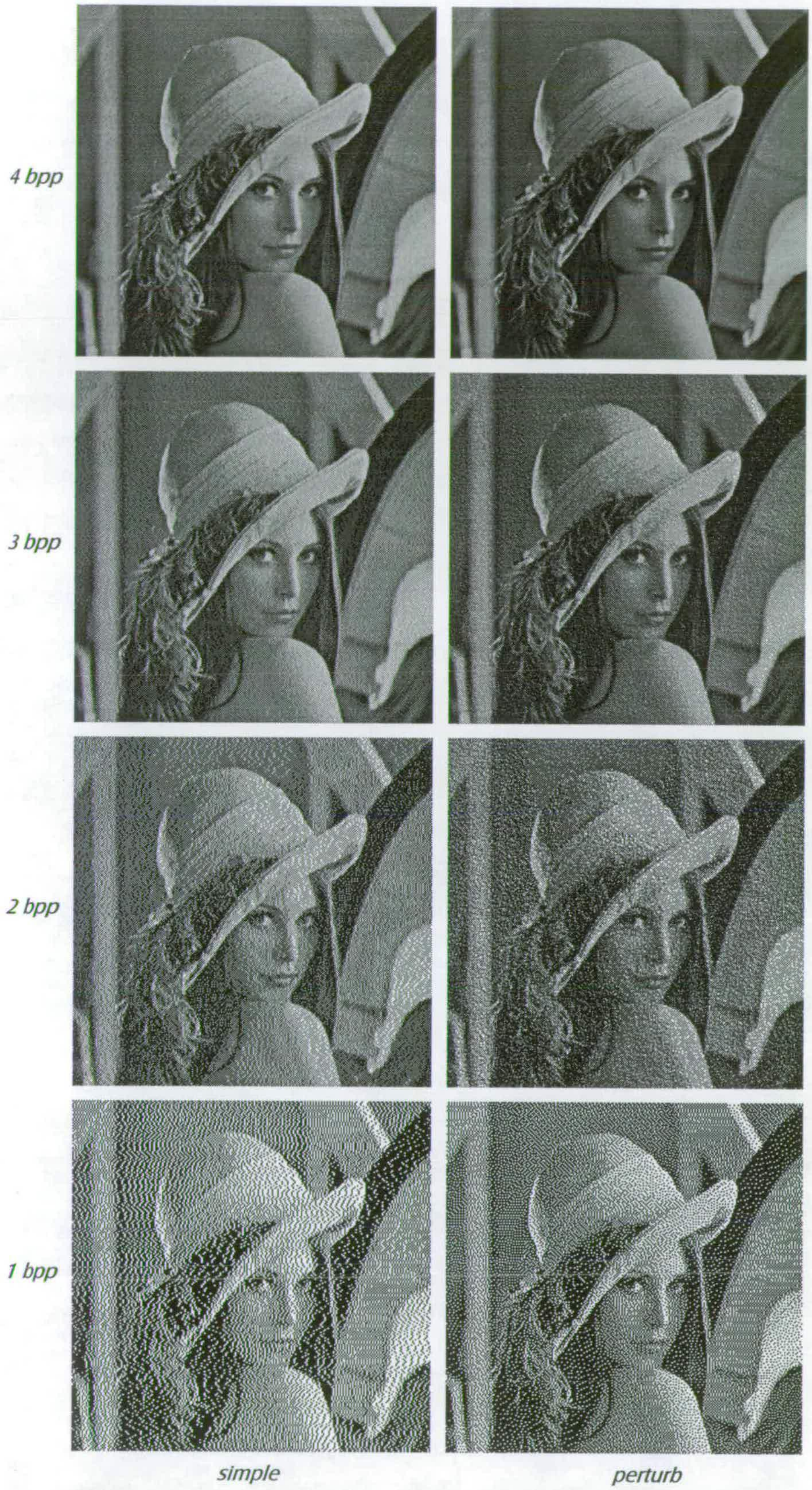


Figure 3.17 ***Comparison of the results of the 'simple' and 'perturb' diffusion filters*** *(lena). The images on the left have been processed with the 'simple' filter, those on right the with 'perturb'. Greyscale resolutions range from 4 bpp in the top line to 1 bpp at the bottom.*

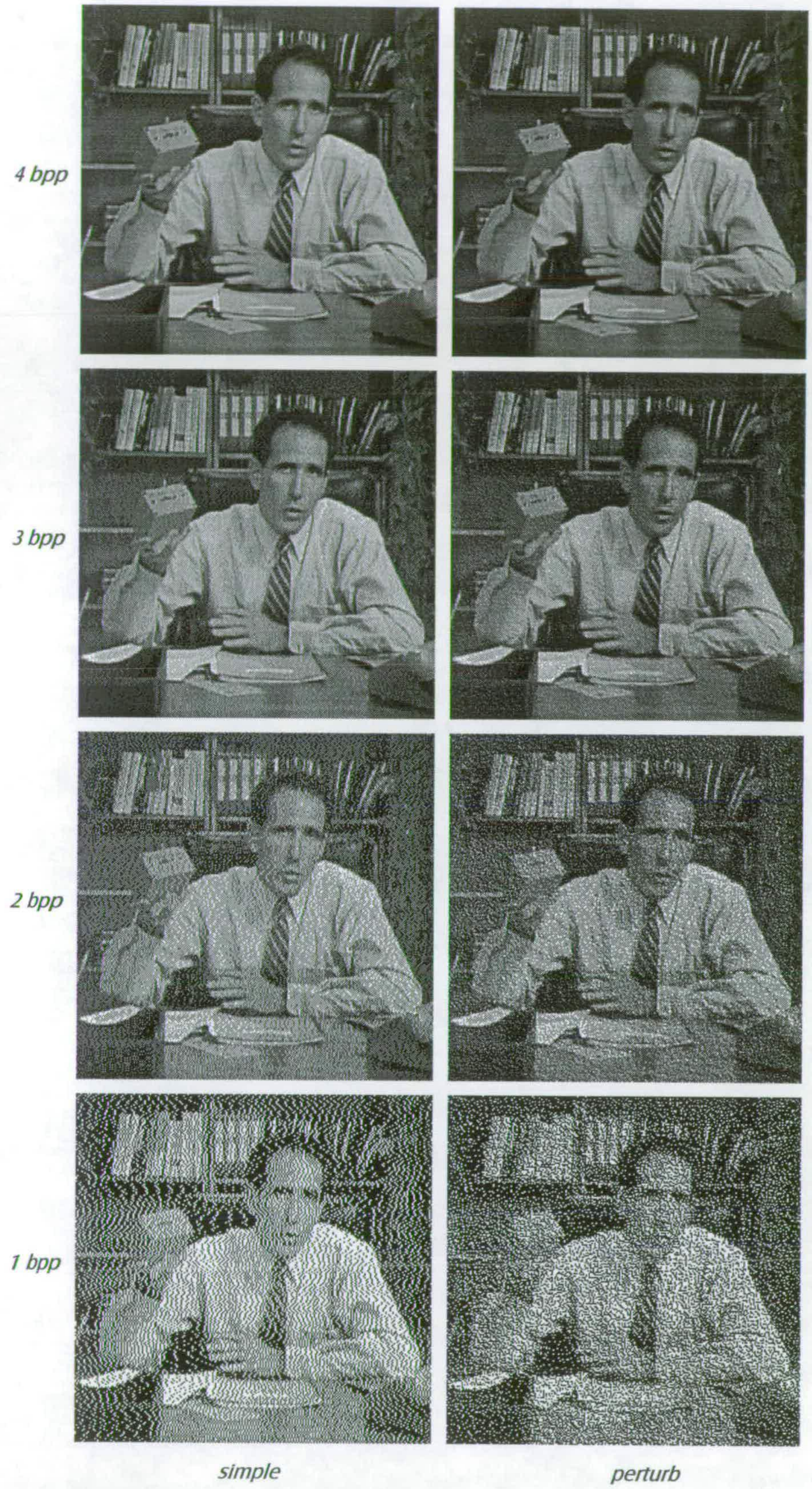


Figure 3.18 ***Comparison of the results of the 'simple' and 'perturb' diffusion filters*** *(salesman frame 0).*

As expected the patterning produced by *perturb* has less visible structure and a more 'noise-like' quality than that produced by *simple*. Unfortunately, the reduction in structure has not led to the desired reduction in visibility of the diffusion pattern.

Closer scrutiny of the diffusion patterns by considering difference images (processed image minus source image) gives some insight into the problem. Consider the sections of the diffusion pattern produced when reducing lena to 2 bpp using *floyd-steinberg*, *perturb* and *simple* shown in Figure 3.19.

(a) 'floyd-steinberg' *(b) 'simple'* *(c) 'perturb'*

*Figure 3.19 Sections of **diffusion pattern** from lena reduced to 2 bpp using (a) 'floyd-steinberg', (b) 'simple', and (c) 'perturb'. These are cropped sections of images that were generated by subtracting the source image from each processed image and biasing the results around mid-grey.*

In Figure 3.19 (b) the wood-grain problem of *simple* is evident. This is the problem that the *perturb* filter was designed to combat. Little of this vertical structure can be seen in the pattern produced by *perturb* (c), thus the introduction of the perturbing element has served to alleviate the wood-grain pattern. This success in removing the wood-grain pattern, however, has not translated to reduced overall pattern visibility. This is due to the complication of a new problem associated with the diffusion pattern produced by *perturb*. Comparison of the grey levels in the three images of Figure 3.19 shows that the pattern produced by the *perturb* filter has a much higher magnitude than the others. It is this magnitude that gives the pattern its visibility.

An increase in diffusion pattern magnitude was not anticipated as a side effect of the *perturb* filter, no mention of such a problem with reference to perturbed error diffusion had been found in the literature. Careful analysis of *perturb* filter output and consideration of the behaviour of the filter in areas where problems were observed revealed the mechanism that was giving rise to the unexpected scale of the noise. With *simple,* the error diffused to any pixel could not exceed the largest step between the quantisation steps used by the quantiser. Indeed, this is true of any deterministic filter as long as the sum of the error components is 100%. In the case of *perturb,* however, the total amount of error energy added to a pixel can exceed the largest quantisation step if the errors from two quantisations are added to it. If the sum of the two errors is greater than the next quantisation step, and the pixel that they are added to was already close to the next quantisation threshold, the resulting pixel may end up two quantised output shades higher than some of its neighbours. The increased inter-pixel contrast that stems from these 'double diffusions' explains the magnitude of the pattern observed in the output of *perturb.*

The problem does not occur in the 1 bpp case as there is only one quantisation threshold and two output shades. Where double diffusions occur that result in the addition of a large amount of energy in this case, much of the energy is simply diffused further away. This difference in behaviour above 1 bpp may explain why mention of this effect was not found in the literature.

The 'safe perturb 1' filter

A solution to the problems of *perturb* was sought. The resulting filter, *safe perturb 1,* does not make the choice between filter candidates at random. The extra-bright pixels in *perturb's* output occurred when two errors were added to one pixel and both the additions caused the value of the unquantised pixel to cross quantisation thresholds. Constraining the freedom with which the filter can make candidate choices in such a way that it only chooses candidates that have not already crossed a quantisation threshold (as the result of a previous diffusion) prevents this 'double significant diffusion' problem from occurring. In the case of the two candidates used in *perturb,* diffusions can only have previously happened to the pixel to the right of the filter origin. Thus the pixel below the origin can always be diffused to without it crossing two quantisation thresholds when using uniform quantisation steps. This is the behaviour exhibited by *safe perturb 1*. Source code for the portion of the simulation function used for *safe perturb 1* that differs from that used for *perturb* is given in Listing 4.

```
void Quantise_SafeRandDiffuse1( byte *pOriginal, byte *pDest, byte *pTemp, byte bits
)
{

    ... [code removed] ...

    for(index=0; index<most_of_frame; index++, pRaster++, spreading=TRUE)       {
       error = (byte)(*pRaster & error_mask); /* impending quantisation error  */
       *pRaster = (byte)(*pRaster & pixel_mask); /* quantise the pixel */

       candidate_below = ((rand()>>6) & 0x01);        /* initial random choice */
       if (candidate_below==FALSE) pSpreadee = (pRaster + 1);
       else {
          if (index<last_line) pSpreadee = (pRaster + frame_cols);
          else spreading = FALSE;
       }
       if (*(pSpreadee+flag_gap)==TRUE) {
          if (index<last_line) pSpreadee = (pRaster + frame_cols);
          else spreading = FALSE;
       }

       if ((spreading==TRUE) && (*pSpreadee < pixel_mask))     {
          old_bit = (byte)(*pSpreadee & flag_mask);
          *pSpreadee = (byte)(*pSpreadee + error);
          if ((*pSpreadee&flag_mask)!=old_bit) *(pSpreadee+flag_gap) = TRUE;
       }
    }

    ... [code removed] ...
```

Listing 4 *Part of the function* ***Quantise_SafeRandDiffuse1()*** *from quantise.c (see appendix one for full listing).*

In terms of hardware, implementation of *safe perturb 1* differs from that of *perturb* in the added requirement for a 'significant diffusion' flagging system. A Boolean flag needs to be generated indicating whether each pixel has undergone significant diffusion when it was the pixel below the filter origin. The error steering logic that controls where error components are diffused needs to be altered to take this flag as an input. In addition, instead of just storing a line of pixel errors (as was possible with *perturb*), the result of the diffusions of error energy from above must be computed a line before those from the left. These partial results are stored for the whole line as the flag from the first diffusion must be calculated before the destination of the quantisation error from the pixel above is decided.

Results of diffusion using the *safe perturb 1* filter are shown together with the same images produced using *perturb* in Figure 3.20 and Figure 3.21. A greyscale ramp is used as the source image in Figure 3.21 to highlight the differences between the two algorithms.

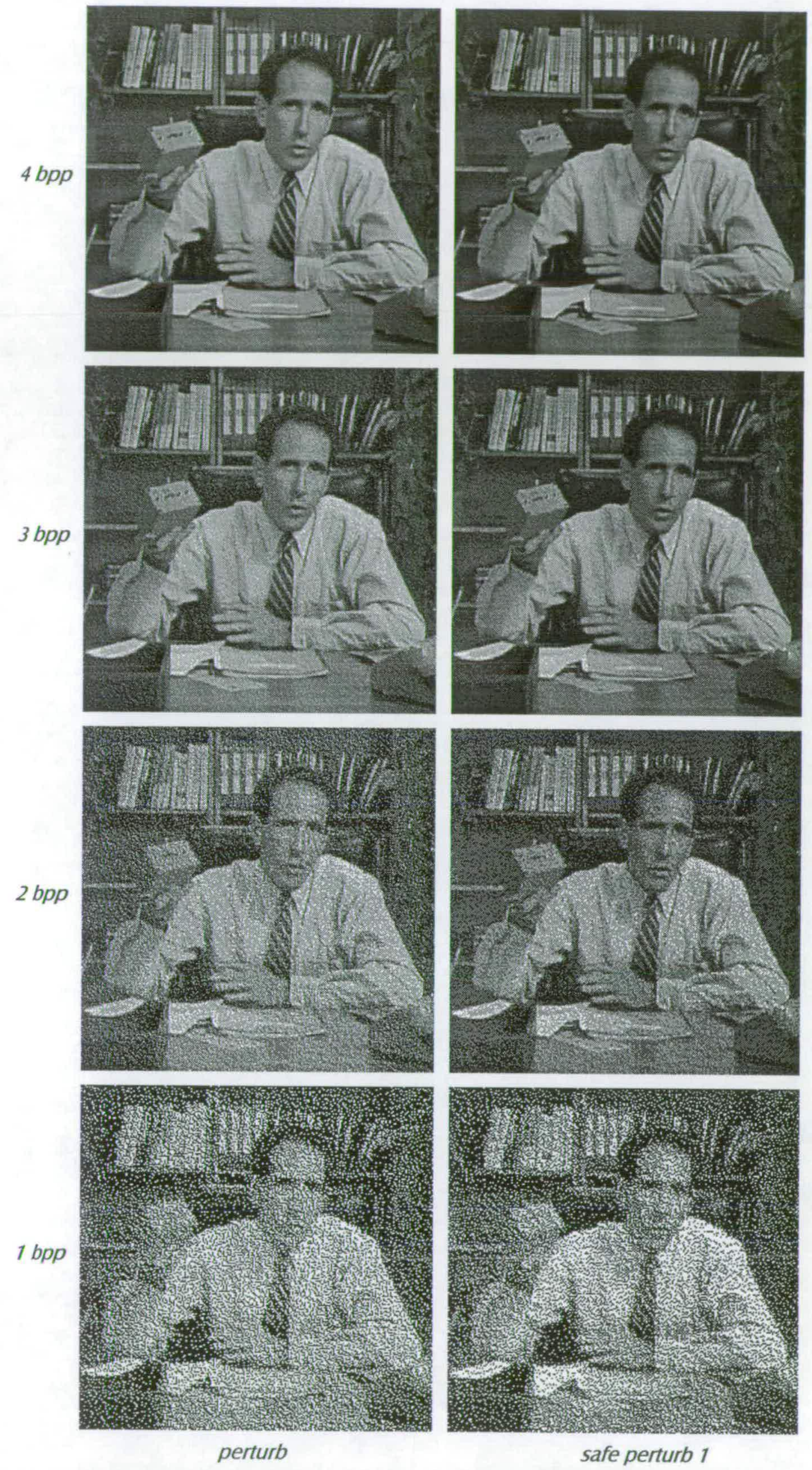


Figure 3.20 ***Comparison of the results of the 'perturb' and 'safe perturb 1' diffusion filters*** *(salesman frame 0).*

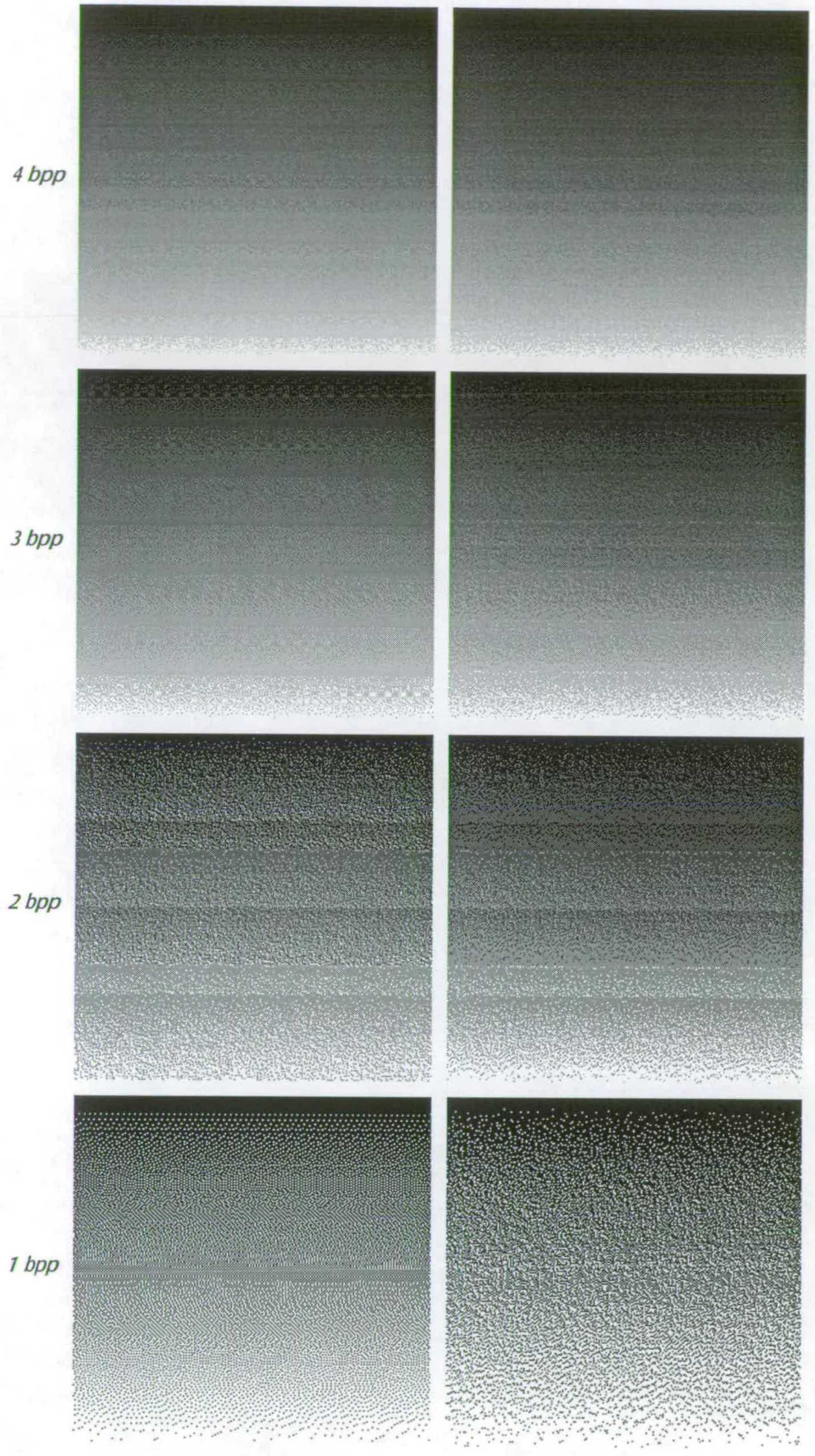


Figure 3.21 ***Comparison of the results of the 'perturb' and 'safe perturb 1' diffusion filters*** *(greyscale ramp).*

In both Figure 3.20 and Figure 3.21 the more limited nature of *safe perturb 1*'s random behaviour can be seen to reduce the visibility of its diffusion pattern in comparison to that of *perturb*. In particular, higher perceived contrast in the greyscale ramps diffused using *safe perturb 1* demonstrates a higher perceived image dynamic range due to the lower magnitude of its diffusion patterning.

The results of *safe perturb 1* are compared with those of *simple* in Figure 3.22. At 3 bpp and 4 bpp little difference between the output of the two algorithms can be spotted without close scrutiny. At 2 bpp a difference between the patterns produced is evident, but the degree to which the patterns are visible is similar. When compared to the comparison of the results of *simple* and *perturb* at 2 bpp (see Figure 3.18), this demonstrates some improvement gained in introducing the safe diffusion scheme of *safe perturb 1*.

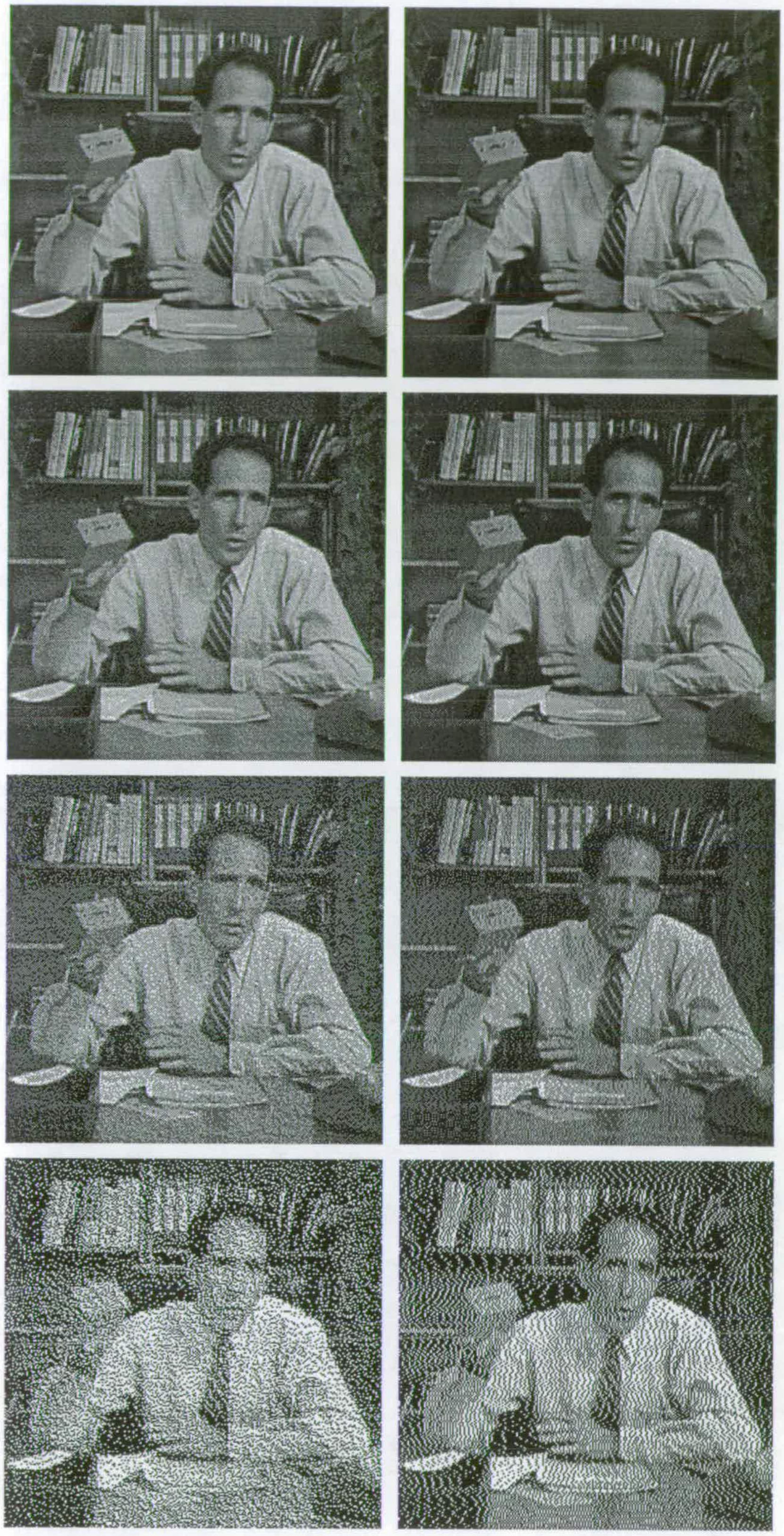


Figure 3.22 ***Comparison of the results of the 'safe perturb 1' and 'simple' diffusion filters*** *(salesman frame 0).*

The modifications to the *perturb* filter do not, however, solve all its problems. Now that the increase in magnitude has been eliminated another problem is evident. The problem is clearest in the greyscale ramps of Figure 3.21 where thin bright horizontal false contours can be seen. The contouring is different from that normally associated with quantisation. Contours are normally due to abrupt changes between areas of uniform tone where the value of the pixels in the original image cross the thresholds used in the quantiser. Those produced by *safe perturb 1* are due to an abundance of bright pixels in these areas.

Little of the contouring problem is evident in the diffused salesman images in Figure 3.20. This is due to the large amount of high frequency content in the original *salesman* image. It would, however, be apparent in the results of processing any images that contained areas of slow greyscale gradient that cross a quantiser threshold. Sections of difference images highlighting the diffusion patterns produced when reducing *lena* to 2 bpp are shown in Figure 3.23 demonstrating the appearance of the contours in a real image.

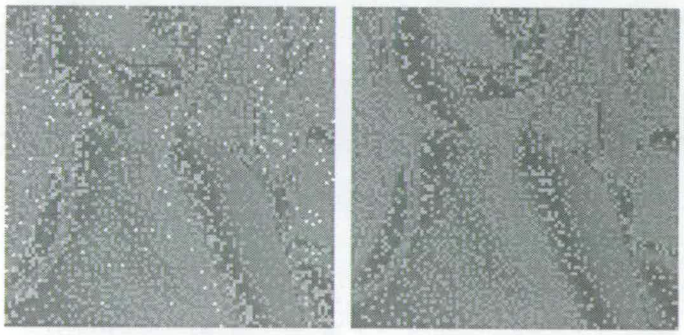

(a) perturb *(b) safe perturb 1*

*Figure 3.23 Sections of **diffusion patterns** produced when reducing lena to 2 bpp using (a) the 'perturb' filter, and (b) 'safe perturb 1'.*

Consideration of the action of *safe perturb 1* as it passed over areas where the original pixels were around one of the quantiser thresholds highlights a factor of the algorithm that contributes to the contours. If, during its first pass through the diffusion filter, an error is added to a pixel whose initial value was just below the threshold, that diffusion is nearly always significant. This can lead to a situation where many of the flags in an area are set. When the filter then passed over this area again on its next line of processing it is forced into diffusing errors downwards where, if the area is relatively flat, many of the pixels may again be close to the same threshold.

The 'safe perturb 2' filter

An attempt was made to alleviate the contouring problem of *safe perturb 1* resulting in a further filter named *safe perturb* 2. Instead of always diffusing downwards when the candidate to the right of the filter origin has already had a significant diffusion, this filter makes a random choice between diffusing the error to the pixel below, or discarding the error completely.

The modification has little impact on the hardware implementation of the filter, other than minimal changes to the error steering logic. It is unlikely that these changes would make it any more expensive to implement than *safe perturb 1*.

Results of diffusion using the *safe perturb* 2 filter on the vertical greyscale ramp are shown together with those produced using *safe perturb 1* in Figure 3.24.

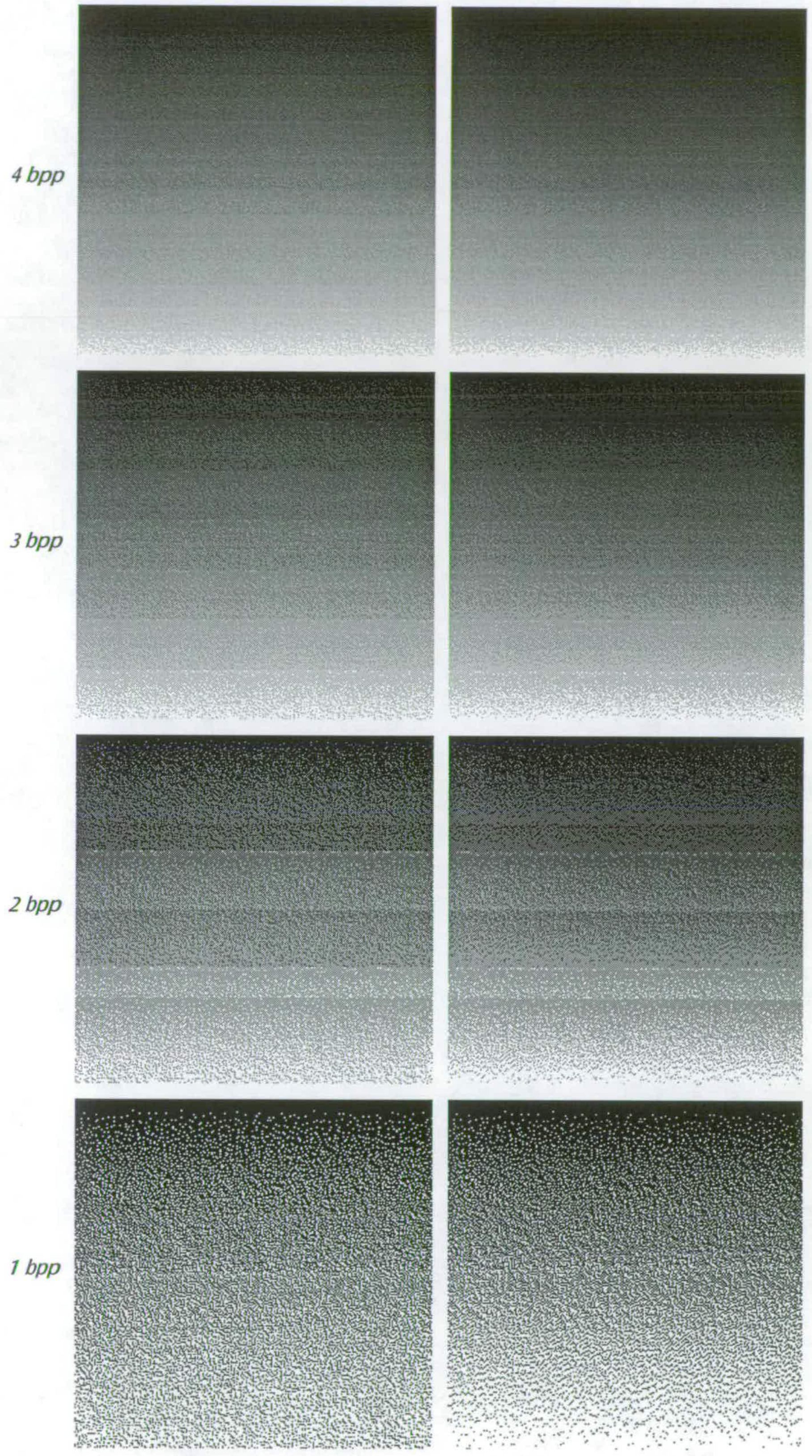


Figure 3.24 ***Comparison of the results of the 'safe perturb 2' and 'safe perturb 1' diffusion filters*** *(greyscale ramp).*

Compared to that of *safe perturb 1*, the results of *safe perturb* 2 show a only a small improvement in the false contours. A loss in overall image brightness (due to discarding the error energy) is, however, quite apparent in some areas (e.g. the bottom of the greyscale ramps).

Summary

This section has considered a selection of different filters: *simple, perturb, safe perturb 1* and *safe perturb 2*. These custom filters are considerably smaller than the multi-element filters (such as *floyd-steinberg*) normally used in image processing applications. The performance of the filters has been compared at a range of greyscale resolutions. All the filters perform at least as well as the raw truncator.

At the higher greyscale resolutions *simple* offers similar performance to that of *floyd-steinberg*. Yet it is much smaller and does not require the dividing up of the error term. These higher resolutions represent the likely pixel depth used in the video link application. Incorporating the *simple* filter into the application would allow video compression to be achieved at very low cost.

The *perturb* filter achieved poorer results than was expected. It produced a pattern with a particularly high magnitude, that proved to be highly distracting. As it is significantly more complicated to implement than *simple*, there is no advantage to be gained from its use.

Subsequent modifications to the *perturb* filter (selectively inhibiting the random element) resulted in the improved *safe perturb 1* filter. These changes reduce the magnitude of the noise pattern, leading to a performance comparable to that of *simple*. In areas of scenes exhibiting smooth changes in grey level *safe perturb 1* can offer superior performance. Movement of its noise-like pattern is often less distracting than that of *simple's* "wood-grain". This is especially true at low resolutions. Whether this improvement is perceived to be significant (and thus justify the extra cost involved in implementing *safe perturb 1*) is explored in the subjective testing of chapter five.

Safe perturb 2 was developed as an attempt to alleviate contouring effects sometimes present in the output from *safe perturb 1*. It achieved only limited success, reducing, but not eliminating the contours. More importantly, it had the unwanted effect of reducing overall image contrast (due to the discarding of error energy). Because of this added problem, it was not considered further.

Conclusions

The aim of this chapter was to design and evaluate the implementation of error diffusion algorithms for use in the coder section of the video link application.

Several features of the quantiser's behaviour and attributes of the diffusion filter have been identified as components that can be altered in order to balance various aspects of behaviour and the system cost.

Flexible quantiser architectures were considered, however insufficient benefit could be found to justify their high implementation cost. Instead, a simple binary truncating quantisation is employed. The main advantage of this technique is that it can be realised without any hardware.

Single element diffusion filters were used in the software simulation. These are much simpler than the multi-element filters generally employed in error diffusion. Their use in the radio video link application can be justified by the relatively high greyscale resolution and the fact that it processes moving images rather than stills.

At greyscale resolutions of 3 bpp and 4 bpp a purely deterministic filter (*simple*) has been shown to offer results similar to those from the much more complicated filter of Floyd & Steinberg. At lower resolutions, however, the limitations of the filter are apparent: a clear wood-grain like structure in the diffusion pattern.

Attempts were made to break up the deterministic qualities of *simple* by introducing a random element. Although the initial results were disappointingly noisy, selective inhibition of the random element alleviates this problem.

Reducing the resolution of the final image to 3 bpp, (the level required to achieve the desired 2:1 compression) results in a loss in image quality with all the filters. The two which incur the least penalty in quality are *simple* and *safe perturb 1*.

The software versions suggest that only a minimum of hardware is needed to implement the algorithms. This is tested via actual hardware implementation of these two algorithms in the next chapter.

Chapter five explores the validity of the hypotheses concerning the relative qualities of each diffusion filter's output developed in this chapter.

chapter four

Hardware Implementation

This chapter catalogues the hardware implementation of the two error diffusion algorithms identified in chapter three as being the most suitable for use in the radio video application. They are implemented in hardware in the form of a Field Programmable Gate-Array (FPGA). System level design is discussed, highlighting the architectural issues involved in designing a system in which the algorithm hardware could both be used and tested. Attention is drawn to the compromises that were made in using the chosen architecture (the imputer). This is followed by consideration of the FPGA design itself: explaining the design philosophy, the architecture of the FPGA design and details given of how the internal architecture is affected by the architecture of the overall system.

Introduction

Hardware implementations of error diffusions algorithms are pursued for two reasons: to allow the evaluation of their output at frame rates higher than is possible in the software simulations and to prove the hardware suitability of their architecture. The two algorithms implemented are *simple* and *safe perturb 1*. *Simple* represents a lowest-cost error diffusion algorithm. *Safe perturb 1* is slightly more sophisticated, but also significantly more expensive to implement.

In addition to the design of the algorithmic processing hardware itself, several other design tasks are posed by successfully implementing the algorithms in a usable and testable hardware form. In particular, decisions need to be made regarding the choice of implementation technology (ASIC, FPGA or discrete logic) and the test system architecture. The technology chosen is field programmable gate array (FPGA). The design of the processors is tailored to operate primarily as a slave-processor in a small image-processing computer architecture called the 'imputer' [VELLACOTT 94]. One of the processors (*safe perturb 1*) is designed to operate with a fixed output resolution of 4 bpp, the other (*simple*) can be operated with output resolutions of 1, 2, 3 or 4 bpp.

The reasons for selecting this particular solution and its subsequent design and testing are the subject of this chapter. System level issues are considered first, including the choice of implementation technology, the architecture used for the processor design and the overall test system architecture. This is followed by a presentation of details of the internal FPGA architecture and its low-level hardware design.

Implementation technology

Both the processors and control logic are purely digital thus there are several options open when considering how to actually produce the hardware implementation. These options include the full design of a digital ASIC, use of a mask-programmable gate-array part, the programming of an FPGA or constructing the entire circuit from discrete logic ICs (e.g. 74 series).

The complexity of the design (the final design is equivalent to approx. 7000 gates) means that implementation in discrete parts would be a lengthy task, difficult to revise or replicate. An integrated solution is therefore sought. A desire for rapid turn-around time and the insensitivity of the prototype system to unit cost leads to the decision to use an FPGA as opposed to fabricating an ASIC or designing a mask for a mask-programmed gate-array.

A combination of software tools (schematic capture, behavioural simulation, and design place & route tools) and the FPGA programmer offer a complete path from design entry to the production of working parts. With these, design revisions can be entered schematically, simulated and a new device produced within a single working day.

If the design is ever to go into volume manufacture, cost reduction of the design is possible through migration of the design from FPGA to mask programmed device.

Processor implementation

The two algorithms are implemented as individual processor modules within the FPGA. The design of the processor hardware is discussed in this and the following sections. Before considering the implementation of logic to perform each of the computational tasks required in the processors, the overall architecture of the processors is considered.

Processor architecture

The software written to perform error diffusion during the algorithm development stage describes the execution of the algorithms using purely serial computation on a Von Neumann machine [*GLASSER & DOBBERPUHL 85*]. In terms of the system architecture, these implementations are limited in speed. This is due to both the use of a single arithmetic logic unit (ALU) for all computation and the need to control the execution of the algorithm using a serial combination of fetch-execute cycles. Although this architecture limits performance, its generality is advantageous during the algorithmic development stage where it permits alteration of the algorithm through changes to the controlling software alone. No such generality is required in the implementation of specialised hardware to execute fixed algorithms. This permits the use of architectures more suited to the computation and data flow involved. In addition, the amount of hardware can be tailored according to design goals, trading off factors such as execution speed against power consumption and implementation cost.

The high level data flow required in the two error diffusion algorithms implemented is shown schematically in Figure 4.1 below.

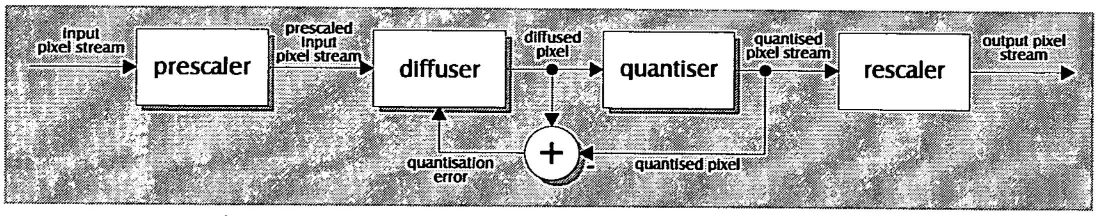


Figure 4.1 ***Data flow within the error diffusion algorithms.***

With the exception of the error feedback, the flow of data in the algorithm is essentially serial. The schematic describes the separate computational tasks of the algorithms and the order in which order they are performed. As such, they also describe the basic architecture of a dedicated hardware implementation of each algorithm. The functional blocks of the schematic are thus equivalent to computational logic and the connectivity shown between the blocks equivalent to signal routing.

In addition to the details of the computation involved in the algorithms many other factors influence the architecture of the processors. These factors include the design goals of the system and limitations of the implementation technology. The speed at which the processors have to operate turns out to be a key influence in the choice of architecture for the processors.

The video sensor intended for use with the hardware processors outputs video at 50 frames per second. Each frame contains 312 x 287 pixels. If the processors can be implemented so as to process a 256 x 256 pixel portion of every frame coming off the sensor they will be able to output a sequence far in excess of the minimum image specification outlined in chapter one. (Four times both the horizontal and vertical resolutions and five times the frame rate). Although this implies an output image sequence bandwidth much higher than intended for the video radio link (3 Mbps even at 1 bpp) it would also allow evaluation of the algorithms for use in other applications. This is the speed target used when evaluating the processor architecture and translates to a pixel rate of approx. 3 MHz (i.e. 300 ns per pixel, ignoring frame overheads).

The mainly serial data flow of the algorithms would permit the near exclusive use of combinatorial logic. The only exception is when implementing the memory required in the feedback paths. This would use the least hardware, but the complexity of the logic required is too great to meet the speed target. This can be shown in the following consideration of the computational tasks of a processor implementing the *simple* algorithm. To process each pixel it must first scale it down, then add a stored error to it, quantise the sum and rescale the result. Ignoring routing inefficiency, FPGA gate delays (approx. 20 ns) limits all critical paths in combinatorial logic to less than 15 gates (if the result is to settle within the specified 300 ns). As implementation of a single full adder itself has a critical path of 16 gates, the computation of the *simple* algorithm could not be performed within the 300 ns target. Purely combinatorial architectures are therefore ruled out.

Although a single purely combinatorial circuit cannot operate fast enough, more than one such circuit could be operated in parallel in order to achieve the required speed. However, the size (and thus cost) of the hardware grows linearly with the addition of each new unit. A more hardware-conservative approach is to break the combinatorial logic into sections that can be implemented in a shorter time, arranging the sections so that they can be used in parallel. Whilst running the hardware in parallel does not cut down the time required to process each pixel, it allows the processing of several pixels to be overlapped. This overlap yields an increase in pixel throughput.

The processing architecture described above is that of a *pipelined* processor. The whole processor is divided up into individual serial steps of processing. These are implemented by separate pieces of combinatorial hardware, connected via private buses through latches (see Figure 4.2 below). These *pipeline latches* control the flow of data along the pipeline and are typically clocked using two or more phases (preventing race hazards). Feedback paths can easily be implemented by creating loops in the pipeline.

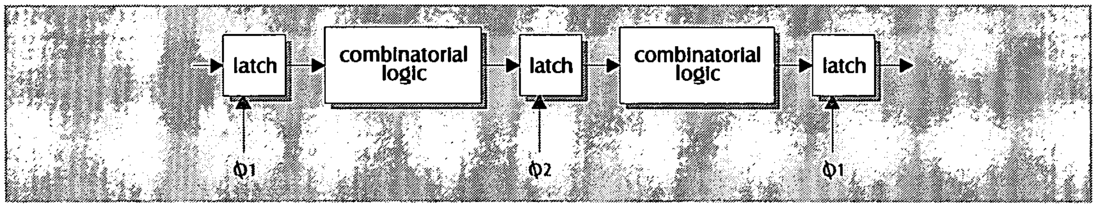


*Figure 4.2 The segmented **architecture of a pipelined processor**; sections of combinatorial logic are separated by pipelined latches.*

To ensure an efficient implementation using a pipelined processor a processing task must satisfy two conditions:

- it must be able to be divided into sub-tasks that take roughly the same amount of time to execute, and
- the degree to which the processing of data in the pipeline depends on the outcome of the processing of other data within the pipeline must be carefully limited.

The degree to which the first condition is satisfied governs how much of the processing can be overlapped. It thus determines to what extent the architecture can speed up the repeated execution of the task. The second criterion relates to how efficiently the problem can be implemented. If there are dependencies in the processing (the result of a calculation from one part of the pipeline affecting how processing earlier on in the pipeline should be performed) the pipeline may have to be branched early on. The sections should run in parallel until it can be determined which branch should feed data into the output path of the processor (the other results being discarded). If there are many such dependencies, implementation becomes expensive as the amount of hardware grows exponentially with the number of processing options.

The diffusion algorithms satisfy both conditions. It will be shown later that they can be broken down into chunks even enough that the processing can be sufficiently overlapped (achieving the 300 ns pixel cycle). The fixed nature of the algorithms satisfies the second condition (processing dependencies are limited such that any intermediate data need only be stored for a short time).

There is a compromise in the use of a simple continuously operated pipeline architecture in conjunction with data in a conventional raster. As a raster stream of image data contains spatial discontinuities a pipelined diffuser will spread error energy across these line and field boundaries in the image. Fortunately, the extent of this problem in the case of the *simple* and *safe perturb 1* algorithms is limited by their simplicity. The first pixel of each line in the output of the *simple* processor will be influenced by the last pixel of the line before. In the output of the *safe perturb 1* processor the problem will extend to the top line of each field.

No distracting edge contamination of this sort was noticed in the still images produced during software simulation. However, the speed of the hardware processors (which it is hoped will mask the noise element of the diffusion) may make any correlation between opposite edges more apparent (and thus objectionable). Interestingly, images produced during the simulations demonstrate how edge contamination can help to break up the particularly deterministic patterning that would otherwise be found at the beginning of lines in very flat areas of an image.

Processor logic design

Designing the processor hardware requires identification of the various processing steps of the algorithm, designing logic to implement these and partitioning the design into parts that will form the individual sections of the processor pipeline. The design is partitioned so that all pipeline sections have propagation delays less than 150 ns. This permits using both edges of a 300 ns period, 50% duty cycle, square wave to clock the pipeline (giving a pixel rate through the processor equivalent to 300 ns - satisfying the speed requirement outlined earlier).

The design of the two processors is discussed below. Full schematics of the processors' implementations can be found in appendix two. Before considering the complete processor circuits, implementation of their constituent parts (the prescalers, diffusers, quantisers and rescalers) is first explained.

Prescalers

The prescalers are required to ensure that the pixel stream that enters the diffuser is always a certain level lower than the maximum size possible (given the number of bits used to represent it). The amount by which it must be lower is dependent on the maximum error that can be fed back from the quantiser and is therefore determined by the output resolution of the processor.

Assuming that the input data is full scale and that a linear function is to be applied the action required of the prescaler logic is to linearly reduce the input pixel signal amplitude. Although this is essentially a division, the divisor is fixed for a given system output resolution and all output resolutions require divisors of the form $2^n/(2^n-1)$ which can be rearranged as a binary shift-subtract operation. In dedicated hardware the binary shift can be implemented via signal routing and a subtracter can be implemented using an adder and 2's complement arithmetic. This arrangement is shown in Figure 4.3 below.

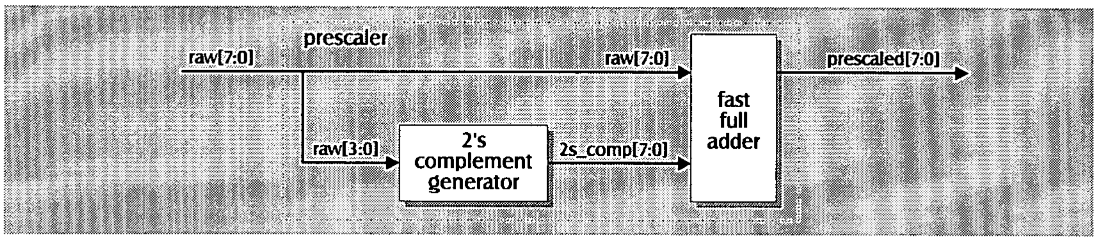


Figure 4.3 ***Hardware Implementation of the prescaler.*** *The bus labels indicate their width. (The bus width shown at the input to the 2s complement generator leads to the correct scaling for a processor with a 4-bit output resolution.)*

The scaling implemented by this circuit is not perfect as the portion of the input pixel discarded during the binary shift leads to an error in the division (unless all discarded bits are zero). This non-linearity could be avoided by not discarding the bits, using a wider adder and increasing the number of bits used to represent the prescaled output. Such a level of accuracy is not deemed important enough to warrant this increased expense in the implementation of the prescaler and the consequence it would have on the expense of the remainder of the processors.

Diffusers

The action required of the diffusers is simply to add an error term to a pixel. 8-bit full adders are used for this task.

Quantisers

Possible quantiser behaviours were considered in chapter three. It was concluded that the behaviour of a truncation operation is sufficiently sophisticated for the needs of the video link. This solution is purely combinatorial and in fact can be implemented using signal routing alone.

Unlike some of the more sophisticated quantisers considered in chapter three (which would have dictated the used of temporary latching of results and/or the dynamic presentation of different quantisation thresholds) implementation of the truncating quantiser does not impact on the overall processor architecture.

With this quantiser, calculation of the quantisation error term is also trivial. This too can be implemented with routing alone.

Rescalers

Used in a radio video link, the data from the output of the quantiser would be sent via the radio transmitter to the receiver. The data would then be interpreted by the receiver for display. Before display it is likely that the data would have to be translated (if only to make best use of the dynamic range). For example, in a system using a 2 bpp compressed data stream and an 8 bpp greyscale display, 11_2 the brightest code of the data stream would be translated to $1111\ 1111_2$ (or 255_{10}). For systems with integer bpp compressed and display pixels, all such translations can be made by using the bits of the compressed pixel as the most significant bits of the display pixel, then filling out the lower bits by replicating the compressed pixel. In the same example as above 10_2 becomes $1010\ 1010_2$, 01_2 becomes $0101\ 0101_2$ and 00_2 $0000\ 0000_2$. This conversion is slightly non-linear where the width of the compressed pixel is not a factor of the display pixel. This is the action of the rescaler.

To reduce the number of processing steps in the test system the rescalers are implemented as though they are part of the diffusion processors. This means images can be processed using a grab-process cycle rather than grab-process-rescale cycle. If the compressed sequence is required from these processors, the relevant number of most-significant bits of the rescaled output can be used and the others simply discarded.

The rescalers can be implemented using signal routing. For the *simple* processor this is complicated slightly as its output resolution (on which the nature of the rescaling operation is dependent) is variable. The variable behaviour can be implemented by multiplexing the various possible results.

The 'simple' processor

The *simple* algorithm adds the error from the current quantisation onto the next pixel in the raster scan. It consists simply of a pre-scaler, a quantiser, a diffuser and a rescaler. These are connected together as shown in Figure 4.4. The feedback path goes directly from the quantiser output to the input of the diffuser (through two latches). The positions of the latches in Figure 4.4 show how the processing logic is divided into sections of pipeline.

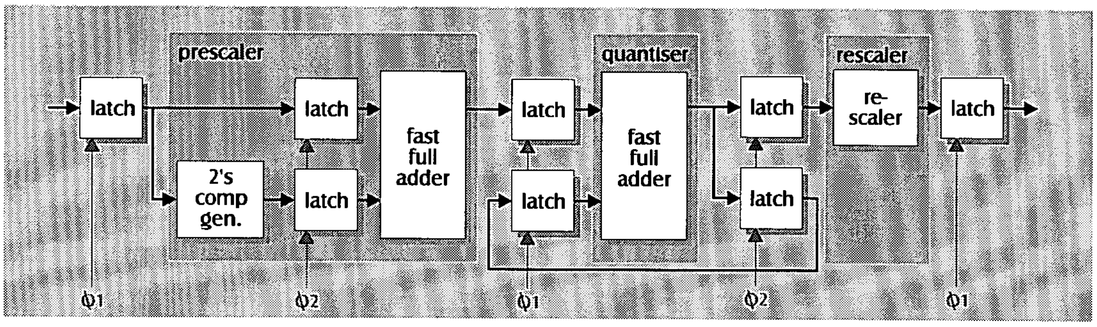


Figure 4.4 ***The 'simple' error diffusion processor** pipeline. The constituent parts of the processor are shown, separated by the pipeline latches. The feedback path followed by the quantisation errors can be seen below the diffusing quantiser.*

The 'safe perturb 1' processor

The *safe perturb* 1 processor is similar to *simple* except that instead of always adding the quantisation error to the next pixel in the raster scan there is a chance that it will instead add it to the pixel one line later in the raster scan (i.e. the pixel below rather than the pixel to the right). The combination of a boolean flag and a bit from a pseudo-random bit sequence (PRBS) is used to determine in which direction the current error should be spread. The flag comes from the first adder of the diffuser and is set if the pixel exiting it has already received a 'significant diffusion' (in this case, the flag ensures that it doesn't receive another diffusion at the second adder).

The line long separation between the adders of the diffuser corresponds to a section of pipeline 9-bits wide and 512 latches long (8-bits for the pixel and 1 bit for the flag). Unfortunately, implementation of this shift register constitutes more gates than are contained in the largest FPGA considered for the design. Instead of implementing this section internally the pipeline has to be broken, the two ends brought out of the FPGA and connected to either side of a FIFO RAM device. Simultaneously reading from the FIFO whilst writing to it will simulate a pipeline section. The length of the section is controlled by loading the FIFO to a certain 'depth' with data before operating the pipeline. (This FIFO setup stage is referred to as 'pre-loading' in the remainder of the text.)

The PRBS generator is constructed using a classic shift register with exclusive-or feedback [PRESS 92]. It creates a sequence that repeats every $(2^{18}-1)$ bits. In order to minimise the chance of visible patterning it is clocked asynchronously to the reset of the *safe perturb 1* processor.

A schematic of the hardware implementation of the *safe perturb 1* processor is shown in Figure 4.5. There are three notable differences between it and the *simple* processor: the addition of the second adder to the diffuser, the break in the pipeline where the FIFO device fits in, and the more complicated error feedback path.

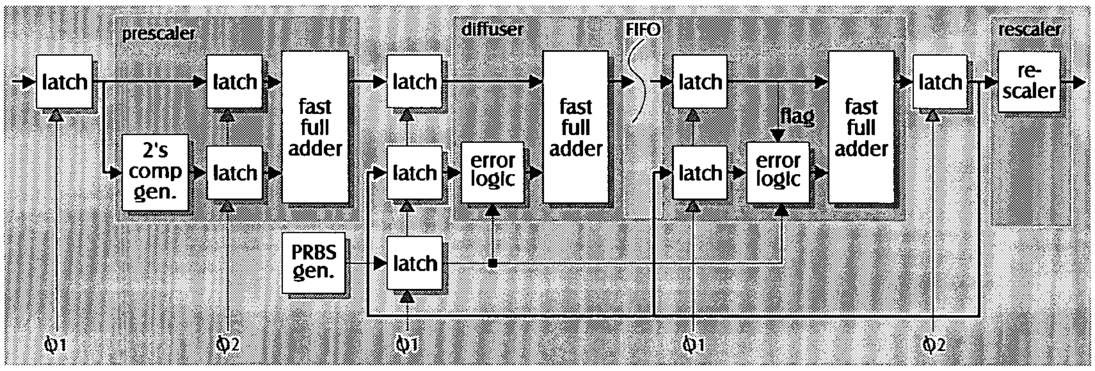


Figure 4.5 ***The 'safe perturb 1' error diffusion processor pipeline.*** *The constituent parts of the processor are shown, separated by the pipeline latches.*

Test system architecture

Hardware implementation of the algorithmic processors themselves is of little use if there is not a system in which they can operate and be tested. The objectives of the hardware implementation are to prove the processor design and to allow the processing of images at high frame rates. The test system must, therefore, allow both careful monitoring of the processors' operation and allow the processors to manipulate sequences of video frames at significant speed.

In order to verify that the diffusion processors work correctly (and, therefore, that the hardware implementation is valid), it is necessary that the input and output data of the processors can be carefully monitored. Analysis of the input and output data and comparison with that expected, can quickly determine whether the algorithms are being executed properly. The ability to load specific data into the system permits two useful abilities: the use of *synthetic* input data (thus giving carefully controlled test conditions that simplify the testing and any necessary debugging) and the ability to make direct comparison between the hardware and software implementations by using the same input data. The ability to store input and output inside the test system for later retrieval is also of value as this allows analysis to be performed off-line. Another factor to consider is that the operation of the processors during the testing should be as close to their intended operation in the radio video link as possible. The less this is true the more artificial the tests and thus the less value in the results.

One option for the test system is to implement the entire radio video link system. Its architecture, which is primarily serial, is shown again in Figure 4.6. Data flows continuously along dedicated paths from the sensor, through the processing and radio hardware, ultimately reaching the display.

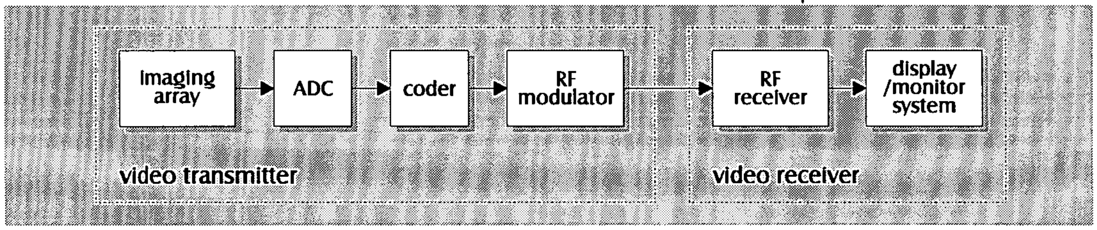


Figure 4.6 ***The proposed radio video link architecture.*** *The diffusion processor would form part (or all) of the coder block of the transmitter.*

By definition, this system would satisfy the requirement to allow operation of the processors in an environment similar to that of the radio video link. In this pixel stream architecture the coding processors operate on live image input 'on-the-fly' and the rest of the system processes and transmits the data with minimal storage. Unfortunately, the continuous processing, lack of memory and the use of separate dedicated private data buses, do not lend the architecture to simple testing and debugging.

The radio video link architecture doesn't offer the degree of testability required thus another architecture must be implemented for the test system. The monitoring and flexibility requirements could be met through extension of the radio video link architecture to include local storage, more general buses and allow more high level control. In addition to offering a test system, this would also offer an example of a radio video link system. It would, however, be costly in terms of implementation time and be a poor example of the intended radio video link - the features that make the system attractive as a cheap low cost radio video link would be compromised.

The fact that the architecture of the final application is not suitable for use in testing the processors has an advantage in that it opens up the possibility of adopting the architecture of an existing test system. This offers savings both in design time and technical risk. One such system is the 'imputer'. This is a miniature image processing system based around an 8032 microcontroller, a local CMOS video sensor (an ASIS-1011), frame grabbing hardware and static RAM [VELLACOTT 94]. The architecture of the imputer is shown schematically in Figure 4.7.

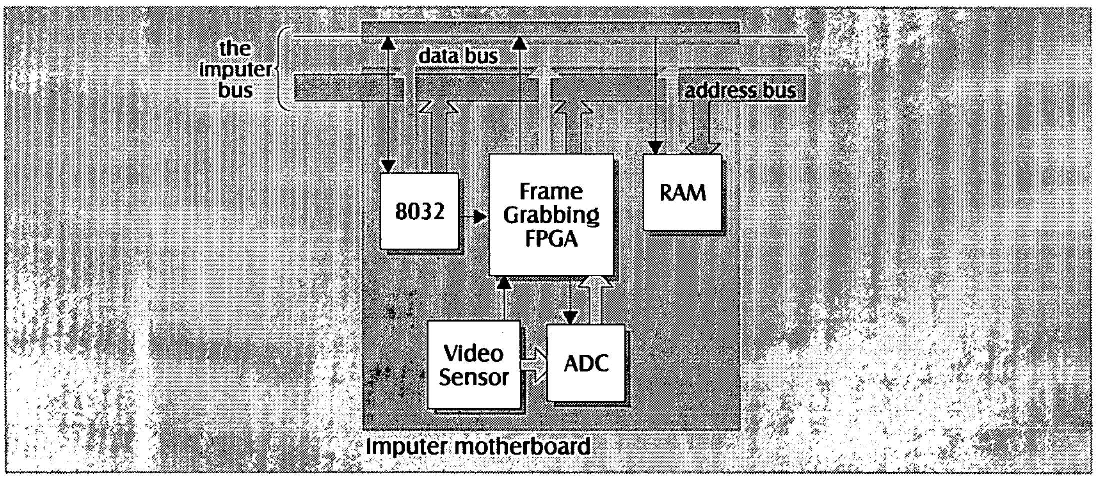


Figure 4.7 ***The imputer architecture***

The provision of a bus with access to most of the system features and the ability to quickly re-program the behaviour of the microcontroller (via firmware) make the inclusion of a co-processor in an imputer system simple. The two diffusion processors could be tested in the imputer architecture by designing them embedded within a hardware co-processor that would act as a slave to the imputer's microcontroller.

Unfortunately there are features of the imputer's architecture that compromise aspects of the diffusion processor operation. Use of the imputer grab logic implies the use of a common data bus for frame grabbing and diffusion processing. This prevents the two tasks from being performed in parallel, thus having a direct impact on image latency and system frame rate. The image latency incurred will be at least 18 ms, as the first pixel cannot even enter the diffusion processor until the end of the grab of the entire frame. Using the serial grab-process cycle necessary, the system would only be able to achieve the full 50 fps frame rate of the sensor if the diffusion processing can be performed in the 3.3 ms of the video frame period not used by the grab hardware. Even ignoring processor set-up, this translates to a processing time of under 50 ns per pixel, which is impossible using the 70 ns imputer RAM as each pixel cycle includes one read and one write to RAM. Grabbing every second frame of data allows at least 20 ms for processing each frame, i.e. a more reasonable 300 ns per pixel (approx.) while achieving a system frame rate of 25 fps and minimum image latency of 40 ms.

An advantage of using a serial grab-diffusion cycle necessary in the imputer architecture is that the pixel-level timings of the two functions can be independent. This is an attractive feature during the testing of the FPGA as it can be clocked at a much slower speed than the sensor without affecting the operation of the rest of the system.

The imputer co-processor architecture frees the algorithmic processors from some of the constraints imposed by the 'on-the fly' pixel-stream processing architecture in which they would normally operate. These include the necessity to process data in raster scan order (the imputer architecture permits the random access of data). However, retaining the restriction of only accessing data in raster scan order keeps the implementation of the computational and control hardware of the algorithmic processors closer to that of a true pixel stream architecture.

Summary

Embedding the diffusion processors in an FPGA that can be used as a co-processor in an imputer system offers the required flexibility of input and output to satisfy the testing requirements laid out above. There are, however, restrictions imposed by the implementation - notably the image latency and upper limit on processing speed imposed by the grab-process cycle.

In order to implement the processors as a part of an imputer co-processor whilst retaining the possibility of running the processors in an architecture that would not suffer from these restrictions, they are implemented using an internal FPGA architecture and control system. (This can easily be expanded to control a stand-alone hardware system.) The architectures of the slave and stand-alone systems are expanded upon below.

The slave FPGA system architecture

As a slave co-processor the 'PINK2 Diffusion FPGA' fits into the imputer architecture as shown in Figure 4.8. The intention is to implement a control system that offers the imputer microcontroller (the 8032) the ability to process complete single frames of video using either of the diffusion processors. The FPGA is required to read the source image data and write resultant image data to the imputer RAM. Test images can be either be generated with code executed by the 8032 or downloaded from a host PC. Live video images can be captured from the local video sensor by the 8032 using its local ADC and 'grab' logic. Live display of the output images can be arranged through use of an imputer 'video generator' card (also shown in Figure 4.8).

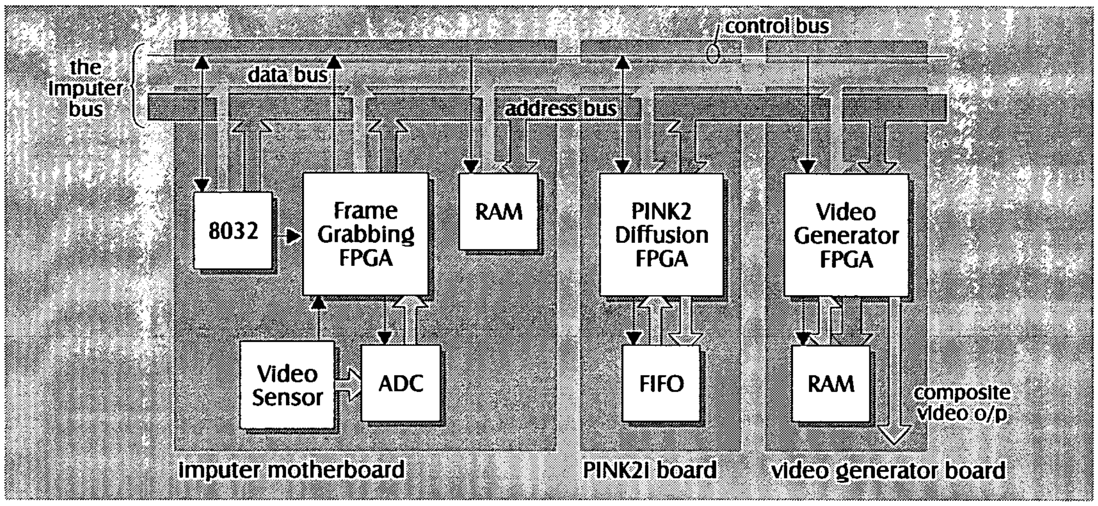


Figure 4.8 ***The imputer-slave PINK2 FPGA system architecture.*** *This shows large the functional blocks of the system, how they are distributed between imputer PCBs and their common connection over the imputer bus.*

The stand alone FPGA system architecture

To demonstrate the abilities of the diffusion processors without suffering the limits on processing speed imposed through the use of the shared data bus in the imputer architecture, the processors and FPGA control system are designed to operate in a stand-alone architecture.

Figure 4.9 shows the constituent parts of a stand alone system. The FPGA is required to control an ADC to provide a constant source of digital video and clock a digital-to-analogue converter (DAC) to produce the active portion of the video output. The pipelines of the diffusion processors would be fed directly with data from the ADC and their output would in turn feed the DAC. An external sync separator IC could be used to multiplex between the DAC output and a voltage reference to produce the sync component of the composite video.

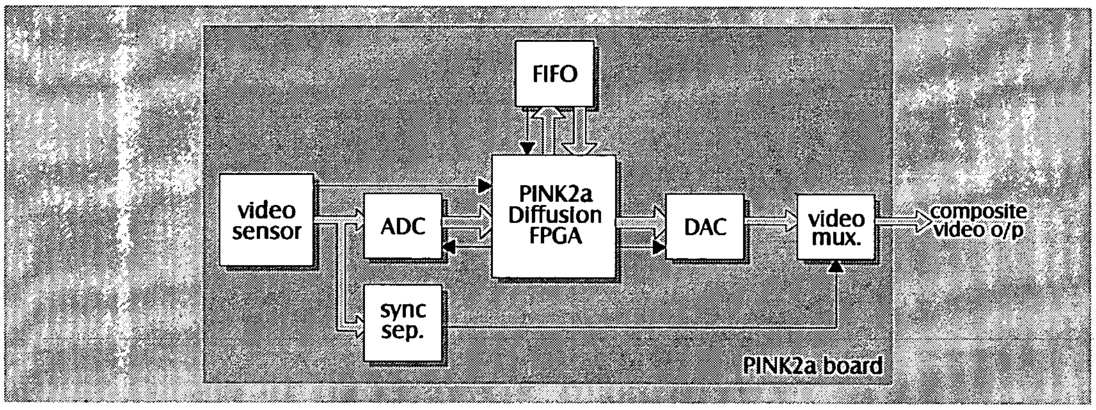


Figure 4.9 ***The stand-alone FPGA system architecture.***

The only external control required over the FPGA in this architecture is to determine which of the processors it uses and at what resolution. Simple control logic and some user switches could be used to achieve this.

Such a system would be capable of performing continuous (and parallel) image capture, processing and display, at 50 fps and with very low image latency.

Although the parts of the FPGA control system common to both imputer and stand-alone architectures were designed to be usable in both, a stand alone system has not yet been realised.

FPGA design philosophy

The design and test philosophy adopted for the FPGA is one of strict hierarchical design and proof of design validity through rigorous simulation. The tools used (Viewlogic behavioural simulation tools and Actel FPGA layout tools) have proved reliable in the past, accurately predicting real-life FPGA performance. Considerable time is spent simulating the circuit behaviour before devices are actually programmed.

The circuit is designed using an iterative design-simulate-evaluate approach. The behaviour required of the entire FPGA is broken down into logical tasks and an overall architecture of large functional blocks devised that will implement this behaviour. The functional blocks are sub-divided until each is relatively small. Finally, logic is designed to realise the behaviour required of each block.

The behaviour of each block of logic is first simulated in isolation from the remainder of the design. Any weaknesses revealed are addressed through the revision of the logic design and the simulation-verification process repeated. Once a number of related logic blocks have been successfully tested in isolation they are combined into larger models and tested together. Not only does this re-testing serve to 're-prove' the design of the individual blocks, it also reveals any errors in the lower level specifications made during the hierarchical partitioning process. Any problems found during simulation require the redesign of the sub-modules, isolated re-testing, then repetition of the larger tests. This process is iterated until a model of the entire FPGA is assembled.

During the final stages of simulation parts of the imputer are modelled. This allows the simulation of both the interaction between microcontroller and FPGA and secondly, the processing of real image data stored in a RAM model. The only major part of the system not modelled is the FIFO. This means that although the individual components of the *safe perturb 1* processor are tested, its entire pipeline cannot be simulated.

Most simulations are conducted using net lists with ideal propagation delays. Once the design has been placed and routed 'worst-case' delay information is extracted and back-annotated onto the simulation net list so that 'worst case' device behaviour can be modelled. It is anticipated that the glitch-free design style used and successfully simulated operation (with both ideal and worst-case delays) will guarantee successful operation of a real-life device.

Internal FPGA architecture

Two tasks are required of the FPGA: low-level pixel processing and interaction with the remainder of the system. The highest level partition in the architecture of the FPGA logic reflects this division of task, as shown in Figure 4.10.

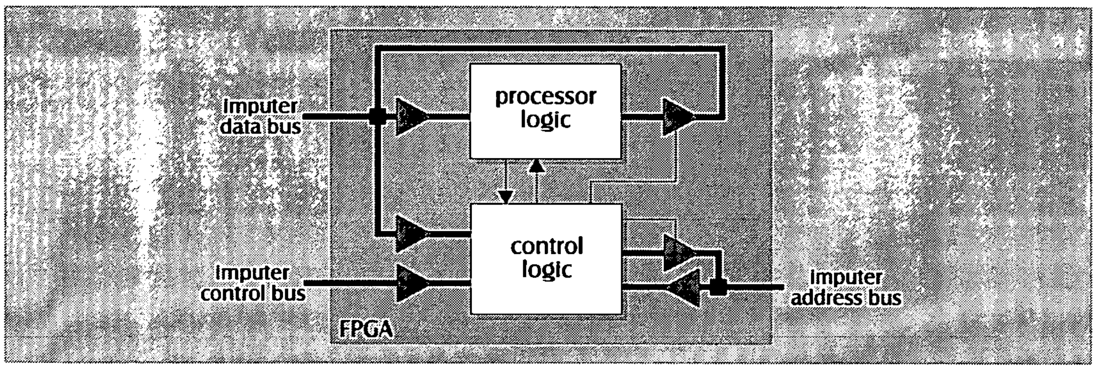


Figure 4.10 ***High level internal FPGA architecture***

Both the diffusion processors are contained within the 'processor logic' block. The control logic supervises the operation of the processors within the rest of the imputer system.

The design of the two diffusion processors is discussed at the beginning of this chapter. The remainder of the FPGA logic design is described below. The arrangement of the processors is described and their control needs identified. This allows implementation of a complete internal control system. The low-level design of the entire PINK2 FPGA is given in the design schematics in appendix two.

The processors

In addition to the implementation of the two error diffusion processors, two further processors (*'raw'* and *'truncate'*) are implemented. These extra processors are included to facilitate the evaluation of the diffusion processors. The *raw* processor allows data to be sent to the imputer systems display without any processing but at the same rate as is output by the diffusion processors. *Truncate* implements only a binary quantiser.

The four pipelined processors are arranged in parallel within the FPGA . This enables them to share the same input data and a common path for output data. The arrangement of the processors is shown in Figure 4.11.

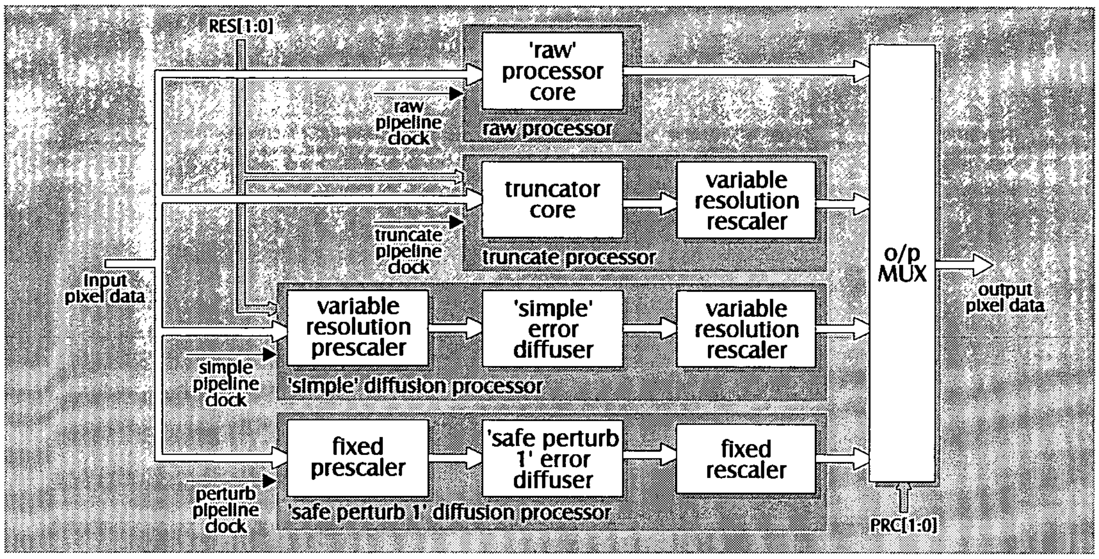


Figure 4.11 ***Data flow through the pipelined image processors.*** *This figure shows how processors are arranged within the FPGA in terms of data flow and outlines the large functional blocks within them. The processors can be seen to sit in parallel, sharing the same input data and outputting data to a common latch.*

Resolution

During the algorithmic research the software algorithms were written to output data at resolutions between 1 and 6 bits per pixel. This degree of flexibility is not echoed in the hardware implementations both because of the difficulty of its implementation and because resolutions higher than 4 bpp offer a low compression ratio (1.6:1 and less).

Two of the processors (*simple* and *truncate*) are implemented with limited variable output resolution (1 to 4 bpp). In order to keep the overall size of the design small, the second error diffusion processor (*safe perturb 1*) is implemented with a fixed resolution of 4 bpp. Software simulations show that this resolution would be more than adequate to achieve good quality images.

To make the two processors variable in resolution their two's complement generators, quantisers and rescalers all need to be programmable. A 2-bit 'resolution' bus is used to communicate the resolution currently desired of the processors to the programmable logic blocks. Most of this programmable behaviour is achieved using multiplexers to make signal routing dependent on the resolution bus.

The control logic

Operation within the imputer architecture requires the control system to interface with the microcontroller, operate the processor pipelines and access the imputer RAM for input and output. Stand alone operation requires synchronisation with the video sensor and control over ADC and DAC devices.

An architecture of gray code state machines has been devised that delivers all these control and interface functions. The modular architecture allows re-use of logic blocks whose functions are common to both imputer and standalone modes of operation, while maintaining isolation between a mode of operation and logic blocks that are not required for it. This isolation means that operation in the imputer architecture is not reliant on the parts of the architecture specific to autonomous operation. Therefore they do not need to be designed before the rest of the FPGA is tested in the imputer system.

A schematic representation of the all the control logic modules is given in Figure 4.12. It includes all the state machines, the counter that generates raster address sequences and all inter-module connections.

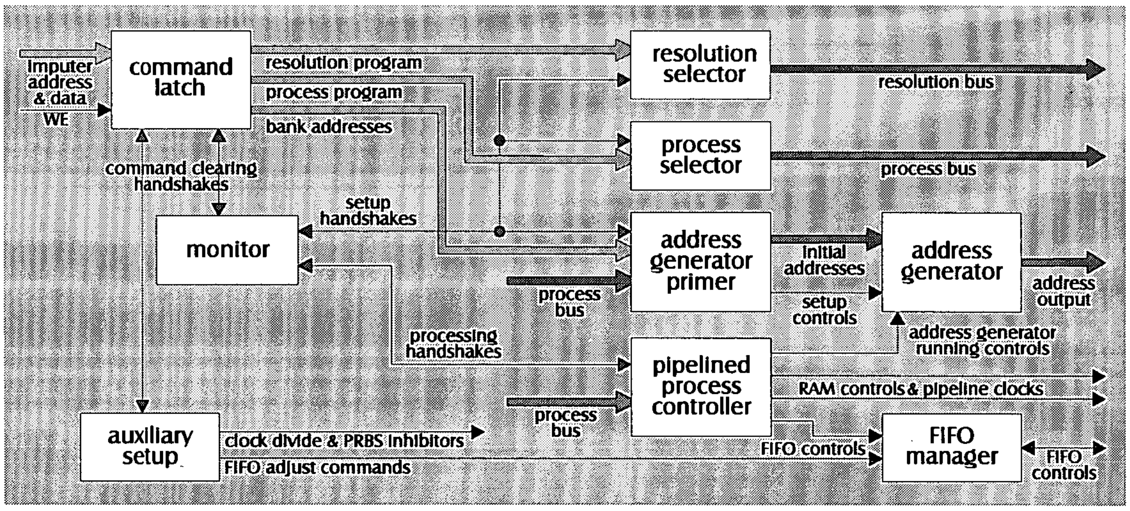


Figure 4.12 ***Schematic representation of the FPGA control logic and address generator.*** *These state machines make up the control system that runs the pipelined processors and interfaces them with the imputer architecture. The modules are arranged within the schematic in approximate order of hierarchy from left to right.*

Overall control

Which mode of operation the FPGA adopts (imputer-slave or stand-alone device) is determined by the sense of an input shortly after power-up. Immediately after power-up, the FPGA is under the control of a state machine called the *monitor*. If the mode input is set to 'autonomous' then the monitor hands over control to the *autonomous controller* (which starts running a processor). Otherwise it retains control and enters an endless loop of waiting for processing commands from the imputer and acting on them.

Control over the internal processors and external devices in autonomous mode is relatively simple as the processing is performed continuously. Other than staying in sync with the video sensor, the processing of each pixel is identical. In imputer-slave mode control is complicated by three factors: the need to stop processing after a frame of pixel iterations, having to generate the address rasters and having to set up all process variables before processing. (In autonomous mode the process and resolution can be changed asynchronously from the processing.)

A simple processing cycle is followed in the imputer architecture. Valid commands from the imputer are decoded and stored in the *command latch*. When the monitor notices that a process command has been received it leaves its idle state, refreshes the *process* and *resolution controllers*, oversees the priming of the *address generator* with initial read and write addresses and then allows the *process controller* to run one of the pipelined processors for a full 256x256 frame of image data. The monitor then stops all activity and returns to its idle state.

In keeping with the nature of the state machine architecture a modular approach is taken to the control protocol used between the modules. At the start of each monitor-led processing cycle, the monitor co-ordinates all the processing and lower-level control logic, in preparation for processing. To keep independence between the design of the individual modules a handshaking protocol is employed that relies on signal states rather than absolute edge timing.

The main requirement of the processing preparation stage is that all the logic that must be prepared is ready before processing begins. When it is time for the pre-processing setup, the monitor raises the PREPARE line and keeps it raised until all relevant low-level machines have answered. The low-level machines answer by raising their _READY outputs, they do this immediately after they notice the raised PREPARE flag and only drop them again once PREPARE has been dropped by the monitor and they have finished their processing cycle. This handshake ensures that all the relevant low-level machines see the PREPARE flag and that the monitor waits until all of them are ready before allowing the process controller to start.

Interdependencies between the low-level machines can be allowed for by making the operation of the dependent machine itself dependent on the _READY flag of the other. This system is used in the FPGA to make the address primer wait until the process selector has updated the current process before priming the address generator with the start address (which is process-dependent).

This handshaking protocol allows a modular approach to be taken to the FPGA control system. As long as the protocol is observed additions or modifications to any of the control system can be made without altering the rest of the system.

Low-level processor control

The operation of the pixel processors is controlled by three of the state machines: the *process selector,* the *resolution selector* and the *pipelined processor controller.* Synchronised control over the address generator and clocking of the processors is achieved by the process controller. Which processor and at what resolution it operates at are determined by the outputs of the process and resolution selectors. To offer compatibility with the imputer and autonomous architectures the selectors are implemented as programmable state machines. They have four stable states and can be forced into any of these using the programming inputs. A simple logic input can be used to cycle around the four stable states. This allows simpler control in an autonomous application.

Imputer RAM address generation

The two binary sequences required to address imputer memory in order to access the image data in raster order are generated by the *address generator.* The address generator is initialised by the *address generator primer* and clocked during processing by the *pipelined process controller*.

Auxiliary parameter setup

Control over a number of auxiliary parameters is available to the imputer in imputer-slave mode via the *auxiliary setup* machine. These parameters include internal clock frequency division, inhibition of the PRBS generator, re-preloading of the FIFO and altering the length of the shift register that the FIFO simulates.

FIFO management

The *FIFO manager* is a simple state machine the preloading and clocking necessary to simulate the long section of *safe perturb 1* pipeline using the external FIFO.

Simulation results

Comments on the tools

The logic simulation and FPGA layout tools prove powerful in allowing the full development of a complete FPGA model, including post-layout worst-case timing delay information. Unfortunately some minor aspects of the tools are problematic.

Extracted layout timing limit information

The timing extraction routines of the layout tool (ALS) cannot provide useful maximum critical path speed information when all the sequential logic is not synchronous. This is the case in parts of the FPGA (particularly around the imputer inputs where, instead of the system clock, an imputer signal is used to clock many latches). This leads to flagging of hundreds of irrelevant asynchronous hazards. The sheer volume of these, combined with the obscure node naming conventions, makes interpretation of the critical path timing analysis information practically impossible.

Version control

Little control over schematic and layout files is evident. Without careful control over file names and locations it is easy to erroneously back-annotate a schematic with the extracted timing information from a layout generated from a different schematic.

Problems exposed by back-annotation

Simulation using schematics back-annotated with the worst-case delay information provided by the layout tools exposed only one problem in the design. An oversight in RAM write cycle timing led to the read/write strobe toggling at the same time as the address is latched on the FPGA outputs. During the ideal simulations this satisfies the RAM models timing requirements as both transitions occur simultaneously. With the addition of realistic delays the address bus takes time to settle violating RAM setup-and-hold requirements. This can, however, be remedied by changing the phase of clock used for a single set of latches.

The lack of problems experienced in the move from ideal to worst-case delays demonstrates that the design methods used are robust The heavily pipe-lined nature of the processor logic protects it well against potential timing problems. The use of gray-code state machines and the handshaking protocols of the control logic prevent glitches and race-hazards.

Hardware tests

Once the FPGA devices were programmed with the PINK design, a series of tests followed to prove the design. The tests included general operation of the FPGA within the imputer architecture, exercise of the internal control system and, most importantly, the operation of the diffusion processors.

The hardware set-up used for the tests, the test procedures and the results are discussed below.

Test system

The full test setup comprises an imputer system (including PINK FPGA), a small video monitor and a host PC. This system is shown in Figure 4.13.

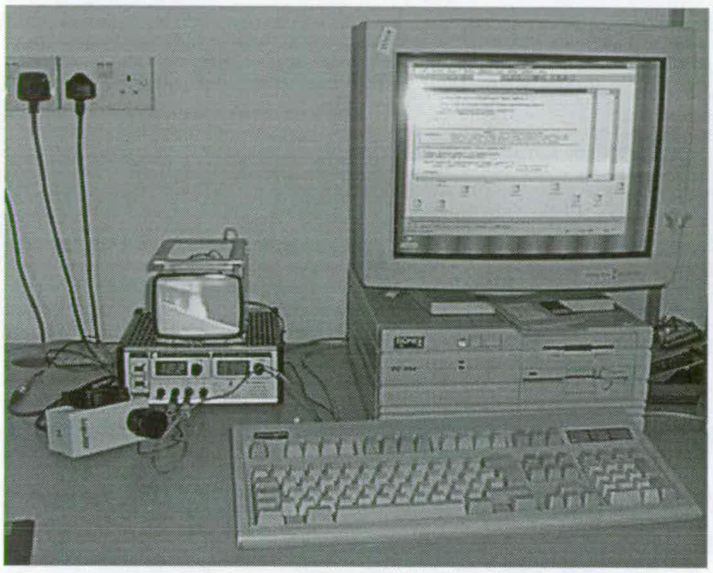


Figure 4.13 ***The PINK FPGA test set-up.*** *The imputer system (containing the FPGA), the power supply and monitor can be seen to the left of the PC.*

Physical connection of an FPGA to the imputer system requires a printed circuit board (PCB). An existing imputer PCB design was used. It required minor modification, but its use saved both design and fabrication time.

To control the use of the FPGA within the imputer system firmware was written to run on the microcontroller of the imputer. A library of 'C' functions was created that implements all the control functions offered by the FPGA design. Together with a menu-based test program, this library was used to perform all the hardware tests. Full listings of the library functions and test software can be found in appendix three.

The PC in the test system was used to write and compile the test firmware, for long-term image storage and as a terminal emulator for interaction with the menu-driven imputer test program.

Test procedures

Initial proof of test system integrity

Initial testing of the system FPGA and software was made without live video input or output and without the video generator in the system. A test image was loaded into imputer memory from the PC. The test software controls processing of this image using one of the diffusion processors. The resulting image was then uploaded to the PC for inspection.

After minor teething troubles with the test software a processed image was successfully uploaded to the PC. This simple result in itself proved much of the basic functionality of the system. Three conclusions can be drawn from the fact that different memory banks are used for the FPGA input image and its output image (and that a processed image has been successfully uploaded). The microcontroller has successfully initiated a frame of image processing, the FPGA has successfully run a complete frame of image data through a processor pipeline and finally, it has returned control of the imputer bus to the microcontroller.

Checking algorithm implementation

Verification that the processors had been implemented correctly was achieved by comparing the results of images processed using the processors against results obtained during the hardware and software simulations. To ease verification, simple test images were first used. The first images used were linear greyscale ramps (both vertical and horizontal) followed by more complicated (and thus more realistic) still images (e.g. lena).

When making detailed analysis of the results from the hardware processors, the continuous nature of their operation has to be taken into account. The processor pipelines operate without ever being flushed, thus the exact result of processing data depends not only on the data and the processor logic, but also on the contents of the processor's pipeline before the processing began. When scrutinising the operation of the processors and comparing their results with those from the simulations, care has to be taken that the contents of the pipeline have been flushed.

This is possible by applying a reset pulse to the FPGA, which resets all internal pipeline nodes to zero.

Live video input and display

Once the basic operation of the processors had been investigated the video generator card was inserted into the system to enable the live display of processor output.

During testing it was noted that data could not be written into the buffers of the video generator as fast as had been expected. In order to successfully write each pixel into the memory the FPGA clock had to be slowed down. The reduction in speed means that the total time taken to grab and process one image now exceeds the period of two video frames. Instead of processing every second video frame from the sensor, the system can only process one in every three. This reduces the live processed frame rate from 25 fps to approx. 17 fps. A second complication is that the use of a single image buffer for both diffusion processor output and video generator input caused tearing of the image on the display. This is due to the two asynchronous rasters (one each for the PINK FPGA and the video generator) occasionally crossing in memory. The tear can be avoided by alternating between two buffers, however, this increases the delay between image capture and processed image display to 60 ms.

Results

This section discusses the results of the programmed device testing, and some of the statistics of the final design.

'Safe perturb 1' failure

Problems were encountered when trying to use the safe perturb 1 processor. Three findings suggest that the most likely source of the problem is in the FIFO interface. Firstly, interaction with the FIFO is the only feature of the design that was not fully simulated. Secondly, that the FIFO is the only major difference between the *safe* *pertub 1* and *simple* processors. Finally, the nature of the failure is consistent with incorrect FIFO clocking.

The problem exhibits itself as a slow scrolling of the output picture from right-to-left after the processor starts. After it has scrolled once its stops. Using the 'nudge' feature of the FIFO manager, the image can be scrolled back, however, it drifts over again. This suggests that the FIFO is interpreting more 'read' than 'write' commands - thus the simulated section of pipeline gets slowly shorter until it disappears. The speed of the scroll is consistent with a single extra 'read' command per frame of image processing (the scroll takes approx. 15 s). This suggests an error when the processor is either starting or stopping.

Perceived display linearity

Careful scrutiny of uploaded output of the *simple* processor verified its successful implementation. Initial comparison of the results of its operation in the live display mode (17 fps) with that of the software simulation of the same algorithm were, however, disappointing.

Apart from a higher update rate (which was expected to improve perceived quality) the main difference in the hardware version is the display. Output from the software simulations is viewed on a small portion of a 17" PC monitor, the hardware test system uses a 5" monochrome monitor.

Investigations into the perceived linearity of this 5" monitor show it to be far from linear. Perceived display linearity is important to the success of error diffusion, as when errors are arranged to cancel the energy is assumed to add and subtract linearly.

Attempts to correct for the non-linearity using conventional 'gamma' power functions were unsuccessful. Investigation of the characteristics of the display showed its excitation vs. perceived luminance transfer function to be sigmoid in shape (see the required correction function shown in Figure 4.14). The imputer can be used to apply a correction function using a look-up table. Unfortunately, this cannot be performed in real time.

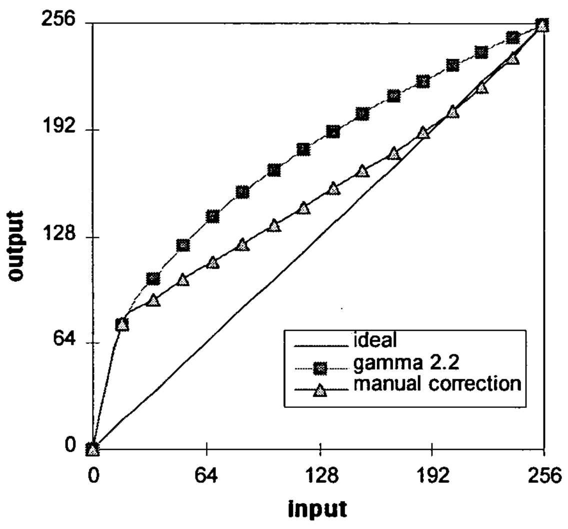


Figure 4.14 ***Correction of the perceived linearity of the test system monitor.*** *A successful manually derived correction function is shown. A gamma correction function (2.2) and y=x are shown for comparison.*

When display linearity was corrected the diffusion images displayed on the 5″ monitor were considerably closer to those seen in the software simulation. This example of how dependent the algorithms output is on factors such as display linearity reinforces the importance of control over the viewing conditions to the successful application of error diffusion.

Processor speed

Most of the testing was performed using a 20 MHz oscillator and the FPGAs internal clock divider to allow display using the video generator. The FPGA was also tested running at the full 20 MHz. At 20 MHz, the originally intended operating speed, the *simple* processor's operation was verified through transferring processed images to the PC and comparing them to simulation results.

Using an external clock generator the processor was also seen to operate properly up to a clock frequency of 27 MHz. At this frequency, the FPGA began to violate the timing requirements of the 70 ns RAM.

Images

Single frame examples of the processed output from the *simple* and *truncate* processors of the FPGA are shown below. As single, static images these are similar to those presented in the software simulation of chapter three, thus only a few examples are shown. When interpreting the static representations contained in this thesis, it should be kept in mind that the processors are designed to output a *series* of images at a high frame rate. This will affect the appearance of the noise produced by the simple diffusion processor. At the higher colour resolutions the moving noise is far less visible than when static. At the lower resolutions, however, it could be argued that it is more objectionable when moving.

A vertical greyscale ramp (one of the initial test images) is shown processed by the *simple* and *truncate* processors in Figure 4.15 below. This image demonstrates the success of the *simple* processor at representing smooth vertical gradients. At the higher colour resolutions the dithering pattern is not very noticeable, as the resolution drops the pattern becomes more obvious. The structure in the dithering noise is a result of the simplicity and deterministic nature of the *simple* algorithm used. Even at 4 bpp, distinct banding can be seen in the output of the *truncate* processor.

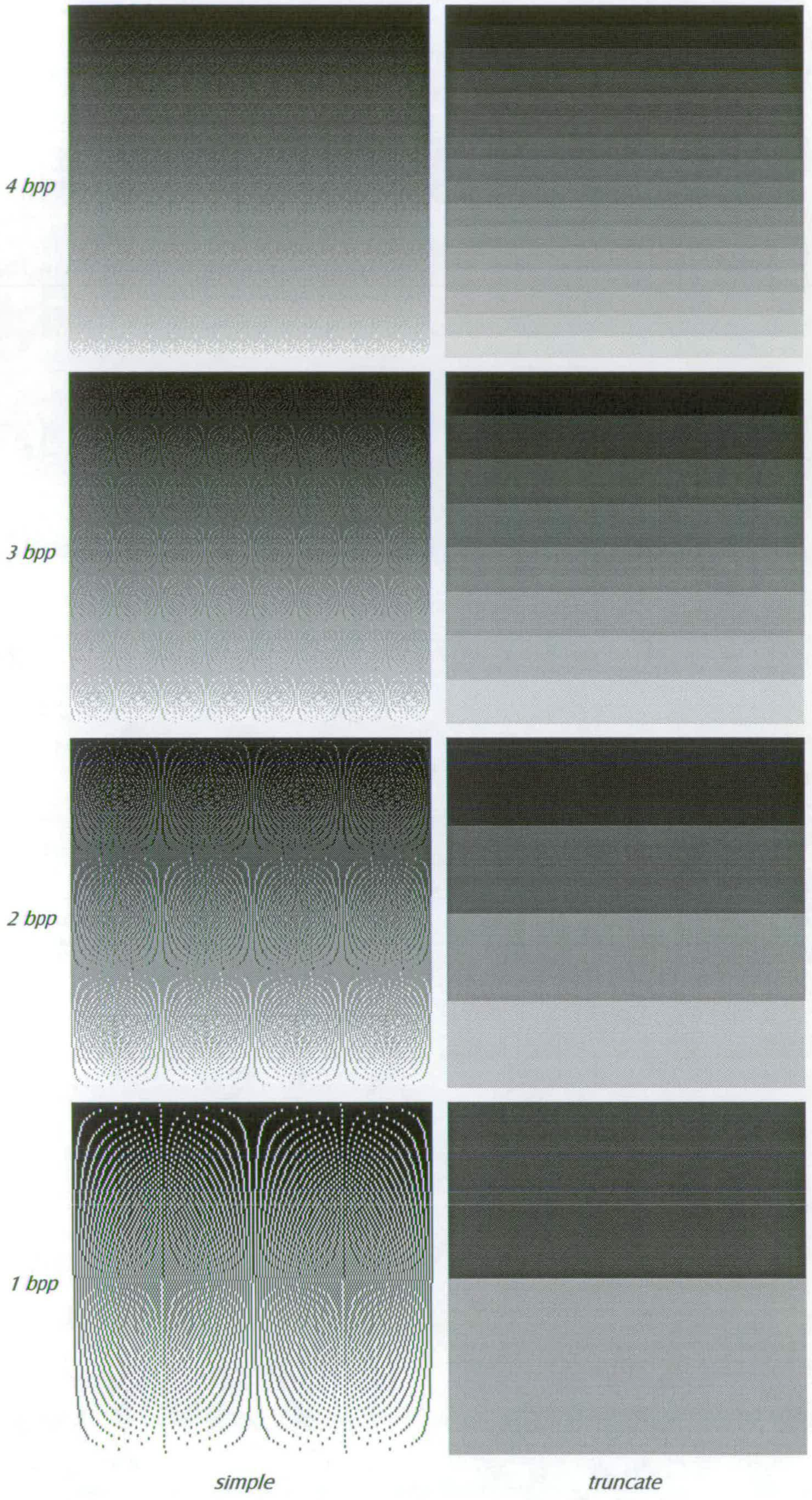


Figure 4.15 ***A comparison of the output of the 'simple' and 'truncate' processors.***

Processing of a horizontal greyscale ramp by the *simple* processor is shown in Figure 4.16. The dramatic difference in diffusion pattern highlights the directional nature of the *simple* algorithm.

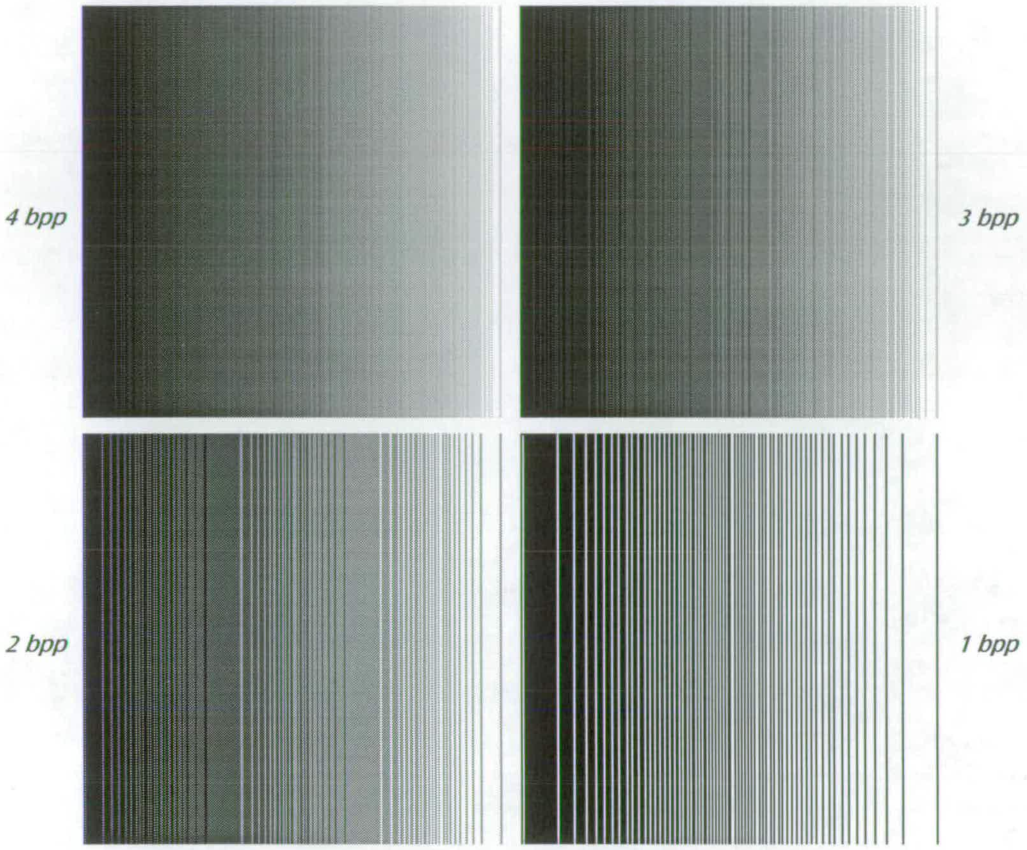


Figure 4.16 ***A horizontal grey ramp processed using the 'simple' processor.*** *4 bpp, to 1 bpp (as marked).*

In Figure 4.17 a comparison between the results of the hardware and software implementations of *simple* is shown. The pictures are almost identical except for slight differences in the dither pattern. These differences are due to the continuous nature of the hardware processor. Unlike the software implementation which gives the same result each time it processes the same data, the output of the hardware processor varies slightly with every frame (as its output is dependent on the contents of the pipeline when processing starts).

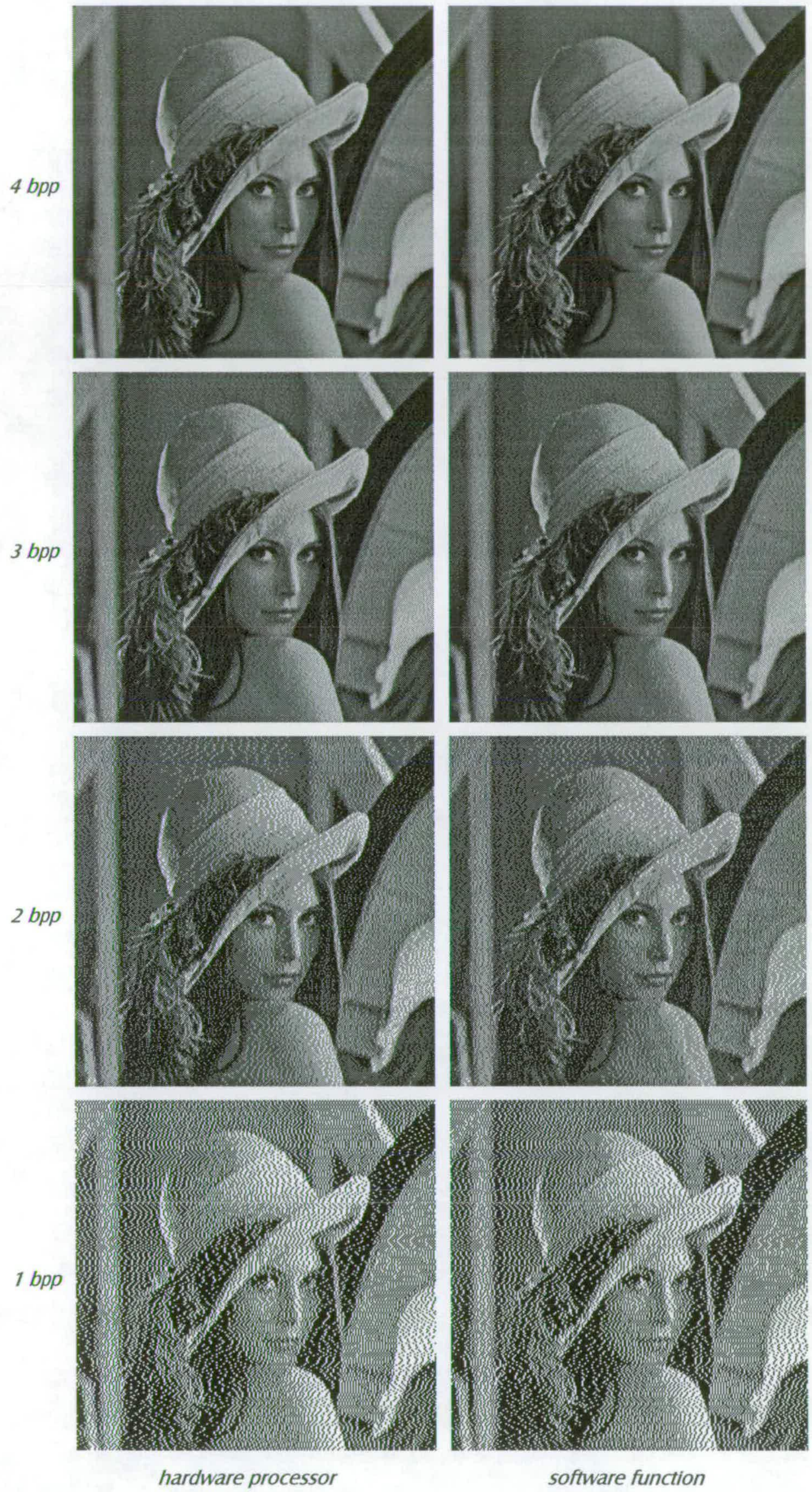


Figure 4.17 ***Comparison of the output of the hardware 'simple' processor with that of its software counterpart.***

Conclusions

This chapter details the implementation of two error diffusion algorithms (*simple* and *safe perturb 1*) in an FPGA. Earlier software simulations suggests that these algorithms would be suitable for use in the radio video application.

The *simple* algorithm has been successfully implemented in hardware. As part of an imputer system it offers processing of images at 17 fps at one of four greyscale resolutions.

Implementation of the *safe perturb 1* algorithm was unsuccessful. This is due to a problem in the interface with the FIFO device used to simulate a section of the processor pipeline too large to fit onto the FPGA.

In the output from the *simple* processor, the high frame rate achievable with the hardware implementation is seen to alleviate the patterning problems of the diffusion algorithms at 3 and 4 bpp. At lower colour resolutions, however, the noise remains clearly structured.

The imputer system proved to be a valuable test bed.

Although the entire FPGA design was equivalent to approx. 7000 gates, less than 7% of these were attributable to the two diffusion processors. Furthermore, if implemented as part of the output stage of a digital-output video sensor, the hardware overhead of the simple processor would be that of a single adder and latch.

The successful implementation of the *simple* processor confirms the ability to implement error diffusion with minimal hardware assumed during software simulations. This justifies the selection of error diffusion as the compression method for the radio video application.

Sync information must be added to the pixel data output from the diffusion processor before it can be transmitted. This could be achieved by time-multiplexing reserved 'escape codes' with the data. As such, the hardware diffusion processor represents the core of a possible coder for the sensor-transmitter interface.

chapter five

Subjective Testing

When evaluating and comparing lossy image compression techniques three factors must be considered: the degree of compression each technique yields, its complexity and the resulting loss in image quality.

In chapter three several implementations of compression through quantisation with error diffusion are explored. Scrutiny of the algorithms concentrates upon the complexity of their hardware implementation and the limited subjective evaluation of their effect on image quality at comparable bit rates. The complexity of implementation and bit rate are easily measured objectively. The effects on image quality, however, are the subjective views of the author. To truly have confidence in these conclusions, more substantial examination of the output of the algorithms involving more observers is required. Such testing is the subject of this chapter.

Assessing image quality

'Image quality' is a term commonly referred to in the literature, but one that lacks a precise definition. In the context of compression, the term is generally used to refer to how closely a processed image represents the original. This can be interpreted as any change in the image data itself, or possibly more importantly, any alteration to the data that leads to a perceivable change in the image.

The matter is complicated further as it is not only the visibility of distortions that is important, but also how objectionable they are and how they affect the perceived image. In some applications the usefulness of an image may be unaffected despite quite obvious processing artefacts as long as certain key aspects remain unaltered (such as the ability to reliably determine the absence or presence of some feature). Indeed some compression techniques that inherently remove random noise could be considered to improve the image quality in such situations [*COSMAN 94*].

Subjective measures of image quality

A common approach to the measurement of image quality is to present a set of images to a panel of observers and ask them to rank the images in terms of relative quality [*WALLACE et al 88; COSMAN 94; PRATT 79*]. Where image qualities are drastically different, ranking by an observer is quick and unlikely to be disputed. However, judging the superior algorithm among those of similar output is not so trivial.

Another problem of such subjective observer tests is that the test conditions vary among researchers, making direct comparison of results difficult. Some standard test features have been adopted (such as how to relate how visible or objectionable errors are to a scale of 1 to 10). As many alternative applications are considered, the desire for a standard test procedure (allowing simple comparison of results) is seen to conflict with the researcher's need to mimic the conditions of the intended application as closely as possible (to increase the accuracy of the test).

The nature of subjective tests opens them up to problems of bias. If for any reason the background of the observer may influence either their objectivity, or their ability to detect types of errors (through adaptation) then there will be a danger of observer bias. There are two schools of thought on observer adaptation, those that believe observers should be allowed considerable time to become comfortable with the experiment and consistent in their ranking of quality before any results are taken [*SAKRISON 77*] and those that see the adaptation of observers to the test pictures and the type of errors as a factor that is prone to make them over-critical, especially in such an artificial situation as the tests where they are being asked to actively search for errors [*WALLACE et al 88*].

For the researcher, subjective trials are tedious, time consuming and their very nature means they are not precisely repeatable. It would be much more convenient to have an objective meter that is repeatable and quick to compute.

Simple objective meters

Various simple meters of averaged individual pixel error exist. These are generally computed by calculating the pixel-to-pixel difference between original and processed images then collapsing this onto a single variable quantity such as the mean-squared error (MSE), signal-to-noise ratio (SNR) or peak signal-to-noise ratio (PSNR).

Although these meters can be applied successfully within narrow fields of research, their accuracy and thus value as general meters is questionable. Examples of the failings of mean-squared error (by far the most commonly used simple meter) include both the over-emphasis it places on image modifications that are often relatively unimportant (such as a small spatial image shift or a small d.c. level change) and its insensitivity to the corruption of small but possibly important image features.

For an objective meter to be of use it must reflect the 'perceptual quality' or the 'usefulness' of an image for a particular application. Unfortunately, no meter that is simple to compute is generally accepted to satisfy this requirement [*COSMAN 94*].

The human visual system

A subtle but important point when considering the design and evaluation of image systems that are to be viewed by people is the effect of the human visual system. When an image is viewed, the information conveyed to the observer is dependent not only on the actual image, but also on the way in which it is interpreted. In reality, factors completely unrelated to the image, such as the experience of the observer, can have a bearing on this interpretation. Other factors, however, relate to features of the image and stem from physiological and psychological features of the human visual system. Such factors are common to all human observers.

Frisby described the problem of seeing as "the problem of building up a symbolic description of a scene using the information contained in an input visual image" [*FRISBY 79*]. The physiological aspects of our visual system determine how the information from the visual image is presented to our cognitive systems. These aspects combined with the psychological mechanisms of data abstraction that construct this 'symbolic description' underlie the interpretation of scenes and thus also the interpretation of image distortions and other forms of compression error.

Understanding characteristics of the human visual system allows an informed approach to be taken when considering the effects of distortions in visual image data on the perception of that image. This necessitates a subjective approach to the design and evaluation of lossy image compression techniques. To that end, a vast number of mathematical models of the human visual system have been proposed [*MANOS & SAKRISON 74; HALL & HALL 77; XIE & STOCKMAN 89*]. The basis for these models comes from two major sources. The first is data gathered from psychophysical experiments using human observers. In these, characteristics of the human visual system such as its sensitivity to stimuli of various spatial frequencies are measured. The second source is data from the physiological study of the visual systems of animals such as cats and monkeys. Results from the experiments are related to the human visual system on the premise that the vision systems of all vertebrates are similar. (For comprehensive surveys of this evidence see [*MARR 82; HALL & HALL 77*].)

Objective meters based on models of the human visual system

Models of the human visual system have been used to offer objective measures of image quality with varying degrees of success by many researchers working on the analysis of image compression techniques. The models are generally used to compute an error metric from the difference between original and processed images or image sequences. Some of the more popular features of models are introduced below with consideration of the limitations of their general application.

Spatial and temporal filtering

The sensitivity of the human visual system is dependent on both the spatial and temporal frequency of a stimulus. Physiological and psychophysical evidence suggests that the sensitivity can be modelled using simple functions [*ROBSON 66; PEARSON 75; CAMPBELL & MAFFEI 74*]. It should be noted, however, that the spatial and temporal responses are not entirely independent [*ROBSON 66*]. Typically, this nature is modelled by linearly weighting the significance of data using a measure of local frequency content.

Multi-channel processing

Evidence exists which supports a multi-channel model of the human visual system. In this, several quite separate channels (sensitive to different spatial or temporal frequencies) are used to analyse image information. The supporting evidence includes the observed ability of people to suppress stimuli of certain temporal frequencies and the independence with which stimuli of quite different spatial frequencies can be detected [*CHAU 90; MULLIGAN 93*]. Multi-channel behaviour has been modelled using linear processing in separate channels followed by non-linear recombination of the results [*SAKRISON 77*].

Correction for non-linearity

It has long been known that the human retina has a highly non-linear response to incident light [*PEARSON 75*]. Models that attempt to incorporate this feature do so by using non-linear transfer functions (such as log [*REED 92; CHADDA & MENG 93*] or cubic-root [*GRANRATH 81*]).

Context masking

The visual context of an error has been shown to affect its significance. The less correlation between local image activity and error, the more obvious it is. This is often referred to as *masking* (as errors are effectively masked by image activity). This feature has been successfully modelled by linear weighting of errors. Use of a global, purely arithmetic average of image activity and (more successfully) a local, geometric average have been proposed as weighting functions [*LUKAS & BUDRIKIS 82; REED 92; COSMAN 94; CHADDHA & MENG 93*].

To allow the easy comparison of results a single scalar figure of merit is often realised from the three-dimensional error image created from the model. Non-linear measures are often employed in this conversion (as in the simple metrics). These include calculation of mean squared error, other 'l_p norms' such as the cubic root of the sum of cubed errors (l_3) and the maximum error (l_∞), as well as signal to noise ratios. There is some theoretical support for the use of non-linear methods. Research has shown that in subjective tests observers base their ratings on the worst areas of images (giving any high errors a disproportionate significance) [*LUKAS & BUDRIKIS 82*].

Although many promising results have emerged from the use of mathematical models of components of the human visual system, no complete model yet exists. Many of the partial models are limited in their application because of the factors outlined below:

- Much of the data used for modelling functions (such as spatial frequency sensitivity) is based on experiments that assume the stimulus is viewed against a uniform background [*PRATT 79*]. In image compression, the stimuli are compression artefacts and the background is the source image. Models based on the experiments with uniform backgrounds are therefore unlikely to be directly applicable.
- A second problem is that the *stimuli* used in many psychophysical experiments differ greatly from those that are important in the compression. Much perception experimentation is based on random perturbations such as additive white noise rather than realistic compression artefacts [*SAKRISON 77*]. Care should be taken when applying models derived from these experiments.
- Another assumption often made is that the image quality is fairly high thus errors are on the verge of being detectable rather than obvious. This again simplifies the model as only the threshold of perception of stimuli needs to be modelled. In the case of error diffusion at low bit rates this simplification cannot be made.

The interpretation of modelling results is also limited. Most analysis ultimately relies on pixel-to-pixel comparisons. No attempt is made to abstract data from the image or consider changes in semantics caused by errors. The complication of constructing objective meters that can perform such tasks often leads back to the use of subjective tests.

Using objective meters with error diffusion

The approach of error diffusion, indeed all halftoning algorithms, is to minimise locally averaged error at the expense of instantaneous pixel fidelity. Amongst compression algorithms, error diffusion faces particular problems in satisfying the simple objective image quality meters.

Although some of the key features of the human visual system models published are not applicable to the analysis of error diffusion (especially at the bit-rates and under the viewing conditions of the radio video link) many of the model features such as spatio-temporal sensitivity and non-linear luminance response are directly relevant.

If a mathematical model were to be used to evaluate algorithms for use in the radio video link there are features extra to the human visual system that could be built in. In particular these include features of the transmission channel (such as bit-error rate) and the display mechanism such as limited frequency response (see Kell factor in [*BLINN 94*]) and non-linearity. In the interpretation of the results of any error modelling, a meter such as the radially averaged power spectrum [*ULICHNEY 88; MITSA 92*] could be profitably used to measure any structural content (indicative of textural patterning from the error diffusion).

Pressure on time meant that an attempt at objective analysis of the implemented error diffusion algorithms using models of the human visual system was not possible. A program of subjective tests was, however, devised. The experimental design and the hypotheses it seeks to test are outlined below.

Hypotheses

The experiment consists of a set of subjective comparisons of images and image sequences. These are devised to test the validity of the author's conclusions (drawn in chapter three) regarding the relative output image quality of the various diffusion algorithms.

These hypotheses are:

1. All algorithms introduce a perceivable degree of degradation at 3 bpp.
2. At 3 bpp and 4 bpp the output of the *simple* algorithm is comparable to that of Floyd & Steinberg's filter.
3. The problems of the *perturb* algorithm are alleviated to some extent by the changes made to produce the *safe perturb 1* and *safe perturb 2* variants. These changes are significant (and therefore merit the increased implementation cost).
4. Amongst the single element filters *safe perturb 1* and *simple* offer the best results at 3 bpp.

Setting up an experiment to test these hypotheses provides an ideal opportunity to explore three other aspects of the perceived quality of the algorithms' output.

1. The output resolution at which processing becomes apparent
2. The degree to which the choice of source image affects the success of the algorithms
3. The degree to which the success of the algorithms is improved by the use of image sequences as opposed to stills

An experiment designed to test all these hypotheses is described below.

Experiment design

Comparing every algorithm at every resolution would constitute an enormous exercise. Instead, the experiment uses a selection of comparisons targeted at testing the hypotheses set out above. It represents a 'pilot' programme of tests. The results of which could be used to direct more comprehensive testing of the particular trends exposed.

Two types of test are used in the experiment: forced-choice between pairs of image sequences and the sorting of still images in order of perceived quality. In the forced-choice tests, the subject is presented with pairs of image sequences and asked to indicate which of the two they judge the better. Detection of a difference between the sequences is indicated when the amalgamated results show that viewer choice departs significantly from random (e.g. from 50%). Results of the still image sorting are used to rank the output of the algorithms.

The experiment comprises three sets of tests. Firstly, a set of forced-choices between sequence pairs where each has been processed to the same output resolution, but using different algorithms. Secondly, the forced-choice between sequence-pairs where one has been processed, the other not. Thirdly, the sorting of sets of six still images in order of their perceived quality. All six images having been processed using different algorithms, but to the same resolution.

These tests relate to the hypotheses outlined earlier as follows:

Processed vs. unprocessed

The forced choice tests using processed and unprocessed sequences test the degree to which processing can be perceived at a particular resolution. The limit of perception is thought by the author to lie around 4 bpp and 5 bpp: the resolutions used in the tests.

The use of three different image sequences in this test allows the investigation of the dependency of the results on the source image.

Algorithm comparisons

Forced choice tests between sequences processed to the same resolution (but using different algorithms) is intended to allow the relative success of the algorithms to be evaluated. In terms of the original hypotheses, this test explores: the relative qualities of simple and floyd-steinberg at 3 bpp and 4 bpp; the significance of any benefit in employing the more complicated perturb algorithms (rather than perturb itself); whether simple and safe perturb 2 are the best of the single element filters at 3 bpp.

Still image sorts

Comparing sets of six still images (all processed to the same resolution but using different algorithms) allows ranking of the perceived effectiveness of all six algorithms at a particular resolution.

Still images are used both to reduce the length of the whole experiment and to allow the effect of image movement to be explored. The latter is achieved by comparing the results from this test with those from the paired image sequence algorithm comparisons.

In this test all three images are used again in order to further explore the dependency of the results on the source image.

Source images

Research has shown that the visibility of errors depends on their visual context [*LUKAS & BUDRIKIS 82*]. In addition, (in common with nearly all image compression techniques) the form of errors produced by quantisation with error diffusion depends on the image content. Thus, the subject matter of the images sequences used in tests can have an effect on the outcome. To give some spread of subject matter, three test sequences are used. Two 'head-and-shoulders' sequences (*claire* and *miss america*) and a wider angle office scene (*salesman*) were obtained from the USC database (see Figure 5.1 below).

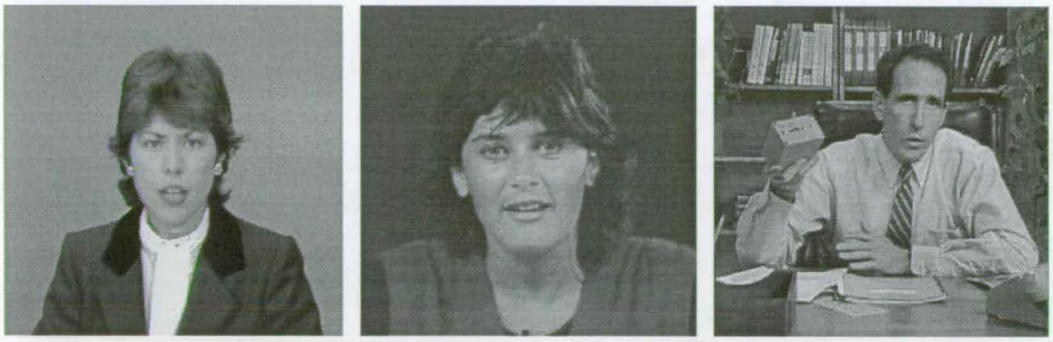

(a) *(b)* *(c)*

Figure 5.1 *Single frames from **the three test sequences** used in the subjective tests: (a) 'claire', (b) 'miss america', and (c) 'salesman'. The sequences were obtained from a database at the University of Southern California, Los Angeles (ftp://ftp.ipl.rpi.edu/pub/image/sequence/).*

These sequences were selected for use in the tests as they are widely recognised in the literature and are considered to be representative of sequences likely to be transmitted in the radio video link application.

Measures employed to reduce false results

Measures were incorporated into the experiment to reduce the possibility of extraneous factors influencing the results.

It was anticipated that seeing the gross artefacts of the low bit rate images might make them more noticeable in their higher bit rate counterparts. Similarly, it was anticipated that observers may become adapted to being able to 'spot' certain classes of defect more easily after viewing the still images (as the visibility of artefacts was anticipated to be clearer in the still images).

Presenting all of the forced-choice pairs in decreasing order of resolution could be used to prevent the first of these sources of adaptation. However, it may introduce another source of error, in that the observers may come to expect increasing levels of artefacts, irrespective of the algorithm. Instead, in accordance with guidelines produced for evaluation of television images [CCIR Rec. 500-3] in these experiments the order of the image pairs is randomised.

To prevent the observer adaptation from viewing the still images affecting their perception of the sequences, all the sequence comparisons are presented first.

In case the physical location of each image on the screen has any influence, this factor is also varied by using three different sets of randomised tests.

Test software

A software application has been written which allows the presentation of up to six still images or image sequences at once (see Figure 5.2 and Figure 5.3). The still images are presented together, whereas sequences are shown individually to ensure they are displayed at the highest possible frame rate.

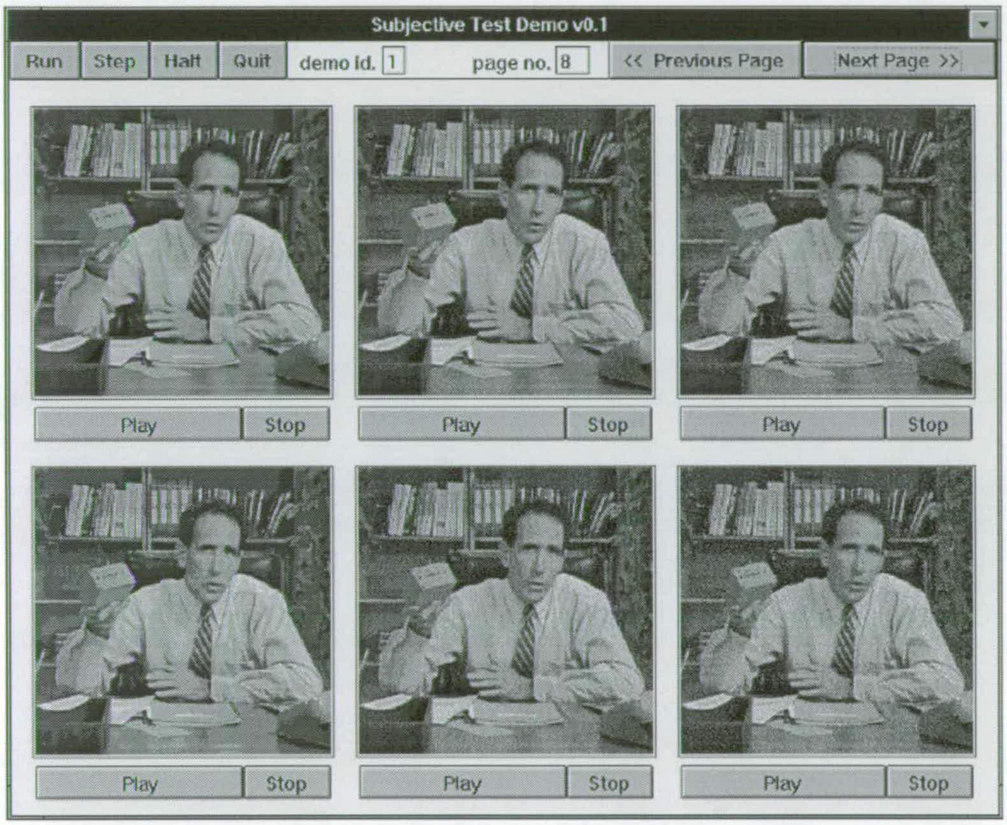


Figure 5.2 ***The window of the subjective test application during display of a set of still images.***

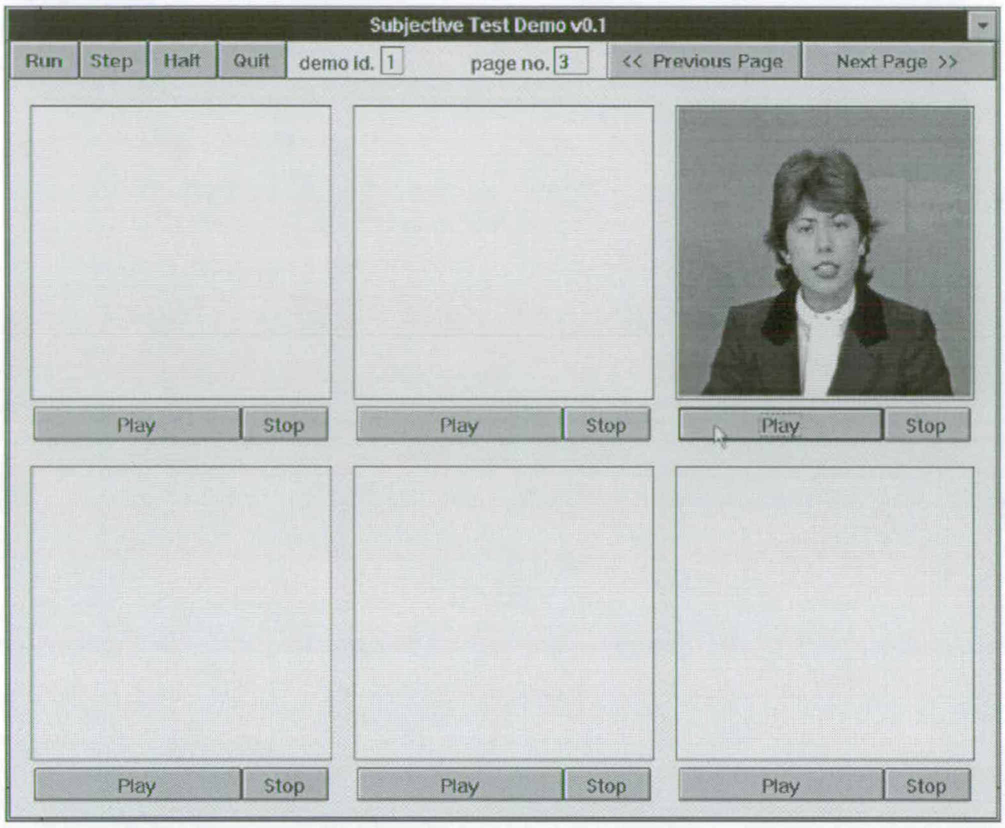


Figure 5.3 ***The window of the subjective test application during display of a moving image sequence.*** *The five image sequences that are not currently playing are left blank rather than displaying a static image so that subjective analysis of the image sequences is not coloured by observation of still frames.*

The playing of image sequences is achieved using the repetitive display of pre-processed bitmaps stored locally on the PCs hard disk. The images are rendered using a version of the simulation software that allows the batch processing of images. The number of images involved requires a significant amount of storage[1]. This approach, as opposed to on-the-fly computing of the processed images (using the software implementations of the algorithms) is employed for two reasons. Firstly, so that the image sequences can be presented at a reasonable frame rate (roughly 12 fps) and secondly, so that all the sequences are presented at the same frame rate, independent of the complexity of the algorithm used.

The C source code for the test software can be found in appendix four.

[1] Each of the three sequences was made up of sixty 64 k frames, in addition to these unprocessed images each was also stored after processing with one of the six algorithms, and all algorithms at five different bit rates. All images for the three different sequences totalled approx. 400 Mb.

Results

The results of the subjectve tests are presented below.

Algorithm comparison tests

Results, comparing the image sequences, are shown in matrices in the three tables below (Table 5.1, Table 5.2 and Table 5.3). The samples sizes vary from 7 to 23. Results shown in bold type are statistically significant (given the sample size). Significance is determined using the 'sign test', with $\alpha=0.05$ [ALDER & ROESSLER 72].

	floyd-steinberg	*simple*	*safe perturb 1*	*safe perturb 2*	*perturb*	*truncate*
floyd-steinberg better than	-					**100%**
simple better than		-		42.9%	87.5%	**78.3%**
safe perturb 1 better than			-	69.6%		
safe perturb 2 better than		57.1%	30.4%	-	**87.0%**	**78.3%**
perturb better than		12.5%		**13.0%**	-	60.0%
truncate better than	**0.0%**	**21.7%**		**21.7%**	40.0%	-

Table 5.1 ***Results of the algorithm comparison tests at 2 bpp***

	floyd-steinberg	*simple*	*safe perturb 1*	*safe perturb 2*	*perturb*	*truncate*
floyd-steinberg better than	-	**87.0%**		**100.0%**	**91.3%**	**95.7%**
simple better than	**13.0%**	-		65.2%	**73.9%**	**91.3%**
safe perturb 1 better than			-	56.5%		
safe perturb 2 better than	**0.0%**	34.8%	43.5%	-	52.2%	**73.9%**
perturb better than	**8.7%**	**26.1%**		47.8%	-	56.5%
truncate better than	**4.3%**	**8.7%**		**26.1%**	43.5%	-

Table 5.2 ***Results of the algorithm comparison tests at 3 bpp***

	floyd-steinberg	*simple*	*safe perturb 1*	*safe perturb 2*	*perturb*	*truncate*
floyd-steinberg better than	-	43.5%		65.2%	**73.9%**	47.8%
simple better than	56.5%	-		56.3%	**73.9%**	56.5%
safe perturb 1 better than			-	37.5%		
safe perturb 2 better than	34.8%	43.8%	62.5%	-	56.5%	47.8%
perturb better than	**26.1%**	**26.1%**		43.5%	-	34.8%
truncate better than	52.5%	43.5%		52.2%	65.2%	-

Table 5.3 ***Results of the algorithm comparison tests at 4 bpp***

Testing performed at 2 bpp is relatively sparse as this is below the intended resolution of the radio video link coder (3 or 4 bpp). However, the results gathered do show the poor performance of the truncator when compared with all but the *perturb* algorithm. There are no significant differences between *simple* and the two *safe perturb* algorithms at 2 bpp.

The 3 bpp and 4 bpp results are based on a larger sample size. Examining the figures for the two resolutions shows a high number of significant results at 3 bpp, fewer significant differences at 4 bpp. At this higher resolution, the only significant difference noted by observers is in the poor performance of *perturb* when compared with both *simple* and *floyd-steinberg.*

The improved ability of observers to discriminate between the different algorithms at 3 bpp (compared with 4 bpp) implies greater perceivable differences between the algorithms at this lower resolution. This corroborates the hypothesis that differences between the output of the algorithms are clear at 3 bpp.

Floyd-steinberg is preferred to the other algorithms at 3 bpp. Table 5.2 shows it to be significantly better than all other algorithms tested at this resolution. *Simple* also performs well, although worse than *floyd-steinberg,* it is preferred over *perturb* and *truncate.*

Safe perturb 1 is omitted from most of the algorithm comparisons, because its performance is expected to be very similar to that of *safe perturb* 2. The validity of this omission is borne out by the results shown above. No significant difference is found between *safe perturb* 1 and *safe perturb* 2 at any resolution.

Still image sorting tests

The resulting ranks from the still image sorting test are shown in Figure 5.4 and Figure 5.5 below. High ranking scores indicate images preferred by the observer.

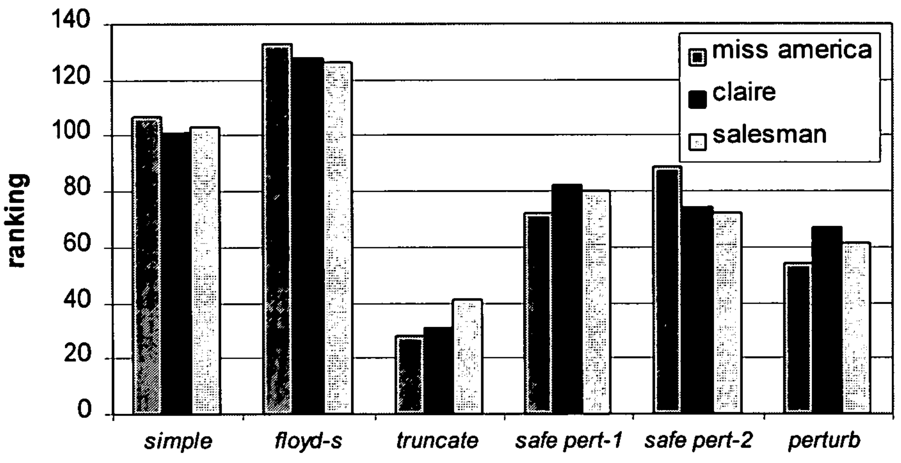


Figure 5.4 ***Results of the 3 bpp still image sorting test.***

At 3 bpp the ranking of the algorithms is distinct. For all three images *floyd-steinberg* has the highest rank, *simple* the second and *truncate* the lowest. The rankings of the variants of *perturb* suggest that the two *safe* variants are preferred to *perturb* itself, alth

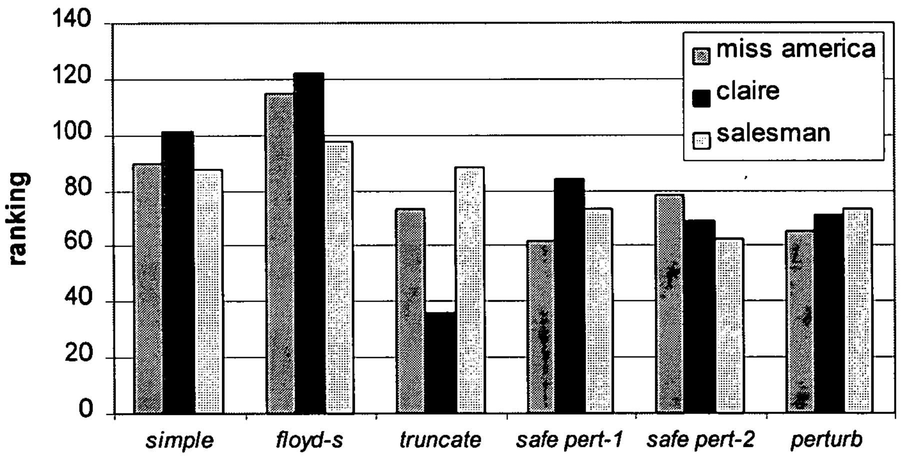


Figure 5.5 ***Results of the 4 bpp still image sorting test.***

The ranking pattern at 4 bpp is not so simple. Again, *floyd-steinberg* is consistently highest, followed by *simple,* but the distinctions between the remaining four are unclear.

The results of the low rankings achieved by *claire* at 4 bpp are interesting in that they indicate a clear incompatibility between *claire* and *truncate.* This is likely to be a result of the almost flat background in the image which will be particularly susceptible to false contouring. This result supports the research hypothesis that perceived algorithm success is image dependent.

Processed vs. un-processed

Table 5.4 shows the results from the comparison of processed and unprocessed images. The figures show the proportion of observers who preferred the processed images to the unprocessed originals.

	claire		***miss america***		***salesman***	
	(count)	(%)	(count)	(%)	(count)	(%)
simple 4 bpp	3/8	38%	1/2	50%	2/7	29%
simple 5 bpp	3/4	75%			7/15	47%
truncate 4 bpp	1/7	14%	1/4	25%	3/8	38%
truncate 5 bpp	2/7	29%	3/8	38%	1/2	50%
perturb 4 bpp	1/8	12%	0/8	0%	2/7	29%
perturb 5 bpp	1/2	50%	1/4	25%	4/7	57%
safe perturb 1 4 bpp					1/4	25%
safe perturb 1 5 bpp					3/8	38%
safe perturb 2 4 bpp	3/7	43%	5/8	62%	3/8	38%
safe perturb 2 5 bpp	3/4	75%	5/8	62%	1/8	12%
floyd-steinberg 4 bpp	4/7	57%	0/8	0%	1/2	50%
floyd-steinberg 5 bpp	3/8	38%	2/7	29%	1/4	25%

Table 5.4 Results from the processed vs. unprocessed sequence comparisons.

The low sample size means that individual comparisons are only statistically significant if observers preferences are unanimous. The data can, however, be used to provide a further test of the research hypothesis that perceived algorithm success is image dependent.

Ranking the individual preference scores for each permutation of image and algorithm, then calculating a rank total for each of the three image sequences, allows the calculation of the Kruskal-Wallis H test statistic [KRUSKAL & WALLIS 52]. The result corroborates the research hypothesis, showing that there *is* a significant difference between the results from the three sequences (at $\alpha=0.05$).

Discussion & conclusions

This chapter uses a set of subjective tests both to question some of the conclusions of chapter three and to explore other hypotheses regarding the relative merits of the diffusion algorithms considered.

From the results of set of subjective tests performed with a modest number of observers (23) the following conclusions can be drawn:

1. At a resolution of 3 bpp, observers are able to detect artefacts from all the algorithms. This is consistent with the opinion of the author expressed in chapter three.
2. When comparing the output of Floyd & Steinberg's filter with that of *simple*, a significant difference is observed at 3 bpp, but not at 4 bpp. The lack of a significant difference at 4 bpp supports the hypothesis that the output of the two algorithms is comparable at this resolution. The significant preference shown by observers for the output of *floyd-steinberg* rather than *simple* at 3 bpp demonstrates that at this lower resolution a detectable difference exists between the two algorithms (that of *simple* is considered to be worse). This latter finding suggests that the research hypothesis might require modification - the output of the *floyd-steinberg* and *simple* filters may only be comparable at resolutions of 4 bpp (and above). This conclusion has to remain tentative at this stage because of the nature of the subjective tests employed. The results indicate that a *relative* difference does exist between the *simple* and *floyd-steinberg* algorithms at 3 bpp. However, the ordinal scale of measurement used means that it is not possible to quantify the *magnitude* of this difference.
3. The only significant difference between *pertub* and the *safe perturb* variants is found at 2 bpp (where *perturb* is considered worse). The lack of a consistent significant difference at different resolutions suggests that the added hardware expense of implementing either '*safe*' scheme rather than *perturb* is not justified.
4. As predicted, *simple* is found to significantly outperform both the *perturb* and *truncate* filters at a resolution of 3 bpp.

5. The results of the sequence comparisons suggests that observers show a greater ability to discriminate between algorithms at 3 bpp than at 4 bpp. This shows that they are able to discriminate between different algorithms at 3 bpp. The failure to differentiate between algorithms at the higher resolution can be interpreted in two ways. It could mean for each algorithm, the degree of image degradation is so minor that an observer cannot detect it. Alternatively, the even scoring could be the result of an similar degree degradation from each algorithm, resulting in a set of 'equally poor' images. From the results of these subjective tests, it is not possible to determine which of these scenarios is the case. Therefore, the threshold at which an observer can perceive processing to have occurred remains unidentified. This is an area which requires further, more targeted testing.

6. The observer's choice of preferred algorithm is found to depend on the image sequence used in the test. This is seen in the results of two tests. A statistically significant difference is found between the image scores in the test which measures an observers ability to discriminate between processed and unprocessed sequences. Secondly, examining the graphs of still image rankings shows considerable inter-image variation (e.g. the results of the *truncate* algorithm at 4 bpp). These findings support the hypothesis that perception of algorithm performance varies according to the composition of the image used.

chapter six

Discussion and Conclusions

This thesis tackles the problem of implementing a radio video link at low cost. Reducing this cost is important because the purchase and running expense of radio video link hardware is considered to be an obstacle to its wider application.

Two measures, the use of a low-power/low-bandwidth radio link and the further integration of the transmitter end of the system, have been identified as ways of reducing the costs. The objective of the remainder of the thesis was the design of an interface between sensor and transmitter that offers a degree of compression. This permits the use of a low bandwidth radio link. The coder design had also to be suitable for integration on the same die as the image sensor. Given the minimum image specification assumed and the estimated available radio bandwidth, a compression ratio of at least 2:1 was required of the coder.

The design of the coder first required the identification of the exact role of the interface. In addition to compression, two further aspects of pre-transmission coding (the vulnerability of the coded data to corruption and the ease with which it can be received) were identified as being important in the radio video link application. Other than favouring the use of fixed width codewords during compression, coding for ease of reception can be considered in isolation from the other aspects. Coding for compression and coding for improved error tolerance are, however, highly interdependent.

From a review of coding techniques that systematically protect transmitted data from corruption, their expense (either in bandwidth or implementation) was deemed prohibitive. Instead, a compression technique was identified that can offer a modest degree of compression without leaving the data too susceptible to transmission errors - error diffusion.

In the design of error diffusion system, various features of the quantiser and the diffusion filter were identified as factors which offer the possibility to trade off aspects of system performance against cost.

Software simulations explored the relationship between these factors. A range of potential quantisers were considered. The increased sophistication in performance associated with the more complex quantisers does not merit their high implementation costs. A truncating binary quantiser offers sufficiently sophisticated behaviour for this application. An output resolution of 3 bpp is dictated by the compression needs of the application. In simulations, single element diffusion filters perform well at 3 bpp. Their output is comparable to that of the popular error diffusion algorithm of Floyd & Steinberg.

Critical review and future work

This section combines a critical review of decisions made during the algorithmic research with suggestions for future work. It reconsiders previously dismissed options for diffusion and new possibilities that could stem from the adoption of alternative system architectures.

Dismissed options for diffusion

The evaluation of diffusion filters in chapter three dismissed anything other than *incidental* temporal diffusion. Diffusion of error energy over time is a natural progression from spatial diffusion, as it represents diffusion in an additional direction from the source of error. The marked similarities in the temporal and spatial insensitivities of the human visual system [*ROBSON 66*] provide a physiological justification. Temporal diffusion was dismissed because of the implied expense of error term storage. If an error is to be diffused in the time axis, an entire frame of error data must be stored. It was dismissed despite the perceived benefits of being able to keep error energy spatially closer to its origin and the reported success of temporal diffusion with still images [*MULLIGAN 93*].

A second option that was dismissed was the processing of data in orders other than the conventional raster. Use of a more pseudo-random raster or even a serpentine raster alleviates the directional nature of the diffusion patterns that result from the use of conventional raster. Like the decision regarding temporal diffusion, use of more sophisticated raster was also dismissed on the grounds of implied memory.

Hardware minimisation

Throughout the project great emphasis was placed on minimising the amount of hardware used in the coder. When considered in the wider context of the video transmitter, this emphasis may have been too high. The final error diffusion system proposed *does* offer the required degree of compression whilst preserving an image quality that is arguably adequate. On reflection, some of the design options ruled out on the grounds of hardware cost may actually have been affordable and might have offered improvements in image quality.

This is especially true if the coder is to be implemented in a system that employs an image sensor that produces video in a standard signal format. This is due to an incompatibility between the nature of the data within standard video signals and the desire for maximum transmission efficiency. For the bandwidth of the radio link to be used efficiently, data must be transmitted as continuously as possible. Unfortunately, the image data in a standard video signal comes in bursts between quite long vertical sync periods (over 10% of a standard PAL signal is taken up with synchronisation information - far more than is required for the synchronous transmission scheme planned for the radio video link). The buffering problem is exacerbated if a sub-array of the full sensor output is transmitted.

Even if the buffering problem was avoided by using an image sensor with specially tailored timing, there remain unavoidable system costs. One example is the logic incorporated in the sensor itself (ASIS-1011 includes approximately 10,000 gates to realise array addressing and automatic exposure control).

In the context of the complete video transmitter system it would appear in hindsight that the hardware budget for the coder could have been slightly more generous. This may have permitted use of the *floyd-steinberg* filter instead of *simple*, or exploration of avenues such as the use of *simple* together with a serpentine raster.

This would be consistent with the results from the subjective tests which suggested that the *floyd-steinberg* filter would offer an error diffusion scheme with slightly better performance.

To have changed the general compression approach (e.g. a move to transform coding rather than error diffusion) or to make substantial changes to the proposed system (e.g. extension to temporal error diffusion) would, however, still imply prohibitively high hardware costs.

Alternative architectures

Making quite significant changes to the architecture of the proposed video transmitter system means that it might be possible to achieve temporal diffusion and/or provide other benefits. Such changes, considered below, offer opportunities for future work. Particular emphasis is given to the advantages of non-standard image sensor architectures and eroding the distinction between sensor and coder that exists in the system.

In an application such as the radio video link, where video data is processed live, it is possible to use a more radical approach to otherwise expensive problems such as temporal diffusion and eliminating directional bias within diffusion. Performing processing within the imaging array itself combines the sensor and processor. Once the distinction between these two elements is eliminated there are many more options for the architecture of the combined system. Two examples of array level processing - pixel-level error manipulation and context dependent pixels - are considered below.

Pixel level error manipulation

An image sensor operates by regularly sampling the amount of light falling on an array of pixels. The data sampled from a single pixel can be considered as a time series. Within a normal sensor, each sample in such a time series is measured independently. If a system was used where:

1. pixels could be sampled to a greater degree of accuracy than each individual sample was communicated, and
2. it was arranged that the error in the communication of each sample from a pixel was used to influence the next sample from that pixel,

then temporal diffusion would have been achieved.

Integrating the means to store the error within the pixel structure would eliminate the requirement for separate storage. Influencing the next sample using the error would also negate the need for separate hardware to add error and sample terms.

A common way to implement an image sensor is to measure the amount of charge that has leaked through a light sensitive junction during a known exposure period. Typically, a pixel circuit is used that contains a capacitive node. The node is forced to a reset level before the exposure period, isolated from everything except the light sensitive element during the exposure period and at the end of the exposure period the amount of charge remaining on the node is sampled.

To apply the previous temporal scheme to such a sensor would require that each pixel began the exposure cycle, not at the reset level, but at the reset level plus or minus an amount that compensated for the previous quantisation error. This could be achieved by writing an error term to the pixel rather than resetting it fully. One problem with such a scheme would be its incompatibility with correlated double sampling (a technique commonly used to combat fixed-pattern noise by measuring each exposed pixel level with reference to its reset level measured immediately afterwards). A second problem is that unless the sensor is operated in a continuously exposed manner (unlikely as frame rate is then dependent on exposure time) the error would need to be stored over the time between each pixel being read and the start of its next exposure period. This would require investigation of reliable methods of analogue storage of the error term within the pixel.

In addition to the possibilities of temporal diffusion, the ability to manipulate error terms within the pixel array opens up possibilities of sensor-level spatial diffusion. Error terms could either be diffused to spatial neighbours as pixels samples are quantised and read out in some form of raster order, or a parallel diffusion stage could take place after exposure of all pixels. This second option would require a winner-takes-all approach to the distribution of what would otherwise be the quantisation errors. Both these options would require significantly more complicated pixel circuits than used in sensors such as ASIS-1011, however the benefits may outweigh the costs, especially as device geometries continue to shrink.

Context dependent pixels

Another option that would introduce compression at the pixel level would be to create a pixel structure whose output was not only dependent on the amount of incident light falling directly on it, but also on the amounts falling on neighbouring pixels. Reducing pixel sensitivity when the local neighbourhood is under bright illumination would allow a 'context-sensitive' output signal to convey an image with a higher dynamic range than the signal itself. Experimental evidence suggests similar systems are used in biological visual systems [*MARR 92*]. Such a form of sensing would essentially remove low frequency components from the image signal (c.f. differential predictive coding).

Conclusions

Whilst considerable scope remains for future research, the successful implementation of the *simple* processor in this thesis confirms that the video transmitter system coder can be implemented with minimal hardware. Thus, the objectives defined in chapter one have been met:

1. The role of the interface between sensor and transmitter has been defined.

2. The design of an example coder that realises modest compression at very little hardware cost has been proven feasible.

The development of a coder that can be easily integrated with a CMOS image sensor provides a significant step forward in the low cost production of a radio video link. The extremely small size of the example coder design means that adding it to the sensor die will not impact on its yield. Furthermore, the cost of the combined sensor/coder will be little more than that of a sensor alone.

References

Bryan ACKLAND, Alex DICKINSON, 'Camera on a Chip', *1996 IEEE International Solid State Circuits Conference (ISSCC96) Technical Digest,* San Fransisco, February 1996, pp. 22-23.

Henry L. ALDER, Edward B. ROESSLER, *Introduction to probability and statistics,* fifth edition, W.H. Freeman (San Fransisco), ISBN 0-7167-0450-1.

Rangarajan ARAVIND, Glenn L. CASH, Donald L. DUTTWELLER, Hsuenh-Ming Hang, Barry G. HASKELL, Atui PURI, 'Image and Video Coding Standards', *AT&T Technical Journal,* Vol. 72 No. 1 (January/February 1993), pp. 67-88.

R.L. ATKINSON, R.C. ATKINSON, E.E. SMITH, E.R. HILGARD, *Introduction to Psychology,* ninth edition, Harcourt Brace Jovanovich, 1987, ISBN 0-15-543682-1.

Thierry M. BERNARD, 'Turning blue sound into blue noise', *ICASSP-92,* Vol. 3, pp. 197-200.

James F. BLINN, 'The World of Digital Video', *IEEE Computer Graphics & Applications,* September 1992, pp. 106-112.

James F. BLINN, 'Quantization Error and Dithering', *IEEE Computer Graphics and Applications,* July 1994, pp. 78-82.

Fergus W. CAMPBELL, Lamberto MAFFEI, 'Contrast and Spatial Frequency', *Scientific American,* Vol. 231 No. 5, 1974, pp. 106-115.

CCITT, *Recomendation H.261 - Video Coding for Audiovisual Services at p x 64 kbits/s,* Geneva, August 1990.

Navin CHADDHA, Teresa H.Y. MENG, 'Psycho-Visual based Distortion Measures for Monochrome Image and Video Compression', *Proc. of 27th Asilomar Conf. on Signals, Systems, and Computers,* Vol. 2, IEEE Computer Society Press, November 1993, pp. 841-845.

W.K. CHAU, S.K.M. WONG, S.J. WAN, 'A Critical Analysis of Dithering Algorithms for Image Processing, *IEEE Region 10 Conf. on Computer and Communication Systems,* September 1990, Hong Kong, pp. 309-312.

Jer-Sen CHEN, 'A Comparative Study of Digital Halftoning Techniques', *Proc. of IEEE 1992 National Aerospace and Electronics Conference NAECON,* Vol 3, pp. 1139-1145.

K. CHEN, M. Afghahi, P.E. Danielsson, C. Svensson, 'PASIC: A Processor-A/D converter-Sensor Integrated Circuit', *Proc. 1990 IEEE Int. Symposium od Circuits and Systems,* 1990, Vol. 3, pp. 1705-1708.

Charles CHIEN, Paul YANG, Etan COHEN, Rajeev JAIN, Henry SAMUELI, 'A 12.7 Mchip/s All-Digital BPSK Direct Sequence Spread-Spectrum IF Transceiver in 1.2 um CMOS', *1994 IEEE International Solid State Circuits Conference (ISSCC94) Technical Digest,* San Fransisco, February 1994, p. 30.

Pamela C. COSMAN, Karen L. OEHLER, Eve A. RISKIN, Robert M. GRAY, 'Using Vector Quantization for Image Processing', *Proc. of the IEEE*, Vol. 81 No. 9, pp. 1326-1341.

Pamela C. COSMAN, Robert M. GRAY, Richard A. OLSHEN, 'Evaluating Quality of Compressed Medical Images: SNR, Subjective Rating, and Diagnostic Accuracy', *Proc. of the IEEE*, Vol. 82 No. 6 (June 1994), pp. 919-932.

M.C. ESHER, 'Still Life with a Street', see (for example): *Life and Works of Escher*, Parragon (Bristol), 1995, ISBN 0-7525-1175-0, p. 21.

Robert FLOYD, Louis STEINBERG, 'An Adaptive Algorithm for Spatial Grey Scale', *SID Int. Symposium 1975 Digest of Technical Papers*, pp. 36-37.

David FOX, 'Maniac Compression', *Personal Computer World*, Vol. October 1994, pp. 342-345.

J. P. FRISBY, *Seeing: Illusion, Brain and Mind*, Oxford University Press, 1979, ISBN 0-19-217672-2, p. 26.

Jean-loup GAILLY, *Compression-FAQ*, ftp://rtfm.mit.edu/pub/usenet/news.answers/compression-faq/part[1-3], 1995.

Lance A. GLASSER, Daniel W. DOBBERPUHL, *The Design and Analysis of VLSI Circuits*, Addison-Wesley (Reading, Massachusetts), 1985, ISBN 0-201-12580-3, p. 57.

Rafael C. GONZALEZ, Richard E. WOODS, *Digital Image Processing*, Addison-Wesley (Reading, Massachusetts), third edition, 1992, ISBN 0-201-50803-6, pp. 248-349.

W.M. GOODHALL, 'Television by pulse code modulation', *Bell Systems Technical Journal*, Vol. 30, 1951, pp. 33-49.

Douglas J. GRANRATH, 'The Role of Human Visual Models in Image Processing', *Proc. of the IEEE*, Vol. 69 No. 5 (May 1981), pp. 552-561.

Charles F. HALL, Ernest L. HALL, 'A Nonlinear Model for the Spatial Charactersitics of the Human Visual System', *IEEE Trans. on Systems, Man, and Cybernetcis*, Vol. SMC-7 No. 3 (March 1977), pp. 161-170.

Fred HALSALL, *Data communications, computer networks and OSI.*, Addison-Wesley (Wokingham, England), second edition, 1988, ISBN 0-201-18244-0.

R. W. HAMMING, *Coding and Information Theory*, Prentice-Hall (Englewood Cliffs, New Jersey), 1980, ISBN 0-13-139139-9.

Paul HECKBERT, 'Color Image Quantisation for Frame Buffer Display', *Computer Graphics*, Vol. 16 No. 3 (July 1982), pp. 297-307.

P. HOROWITZ, W. HILL, *The Art of Electronics*, second edition, Cambridge University Press (Cambridge), 1990, pp. 415-416.

David A. HUFFMAN, 'A Method for the Construction of Minimum-Redundancy Codes', *Proc. of the IRE*, Vol. 40 September 1952, pp. 1098-1101.

L.P. HURD, M.A. GUSTAVUS, M.F. BARNSLEY, 'Fractal Video Compression', *Technical Digest of COMPCON Spring 92*, 1992, pp. 41-42.

Keith JACK, *Video Demystified: a handbook for the digital engineer*, Hightext (Solana Beach,CA), 1993, ISBN 1-878707-09-4, p. 331.

Anil K. JAIN, 'Image Data Compression: A Review', *Proc. of the IEEE*, Vol. 69 No.3 (March 81), pp. 349-389.

Arnaud E. JACQUIN, 'Image Coding Based on a Fractal Theory of Iterated Contractive Image Transformations', *IEEE Trans. on Image Processing,* Vol. 1 No. 1, pp. 18-30.

JARVIS, JUDICE, NINKE, 'A survey of techniques for the display of continuous tone pictures on bilevel displays', *Computer Graphics and Image Processing Proc. 5,* 1 March 1976, pp. 13-40.

Shanika KARUNASEKERA, Nick KINGSBURY, 'A Distortion Measure for Blocking Artifacts in Images Based on Human Visual Sensitivity', *IEEE Trans. on Image Processing,*Vol. 4 No. 6 (June 1995), pp. 713-724.

D. KNUTH, 'Digital Halftones by Dot Diffusion', *ACM Trans. on Graphics,* Vol. 6 No. 4 (October 1987), pp. 245-273.

Herbert L. KRAUSS, Charles W. BOSTIAN, Frederick H. RAAB, *Solid State Radio Engineering,* John Wiley & Sons (New York), 1980, ISBN 0-471-03018-X.

W.H. KRUSKAL, W.A. WALLIS, 'Use of ranks in one-crtierion variance analysis', *Journal of the American Statistical Association,* No. 47, 1952, pp. 583-621.

Frank X. LUKAS, Zigmantas L. BUDRIKIS, 'Picture Quality Prediction Based on a Visual Model', *IEEE Trans. on Commumincations,* Vol. COM-30 No. 7 (July 1982), pp. 1679-92.

James L. MANNOS, David J. SAKRISON, 'The Effects of a Visual Fidelity Criterion on the Encoding of Images', *IEEE Trans. on Information Theory,* Vol IT-20 No. 4 (July 1984), pp. 525-536.

D.C. MARR, *Vision,* W.H. Freeman (San Fransisco, CA), 1982, ISBN 0-7167-1567-8.

S.K. MENDIS, S.E. KEMENY, E.R. FOSSOM, 'A 128*128 CMOS active pixel image sensor for highly inegrated imaging systems', *IEDM 1993 Technical Disgest,* pp. 583-586.

Theophano MITSA, Kevin J. PARKER, 'Digital halftoning technique using a blue-noise mask, *Journal of the Optical Society of America,* Vol. 9 No. 11 (November 1992), pp. 1920-1929.

J.B. MULLIGAN, A.J. AHUMADA, 'Principled Methods for Color Dithering based on Models of the Human Visual System', *SID Int. Symposium Digest of Technical Papers,* 1992, pp. 194-197.

Jeffrey B. MULLIGAN, 'Improving Digital Halftones by Exploiting Visual System Properties', *Proc. of the 27th Asilomer Conf. on Signals Systems and Computers,* November 1993, pp. 961-965.

Nasser M. NASRABADI, Robert A. KING, 'Image Coding Using Vector Quantisation: A Review', *IEEE Trans. on Communications,* Vol. 36 No. 8 (August 1988), pp. 957-971.

J. NIETROJ, W. ZAPSKY, H. LANG, 'Cost-Attractive, Reliable Remote Controls Use SAW Resonators', *Siemens Components*, No. 4, 1990, pp. 142-145.

D. E. PEARSON, *Transmission and Display of Pictorial Information*, Pentech Press Limited (London), 1975.

Raymond L. PICKHOLTZ, Donald L. SHILLING, Laurance B. MILSTIEN, 'Theory of Spread-Spectrum Communications - A Turorial', *IEEE Trans. on Communications*, Vol COM-30 No. 5 (May 1982), pp. 855-884.

Charles A. POYNTON, *A Technical Intorduction to Digital Video*, John Wiley & Sons (New York), 1996, ISBN 0-471-12253-X.

William K. PRATT, Julius KANE, Harry C. ANDREWS, 'Hadamard Transform Image Coding', *Proc. of the IEEE*, Vol. 57 No. 1 (January 1969), pp. 58-70.

William K. PRATT (editor), *Image Transmission Techiques*, Academic Press (New York), 1979, ISBN 0-12-014572-3, pp. 73-113.

William H. PRESS, Saul A. TEUKOISKY, Willain T. VETTERLING, Brian P. FLANNERY, *Numerical Recipies in C: The Art of Scientific Computing*, 2nd edition, Cambridge University Press (Cambridge), 1992, ISBN 0-521-43108.

RA (Radiocommunications Agency), *MPT 1340 Performance Specification: Transmitter and Receivers for use in the telemetry, telecommand in-building security equipment operating in the frequency band 417.90-418.10 MHz*, revised edition, October 1989.

RA (Radiocommunications Agency), *MPT 1336 Performance Specification: Low power transmitters and receivers for use in the VHF bands 36.61-36.79 MHz and 37.01-37.19 MHz*, revised edition, July 1992.

T.R. REED, V.R. ALGAZI, G.E. FORD, I. HUSSAIN, 'Perceptually based coding of monochrome and color still images', *Proc. of DCC '92*, March 1992, pp. 142-151.

D. RENSHAW, P.B. DENYER, G. WANG, M. LU, "ASIC Image Sensors", *Proc. IEEE ISCAS 90*, New Orleans, May 1990, pp. 3038-3041.

L.G. ROBERTS, 'Picture Coding using Pseudeo-Random Noise', *IRE Trans. on Information Theory*, Vol IT-8, 1962, pp. 145-154.

J. G. ROBSON, 'Spatial and Temporal Contrast-Sensitivity Functions of the Visual System', *Journal of the Optical Society of America*, Vol. 56 August 1966, pp. 1141-2.

John C. RUSS, *The Image Processing Handbook*, 2nd edition, CRC Press (Boca Raton, Florida), 1995, ISBN 0-8493-2516-1, pp. 21-22.

David J. SAKRISON, 'On the Role of the Observer and a Distortion Measure in Image Transmission', *IEEE Trans. on Communication*, Vol. COM-25 No. 11 (November 1977), pp. 1251-1267.

Manfred R. SCHROEDER, 'Images from computers', *IEEE spectrum*, Vol 6 March 1969, pp. 66-78.

Peter SEITZ, Graham K. LANG, 'A Practical Adaptive Image Compression Technique Using Visual Criteria for Still-Picture Transmission with Electronic Mail', *IEEE Trans. on Communications*, Vol. 38 No. 7 (July 1990), pp. 947-949.

Claude E. SHANNON, Warren WEAVER, *The Mathematical Theory of Communication*, The University of Illinois Press (Urbana, Illinois), 1963, 0-252-72548-4.

Jack SMITH, *Modern Communication Circuits*, McGraw-Hill (Singapore), 1986, ISBN 0-07-058730-2, pp. 455-502 (chap. 12).

Mark SOMMERS, illustration of Thomas Paine, *WIRED (UK Edition)*, ISSN 1357-0978, Vol. 1 No. 1 (April 1995), p. 65.

Jon M. STERN, Peter A. IVEY, Steven P. LARCOMBE, N. John GODDENOUGH, N. Luke Seed, Andrew J. SHELLY, 'An Ultra Compact, Low-Cost, Complete Image-Processing System', *1995 IEEE International Solid-State Circuits Conference (ISSCC95) Technical Digest*, San Fransisco, February 1995, pp. 230-231.

J. C. STOFFEL, J. F. MORELAND, 'A survey of Electronic Techniques for Pictorial Image Reproduction', *IEEE Trans. of Communications*, Vol COM-29 No. 12 (December 1981), pp. 1898-1925.

S.M. SZE, *VLSI Technology*, second edition, McGraw-Hill (New York), 1988, ISBN 0-07-100347, pp. 616-617.

T.S.D. TSUI and T.G. CLARKSON, 'Spread-spectrum communication techniques', *Electronics & Communication Engineering Journal*, February 1994, pp. 3-12.

Robert ULICHNEY, *Digital Halftoning*, MIT Press (Cambridge Massachusetts), 1987, ISBN 0-262-21009-6.

Robert A. ULICHNEY, 'Dithering with Blue Noise', *Proc. of the IEEE*, Vol. 76 No. 1 (January 1988), pp. 56-79.

L. VELHO, J.M. GOMES, 'Digital Halftoning with Space Filling Curves', *SIGRAPH 91*, Vol. 25 No. 4 (1991), pp. 81-90.

Oliver VELLACOTT, 'CMOS in camera', *IEE Review*, May 1994, pp. 111-114.

Gregory WALLACE, Toy VIVIAN, Henning POULSEN, 'Subjective testing results for still picture compression algorithms for international stadardization' *Proc. GLOBECOM '88*, Vol. 2, 1988, pp. 1022-1027.

Terry A. WELSH, 'A Technique for High-Performance Data Compression', *Computer*, Vol. 17 No. 6 (June 1984), pp. 8-19.

Ross N. WILLIAMS, *A Painless Guide to CRC Error Detection Algorithms*, ftp://ftp.rocksoft/com/clients/rocksoft/papers/crc_v3.txt, 1993.

Ian H. WITTEN, Radford M. NEAL, 'Using Peano Curves for Bilevel Display of Continuous-Tone Images' *IEEE Computer Graphics and Applications*, May 1982, pp. 47-52.

Zhenhua XIE, Thomas G. STOCKMAN, 'Towards the Unification of Threee Visual Laws and Two Models in Brightness Perception', *IEEE Trans. on Systems, Man, and Cybernetics*, Vol. 19 No. 2 (March/April 1989), pp. 379-387.

R.B. YATES, P.A. IVEY, 'Approches to Image Data Compression for Video Coding', *Proc. IEE Colloquium on Low Bit Rate Image Coding*, Digest No. 75, IEE (London), 1995, pp. 10/1-10/5.

Paul YOUNG, *Electronic Communication Techniques*, third edition, Macmillan, 1994, pp. 534-539.

Jacob ZIV, Abraham LEMPEL, 'A Universal Algorithm for Sequential Data Compression', *IEEE Trans. on Information Theory,* Vol IT-23 No. 3 (May 1977), pp. 337-343.

appendix one

Simulation software source code

The most important parts of the simulation software source code are listed below. The three C files listed are as follows:

- main.c - the main function
- simulate.c - called by main() to control the simulation environment.
- quantise.c - contains all the error diffusion based compression functions.

Main.c

This functions sets up the general program environment. Command line flags can be used to start it in simulate, record, demo or test modes.

```
/*############################################################################
 *
 * main.c                          Andrew Murray July 95
 *
 * The program can be started in one of three modes:
 * (1) simulation mode (the default mode) this is the original version of
 *     the software which performs variable depth quantisation with or
 *     without various types of diffusion, either on images from file or
 *     from live video input (via a 'PC Card Camera'). A limited amount
 *     of block-based DPCM code is also included.
 * (2) demo mode (envoked using the -d flag) this mode is used to give self-
 *     running demos of different types of diffusion. It was written
 *     primarily for collection of subjective test results.
 * (3) recording mode (envoked using -r) used for the recording of stills
 *     or sequences, mainly used for creating the demos.
 *
 * All the code is written and complied for DOS, using the Microsoft C/C++
 * Complier v8.00. The graphics routines use VESA mode 105h (1024x786 by 256
 * colours) and require a graphics card (and driver - eg. univesa.exe) that
 * support this mode.
 *
 *############################################################################*/

#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>

#include "vvl_defs.h"
#include "array.h"
#include "camlib.h"
#include "pcmcia.h"
#include "display.h"
#include "simulate.h"
#include "demo.h"
#include "record.h"
#include "input.h"
#include "test.h"
```

```
#include "cmd_line.h"

/* mode definitions */
#define SIMULATE 0
#define DEMO 1
#define RECORD 2
#define TEST 3
#define CMD_LINE 4

/* Forward Declaration of Private Functions */
int DealWithCommandLineFlags( int argc, char *argv[], int *pMode );

/* Global Variable Declarations */
colour_triplet green={ 158, 80, 25 };
colour_triplet grey={ 195, 131, 51 }; /* new */
colour_triplet purple={ 107, 67, 65 };
colour_triplet blue={ /*147*/218, /*67*/74, /*25*/38 };
colour_triplet on_green={ 223, 89, 84 }, off_green={ 154, 77, 76 };
colour_triplet on_red={ 251, 201, 160 }, off_red={ 210, 105, 96 };
colour_triplet dull_red={ 210, 169, 137 };
colour_triplet win95grey = { 1, 2, 4 };

byte *gpFrame0, *gpFrame1, *gpFrame2; /* global pointers to the frame stores*/
int g_frame0_state=FALSE, g_frame1_state=FALSE, g_frame2_state=FALSE, g_ignore=0;
dword frame_rows = FRAME_ROWS, frame_cols=FRAME_COLS; /* the frame dimensions*/
int g_pcmcia = FALSE, g_slot=FALSE, g_bailout = FALSE;
int g_graphics=FALSE, g_diagnose=FALSE, g_logo = TRUE;
CAMERA camera_A;        /* structure containing PCMCIA card details */
char buffer[80];        /* temp. character buffer used mainly for sprintf's */
dword g_histogram[256], gamma_table[256];
float g_gamma=1.4F;
demo_page_details g_page_array[DEMO_MAX], tempDemo;

/* #### *
 * main *
 * #### */

void __cdecl main ( int argc, char *argv[] )
{
   int mode=SIMULATE, rtn=TRUE, user_input=FALSE;

   rtn = DealWithCommandLineFlags( argc, argv, &mode );
   if (!rtn) goto error;

   switch (mode) {
   case SIMULATE:
      rtn = Simulate();
      if (!rtn) goto error;
      break;
   case DEMO:
      rtn = Demo();
      if (!rtn) goto error;
      break;
   case TEST:
      rtn = Test();
      if (!rtn) goto error;
      break;
   case RECORD:
      rtn = Record();
      if (!rtn) goto error;
      break;
   case CMD_LINE:
      rtn = CommandLineMode( argv, argc );
      if (!rtn) goto error;
      break;
   default:
      printf(" main: invalid 'mode' value\n");
      goto error;
   }

   if (g_frame0_state) _hfree( gpFrame0 );
   if (g_frame1_state) _hfree( gpFrame1 );
   if (g_frame2_state) _hfree( gpFrame2 );
   printf("\n main: exiting normally :)\n");
   exit (0);

error:
   if (g_frame0_state) _hfree( gpFrame0 );
```

```
    if (g_frame1_state) _hfree( gpFrame1 );
    if (g_frame2_state) _hfree( gpFrame2 );
    printf("\n%s", buffer );
    printf("\n main: exiting on error :(\n");
    exit (0);
}

/* ############################## *
 * Private Function Declarations *
 * ############################## */

/****************************************************************************
 *                          NAME:  DealWithCommandLineFlags

 * PURPOSE:  takes action on any command line flag that were given at the
 *           command line when the program was executed.
 ****************************************************************************/
int DealWithCommandLineFlags( int argc, char *argv[], int *pMode )
{
    char tmp;
    int count;

    printf("\n");
    if (argc!=1){
        for (count=1; count<argc; count++){
            tmp = argv[count][0];
            if (/*argv[count][0]*/(tmp == '/')||(tmp == '-')) {
                tmp = argv[count][1];
                switch (/*argv[count][1]*/tmp) {
                case 'd':
                    printf(" Demo mode envoked\n");
                    *pMode = DEMO;
                    break;
                case 'r':
                    printf(" Recording mode envoked\n");
                    *pMode = RECORD;
                    break;
                case 'x':
                    printf(" Diagnose set TRUE\n");
                    g_diagnose = TRUE;
                    g_ignore++;
                    break;
                case 'l':
                    printf(" Logo suppressed :(\n");
                    g_logo = FALSE;
                    break;
                case 't':
                    printf(" Test mode envoked ...\n");
                    *pMode = TEST;
                    break;
                case 'c':
                    printf(" command line mode envoked ...\n");
                    *pMode = CMD_LINE;
                    return TRUE;
                    break;
                default:
                    printf(" %c - unrecognised flag\n", tmp);
                    goto explain;
                }
            }
            else {
                printf(" %s - unrecognised argument\n", argv[count]);
                goto explain;
            }
        }
    }
    return TRUE;

explain:
    printf("\n Usage: diffuse [options]\n");
    printf(" -d envoke Demo mode\n");
    printf(" -r envoke Record mode\n");
    printf(" -x run with Diagnosis");
    return FALSE;
}

/* ########## *
 * main.c end *
 * ########## */
```

Simulate.c

This file contains the umbrella function that controls the flow of processing during software simulation.

```
/*#########################################################################
 *
 * simulate.c                  - Andrew Murray February 95
 *
 * This is a piece of code that demonstrates all the features of the software
 * simulation of compression, quantisation, error diffusion and palette
 * optimisation created during my PhD.
 *
 * The program can either take input from '.img' files or live from a 'PC Card
 * Camera' PCMCIA card.
 *
 *#########################################################################*/

#include <conio.h>
#include <stdio.h>
#include <graph.h>
#include <math.h>
#include <malloc.h>
#include "vvl_defs.h"
#include "simulate.h"
#include "display.h"
#include "trio.h"
#include "array.h"
#include "camlib.h"
#include "pcmcia.h"
#include "input.h"
#include "fileio.h"
#include "quantise.h" /* for GenFloydSTest() and Prescale() */

/* Forward Declarations of Private Functions */
int SetupSimulateDosGraphics( byte active );
void GenerateTestFrame( byte *pFrame );
int InterpretKeypress(char letter, byte *pActive, byte *pSave_win, int *pSave_output );
void LoadSequence();
void SaveSequence( int *pSave_out, byte *pSave_win );
void ToggleProcess( trio *pTrio, int requested_process, char *label );
void ToggleAnalysis( trio *pTrio, int requested_analysis, char *label );
int Simulate_ReadWindows( );

/* External Variable Declarations */
extern colour_triplet green, grey, purple, blue, win95grey;
extern colour_triplet on_green, off_green;
extern colour_triplet on_red, off_red, dull_red;
extern byte *gpFrame0, *gpFrame1, *gpFrame2; /* global pointers to the frame stores*/
extern int g_frame0_state, g_frame1_state, g_frame2_state;
extern dword frame_rows, frame_cols; /* the frame dimensions*/
extern int g_pcmcia, g_slot, g_bailout, g_diagnose, g_logo;
extern CAMERA camera_A;       /* structure containing PCMCIA card details */
extern char buffer[80];       /* temp. character buffer used mainly for sprintf's */
extern dword g_histogram[256];
extern dword gamma_table[256];
extern float g_gamma;
extern int g_graphics;

/* Global variable definitions */
XY pcmcia_led = {30, 610}, file_led = {30, 626}, load_led = {30, 658}, g_save_led = {30, 706};
trio *pTrio_array[3]; /* an array of pointers to the trios*/
char gSavename[13]="saved.tst", gLoadname[13]="hamster.img";
int g_save_analysis=FALSE, g_save_image=FALSE;

/* ############################ *
 * public function declarations *
 * ############################ */

/*************************************************************************
 *                          NAME: Simulate
 * PURPOSE: This is the overall simulation function. Once called it will
 *          perform it goes into a loop of updating images that are 'live'
 * and checking for user input.
 ***************************************************************************/
int Simulate()
{
   int rtn=TRUE, user_input=FALSE;   /* flags */
   byte save_win=0;
   int save_output=TRUE;
   char tmp='x'; /* used for storing 'keypresses'*/
   byte i, active_trio=0;
   trio left_trio    = Trio_Init( "left.dat" );        /* the three trios*/
   trio middle_trio     = Trio_Init( "middle.dat" );
   trio right_trio   = Trio_Init( "right.dat" );

   printf("\n Simulate: called...\n");

   /* set up the frame store */
   gpFrame0 = (byte __huge *)_halloc( (frame_rows*frame_cols), sizeof(byte) );
   if (gpFrame0 == NULL) printf(" failed to _halloc for frame0\n");
   else {
      printf(" halloc'd frame0 okay :)\n");
      g_frame0_state=TRUE;
   }
   gpFrame1 = (byte __huge *)_halloc( (frame_rows*frame_cols), sizeof(byte) );
   if (gpFrame1 == NULL) printf(" failed to _halloc for frame1\n");
   else {
      printf(" halloc'd frame1 okay :)\n");
      g_frame1_state=TRUE;
   }
   gpFrame2 = (byte __huge *)_halloc( (frame_rows*frame_cols), sizeof(byte) );
   if (gpFrame2 == NULL) printf(" failed to _halloc for frame2\n");
   else {
      printf(" halloc'd frame2 okay :)\n");
      g_frame2_state=TRUE;
   }

   if (g_bailout) {
      sprintf( buffer, " Simulate: error initialising the trios\n");
      goto sim_error;
   }

   /*## initialise the pointers to the trios ##*/
   pTrio_array[0] = &left_trio;
   pTrio_array[1] = &middle_trio;
   pTrio_array[2] = &right_trio;

   g_slot = Pcmcia_SetupCamera( &camera_A );
   if (g_slot) g_pcmcia = TRUE;

   if (g_diagnose){
      printf(" Simulate: Diagnostics...\n");
      printf(" Simulate: g_slot=");
      if (g_slot) printf("TRUE");
      else printf("FALSE");
      printf(" and g_pcmcia=");
      if (g_pcmcia) printf("TRUE\n");
      else printf("FALSE\n\n");
      for (i=0; i<3; i++) Trio_PrintContents( pTrio_array[i] );
      Input_WaitForKey( NULL );
   }

   rtn = SetupSimulateDosGraphics( active_trio );
   if (!rtn) goto sim_error;

   while (!g_bailout) {
      while (!user_input) {
         if ( g_slot && g_pcmcia ) {
            Pcmcia_GrabFrame( &camera_A, gpFrame0, 0, 0 );
         }

         for (i=0; i<3; i++)
            if (pTrio_array[i]->live==TRUE) {
               Trio_UpdateImages( pTrio_array[i] );
            }

         user_input = Input_CheckForKey( &tmp );
      }
      g_bailout = InterpretKeypress( tmp, &active_trio, &save_win, &save_output );
      user_input=FALSE;
   }
   Display_EndGraphics();
   return TRUE;

sim_error:
   if (g_graphics) Display_EndGraphics();
   return FALSE;
}

/* ############################ *
 * Private Function Declarations *
 * ############################ */

/***************************************************************************
 *                          NAME:  SetupSimulateDosGraphics

 *  PURPOSE:  called from within Simulate() to set the graphics mode and
 *            palette and draw all the window borders.
 ***************************************************************************/
int SetupSimulateDosGraphics( byte active )
{
   byte i;
   int ret;
   window temp;
   XY logoPos = {34,34};

   ret = Display_SetupGraphics();
   if (!ret) {
      sprintf( buffer, " Simulate_SetupDosGraphics: graphics mode change failed" );
      return(FALSE);
   }
   Display_CreateGammaPalette( 1 );
   Display_CreateSpreadGreyPalette();
   Display_CreateSpreadGreyGammaPalette( g_gamma );

   if (g_logo) {
      FileIO_LoadFrame( "logo.img", gpFrame0 );
      Array_CreateLogo( gpFrame0 );
      Display_ColourFrame( gpFrame0, logoPos );
   }
   GenerateTestFrame( gpFrame0 );

   for (i=0; i<3; i++) {
      Display_Window( &(pTrio_array[i]->image) );
      Display_Window( &(pTrio_array[i]->analysis_win) );
      Display_Window( &(pTrio_array[i]->status) );
      Trio_DrawLegends( pTrio_array[i] );
      Trio_RefreshLeds( pTrio_array[i] );
   }
   Trio_RedrawControlWindow2( pTrio_array[active], DARK_GREEN );

   temp.height = 147; temp.width = 140; temp.org.x = 25; temp.org.y = 595;
   temp.title_depth = 18; temp.text = TRUE; temp.title_row = 37;
   temp.title_col1 = 5; temp.title_col2 = 19;
   sprintf( temp.title, "Input/Output" ); temp.title_colour = DARK_TEXT;
   temp.col1 = 7; temp.col2 = 19; temp.row1 = 39; temp.row2 = 45;
   temp.shade = win95grey;

   Display_Window( &temp ); Display_SetTextWindow( &temp );
   Display_GraphicalText( "PCMCIA input\n", DARK_TEXT );
   Display_GraphicalText( "File input\n\n", DARK_TEXT );
   Display_GraphicalText( "Load image\n\n\n", DARK_TEXT );
   Display_GraphicalText( "Save image", DARK_TEXT );
   Display_Led( pcmcia_led.x, pcmcia_led.y, on_green, off_green, g_pcmcia );
   Display_Led( file_led.x, file_led.y, on_green, off_green, FALSE );
   Display_Led( load_led.x, load_led.y, on_red, off_red, FALSE );
   Display_Led( g_save_led.x, g_save_led.y, on_red, off_red, FALSE );

  return TRUE;
}

/***************************************************************************
 *                          NAME: GenerateTestFrame
 *  PURPOSE: Generates a test picture in a plane of the array
 ***************************************************************************/
void GenerateTestFrame( byte *pFrame )
{
   dword index, end_of_frame=(dword)(FRAME_ROWS*FRAME_COLS);
```

```
    for( index=0; index<end_of_frame; index++)
        *pFrame++ = (byte)(index/256);
}

/************************************************************************
 *                             NAME:  InterpretKeypress
 
 *  PURPOSE:  takes action on key presses, after a key press has been
 *            detected this function should be called (passing the key pressed)
 ************************************************************************/
int InterpretKeypress(char letter, byte *pActive, byte *pSave_win, int *pSave_output )
{
    trio temp_trio=*pTrio_array[*pActive];
    XY save_ind;

    switch (letter) {
    case ',':
        if(*pActive>0){
            Trio_RedrawControlWindow2( pTrio_array[*pActive], WINGREY );
            temp_trio = *pTrio_array[--(*pActive)];
            Trio_RedrawControlWindow2( pTrio_array[*pActive], DARK_GREEN );
        }
        break;
    case '.':
        if(*pActive<2) {
            Trio_RedrawControlWindow2( pTrio_array[*pActive], WINGREY );
            temp_trio = *pTrio_array[++(*pActive)];
            Trio_RedrawControlWindow2( pTrio_array[*pActive], DARK_GREEN );
        }
        break;
    case '1':
    case '2':
    case '3':
    case '4':
    case '5':
    case '6':
        temp_trio.bits = (byte)( (byte)letter - 48 );
        break;
    case 'a':
        if (temp_trio.live){
            temp_trio.live=FALSE;
            temp_trio.image.title_colour = INACTIVE_TITLE;
            Display_WindowTitle( &(temp_trio.image) );
            temp_trio.analysis_win.title_colour = INACTIVE_TITLE;
            Display_WindowTitle( &(temp_trio.analysis_win) );
        }
        else {
            temp_trio.live=TRUE;
            temp_trio.image.title_colour = ACTIVE_TITLE;
            Display_WindowTitle( &(temp_trio.image) );
            temp_trio.analysis_win.title_colour = ACTIVE_TITLE;
            Display_WindowTitle( &(temp_trio.analysis_win) );
        }
        break;
    case 'c': ToggleProcess( &temp_trio, NONE, NONE_STRING );        break;
    case 't': ToggleProcess( &temp_trio, TRUNCATE, TRUNCATE_STRING ); break;
    case 'i': ToggleProcess( &temp_trio, SIMPLE_DF, SIMPLE_DF_STRING ); break;
    case 'f': ToggleProcess( &temp_trio, FLOYD_S, FLOYD_S_STRING ); break;
    case 'r': ToggleProcess( &temp_trio, RANDOM, RANDOM_STRING ); break;
    case 'e': ToggleProcess( &temp_trio, SAFE_1, SAFE_1_STRING ); break;
    case 'w': ToggleProcess( &temp_trio, SAFE_2, SAFE_2_STRING ); break;
    case 'y': ToggleProcess( &temp_trio, OPTIMISE, OPTIMISE_STRING ); break;
    case 'u': ToggleProcess( &temp_trio, OPTIMISE_ED, OPTIMISE_ED_STRING ); break;
    case 'h': ToggleAnalysis( &temp_trio, HISTOGRAM, HISTOGRAM_STRING ); break;
    case 'd': ToggleAnalysis( &temp_trio, ERROR, ERROR_STRING );     break;
    case 'o': ToggleAnalysis( &temp_trio, SQR_ERROR, SQR_ERROR_STRING ); break;
    case 'g': ToggleAnalysis( &temp_trio, DEBUG, DEBUG_STRING );     break;
    case 'l': LoadSequence(); break;
    case 'z':
        Display_Led( load_led.x, load_led.y, on_red, off_red, TRUE );
        FileIO_Load360x288PgmFrame ( "gs000.pgm", gpFrame0, 27 );
        g_pcmcia = FALSE;
        Display_Led( load_led.x, load_led.y, on_red, off_red, FALSE );
        Display_Led( pcmcia_led.x, pcmcia_led.y, on_green, off_green, g_pcmcia );
        Display_Led( file_led.x, file_led.y, on_green, off_green, TRUE );
        break;
    case 'v':
         Display_Led( load_led.x, load_led.y, on_red, off_red, TRUE );
         FileIO_Load512PgmFrame( "peppers.pgm", gpFrame0 );
         g_pcmcia = FALSE;
         Display_Led( load_led.x, load_led.y, on_red, off_red, FALSE );
         Display_Led( pcmcia_led.x, pcmcia_led.y, on_green, off_green, g_pcmcia );
         Display_Led( file_led.x, file_led.y, on_green, off_green, TRUE );
         break;
      case 'x':
         Display_Led( g_save_led.x, g_save_led.y, on_red, off_red, TRUE );
         Array_WriteFrameToTextFile ( gpFrame0, "hamster.txt" );
         Display_Led( g_save_led.x, g_save_led.y, on_red, off_red, FALSE );
         break;
      case 's': SaveSequence( pSave_output, pSave_win ); break;
      case ';': FileIO_SavePCXFrame( gpFrame0, "test.pcx" ); break;
      case '=':
         if (temp_trio.save_image) temp_trio.save_image = FALSE;
         else temp_trio.save_image = TRUE;
         break;
      case '-':
         if (temp_trio.scale) temp_trio.scale = FALSE;
         else temp_trio.scale = TRUE;
         break;
      case 'p':
         if (g_pcmcia) g_pcmcia = FALSE;
         else {
            g_pcmcia = TRUE;
            Display_Led( file_led.x, file_led.y, on_green, off_green, FALSE );
         }
         Display_Led( pcmcia_led.x, pcmcia_led.y, on_green, off_green, g_pcmcia );
         break;
      case '[':
         if (g_gamma>0.1F) g_gamma -= 0.1F;
         Display_CreateSpreadGreyGammaPalette( g_gamma );
         break;
      case ']':
         g_gamma += 0.1F;
         Display_CreateSpreadGreyGammaPalette( g_gamma );
         break;
      case '9':
         if(*pSave_win>0){
            if (*pSave_output) {
               save_ind =  pTrio_array[*pSave_win]->image.org;
               Display_BlankLed( (short)(save_ind.x+240), (short)(save_ind.y-17), WINGREY);
               (*pSave_win)--;
               save_ind =  pTrio_array[*pSave_win]->image.org;
               Display_Led( (short)(save_ind.x+240), (short)(save_ind.y-17), on_red, off_red,
FALSE);
            }
            else {
               save_ind =  pTrio_array[*pSave_win]->analysis_win.org;
               Display_BlankLed( (short)(save_ind.x+240), (short)(save_ind.y-17), WINGREY);
               (*pSave_win)--;
               save_ind =  pTrio_array[*pSave_win]->analysis_win.org;
               Display_Led( (short)(save_ind.x+240), (short)(save_ind.y-17), on_red, off_red,
FALSE);
            }
         }
         break;
      case '0':
         if(*pSave_win<2){
            if (*pSave_output) {
               save_ind =  pTrio_array[*pSave_win]->image.org;
               Display_BlankLed( (short)(save_ind.x+240), (short)(save_ind.y-17), WINGREY);
               (*pSave_win)++;
               save_ind =  pTrio_array[*pSave_win]->image.org;
               Display_Led( (short)(save_ind.x+240), (short)(save_ind.y-17), on_red, off_red,
FALSE);
            }
            else {
               save_ind =  pTrio_array[*pSave_win]->analysis_win.org;
               Display_BlankLed( (short)(save_ind.x+240), (short)(save_ind.y-17), WINGREY);
               (*pSave_win)++;
               save_ind =  pTrio_array[*pSave_win]->analysis_win.org;
               Display_Led( (short)(save_ind.x+240), (short)(save_ind.y-17), on_red, off_red,
FALSE);
            }
         }
         break;
```

```
    case '7':
        if (*pSave_output) {
            *pSave_output = FALSE;
            save_ind =  pTrio_array[*pSave_win]->image.org;
            Display_BlankLed( (short)(save_ind.x+240), (short)(save_ind.y-17), WINGREY);
            save_ind =  pTrio_array[*pSave_win]->analysis_win.org;
            Display_Led( (short)(save_ind.x+240), (short)(save_ind.y-17), on_red, off_red, FALSE);
        }
        else {
            *pSave_output = TRUE;
            save_ind =  pTrio_array[*pSave_win]->analysis_win.org;
            Display_BlankLed( (short)(save_ind.x+240), (short)(save_ind.y-17), WINGREY);
            save_ind =  pTrio_array[*pSave_win]->image.org;
            Display_Led( (short)(save_ind.x+240), (short)(save_ind.y-17), on_red, off_red, FALSE);
        }
        break;
    case 'k':
        Array_CreateGammaTestImage( gpFrame0 );
        break;
    case ' ':
    case 'q':
        sprintf( buffer, " InterpretKeypress: user termination '%c'\n", letter);
        return TRUE;
        break;
    }

    *pTrio_array[*pActive]=temp_trio;
    Trio_RefreshLeds( pTrio_array[*pActive] );
    return FALSE;
}

/*************************************************************************
 *                         NAME:  LoadSequence
 *  PURPOSE:  Loads an image from a file, and alters the global variables
 *            and the LEDs accordingly
 *************************************************************************/
void LoadSequence()
{
    int rtn;
    char temp_str[13]="x";

    _settextwindow( 43, 7, 43, 19 );
    _settextcolor( MIDGREY_TEXT );
    sprintf( buffer, "%s?", &gLoadname ); _outtext( buffer );
    rtn = Input_GetFilename( temp_str );
    if (!rtn) sprintf( gLoadname, "%s", &temp_str );
    sprintf( buffer, "\n%s", &gLoadname ); _outtext( buffer );
    Display_Led( load_led.x, load_led.y, on_red, off_red, TRUE );
    rtn = FileIO_LoadFrame ( gLoadname, gpFrame0 );
    if (!rtn) return;
    g_pcmcia = FALSE;
    Display_Led( load_led.x, load_led.y, on_red, off_red, FALSE );
    Display_Led( pcmcia_led.x, pcmcia_led.y, on_green, off_green, g_pcmcia );
    Display_Led( file_led.x, file_led.y, on_green, off_green, TRUE );
}

/*************************************************************************
 *                         NAME:  SaveSequence
 *  PURPOSE:  Saves an image to a file, and alters the global variables
 *            and the Leds accordingly
 *************************************************************************/
void SaveSequence( int *pSave_prcd, byte *pSave_win )
{
    int rtn;
    char temp_str[13]="x";
    char ch;

    _settextwindow( 46, 7, 46, 19 );
    _settextcolor( MIDGREY_TEXT );

    sprintf( buffer, "analysis?" ); _outtext( buffer );
    Input_WaitForKey( &ch );
    if (ch=='y') g_save_analysis=TRUE;
    else g_save_image=TRUE;

    sprintf( buffer, "%s?", &gSavename ); _outtext( buffer );
    rtn = Input_GetFilename( temp_str );
    if (!rtn) sprintf( gSavename, "%s", &temp_str );
```

```
   sprintf( buffer, "\n%s", &gSavename ); _outtext( buffer );
}

/*****************************************************************************/
void ToggleProcess( trio *pTrio, int requested_process, char *label )
{
   if (pTrio->process == requested_process){
      pTrio->process = NONE;
      sprintf( pTrio->image.title, NONE_STRING );
   }
   else {
      pTrio->process = requested_process;
      sprintf( pTrio->image.title, label );
   }
   Display_WindowTitle( &(pTrio->image) );
}

/*****************************************************************************/
void ToggleAnalysis( trio *pTrio, int requested_analysis, char *label )
{
   if (pTrio->analysis == requested_analysis){
      pTrio->analysis = NONE;
      sprintf( pTrio->analysis_win.title, "          not live          " );
   }
   else {
      pTrio->analysis = requested_analysis;
      sprintf( pTrio->analysis_win.title, label );
   }
   Display_WindowTitle( &(pTrio->analysis_win) );
}

/* ############## *
 * simulate.c end *
 * ############## */
```

quantise.c

All the quantisation based compression functions are contained in this file.

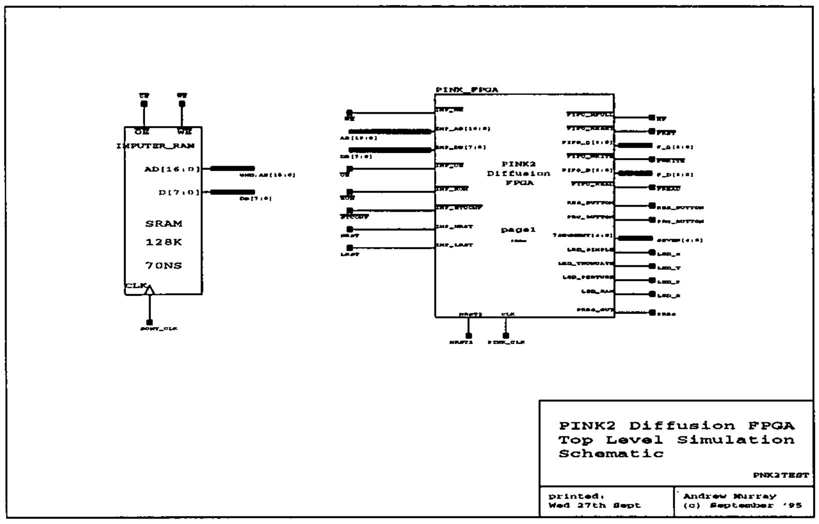
```
/*###########################################################################
 *
 * quantise.c                 - Andrew Murray February 95
 *
 * a library of routines to perform different types of quantisation.
 *
 *###########################################################################*/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "vvl_defs.h"
#include "quantise.h"
#include "array.h"
#include "input.h"  /* debug */
#include "display.h"

/* external declarations */
extern dword frame_rows, frame_cols;
extern int g_diagnose, g_graphics;

/* ########################### *
 * Public Function Declarations *
 * ########################### */


/*****************************************************************************
 *                          NAME:  Quantise_Truncate

 * PURPOSE:  reduces the pixel depth of the image to the number of 'bits'
 *           specified. The truncation is performed by ANDing each pixel
 *           with a mask - leaving the most significant bits in their
 *           original positions.
 *****************************************************************************/
void Quantise_Truncate( byte *pSource, byte *pDest, byte bits )
```

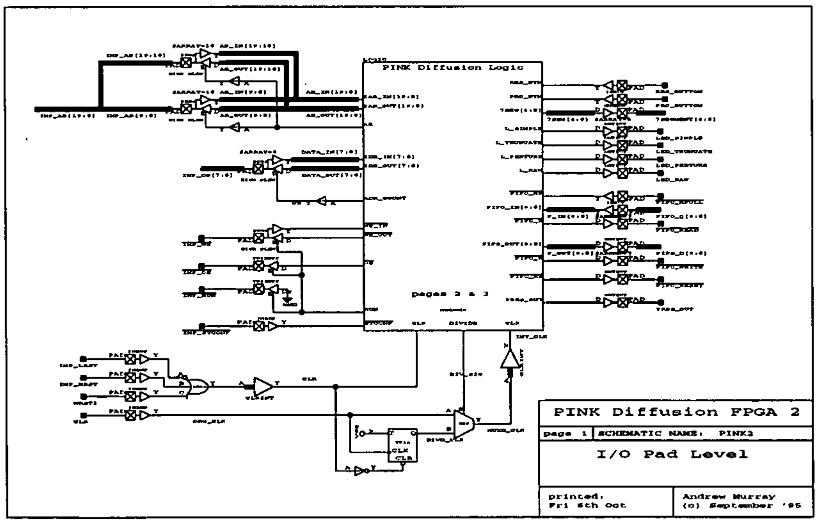
```
{
   dword pixel, end_of_frame=(dword)(frame_rows*frame_cols);
   byte mask=(byte)(0xff<<(8-bits));

   for(pixel=0; pixel<end_of_frame; pixel++)
      *pDest++ = (byte)(*pSource++ & mask);
   return;
}

/************************************************************************
 *                              NAME:  Quantise_Diffuse
 *  PURPOSE:  reduces the depth of an image, but takes account of the
 *            truncation errors by adding them onto yet untruncated pixels.
 *            This is the most simple example of such a scheme, where the
 *            whole error is added to the next pixel (a trap is included to
 *            make sure no pixels 'roll-round')
 ************************************************************************/
void Quantise_Diffuse( byte *src_ptr, byte *dest_ptr, byte bits )
{
   dword pixel, end_of_frame=(dword)(FRAME_ROWS*FRAME_COLS);
   byte sum=0, sum_mask=0xff, pixel_mask=0xff;

   sum_mask= (byte)(sum_mask>>bits);
   pixel_mask= (byte)(pixel_mask<<(8-bits));

   for(pixel=0; pixel<end_of_frame; pixel++, src_ptr++, dest_ptr++)
   {
      if (*src_ptr < pixel_mask) sum = (byte)( *src_ptr + (sum & sum_mask));
      else sum = *src_ptr;
      *dest_ptr = (byte)(sum & pixel_mask);
   }
}

/************************************************************************
 *                              NAME:  Quantise_RandDiffuse
 *  PURPOSE:  the same as QuanitseWithDiffusion except that the location
 *            where the error of the truncation is 'diffused' to is chosen
 *            randomly between two equally likely candidates (imeadiately to
 *            the right or below. A psuedo-random bit sequence generator is
 *            used to decide on the location. A high bit of the rand() result
 *            is used as they higher ones tend to be 'more' random!
 *            (cf. Numerical recipies in C).
 ************************************************************************/
void Quantise_RandDiffuse( byte *pOriginal, byte *pDest, byte bits )
{
   dword pixel_index, end_of_frame=(dword)(frame_rows*frame_cols);
   dword most_of_frame=(end_of_frame-1), last_line=(end_of_frame-frame_cols);
   int candidate_below;
   byte error_mask=0xff, pixel_mask=0xff, dummy;
   byte *pRaster=pDest, *pSpreadee=pDest, error=0;

   error_mask = (byte)(error_mask>>bits);
   pixel_mask = (byte)(pixel_mask<<(8-bits));

   Array_CopyFrame( pOriginal, pDest );

   for(pixel_index=0; pixel_index<most_of_frame; pixel_index++, pRaster++)
   {
      error = (byte)(*pRaster & error_mask);         /* calculate the impending truncation error
*/
      *pRaster = (byte)(*pRaster & pixel_mask);   /* truncate the pixel */

      candidate_below = ((rand()>>6) & 0x01); /* choose the 'spreadee' from the two candidates */
      if (candidate_below==FALSE) pSpreadee = (pRaster + 1);
      else {
         if (pixel_index<last_line) pSpreadee = (pRaster + frame_cols);
         else pSpreadee = &dummy;
      }

      if ( *pSpreadee < pixel_mask )
         *pSpreadee = (byte)(*pSpreadee + error);
   }
   *pRaster = (byte)(*pRaster & pixel_mask);      /* quanitse the last pixel without spreading */
}

/************************************************************************
 *                              NAME:  Quantise_SafeRandDiffuse1
 *  PURPOSE:  the same as the random one except that the candidate locations
 *          are flagged when they should no longer be added to, this limits
 *          the accumulated errors where by random lots of errors would
 *          otherwise have been spread.
 ***************************************************************************/
void Quantise_SafeRandDiffuse1( byte *pOriginal, byte *pDest, byte *pTemp, byte bits )
{
   dword pixel_index, end_of_frame=(dword)(frame_rows*frame_rows);
   dword most_of_frame=(dword)(end_of_frame-1), last_line=(end_of_frame-frame_rows);
   int candidate_below, spreading=TRUE;
   byte *pRaster=pDest, *pSpreadee=pDest;
   byte error=0;
   byte old_bit;
   long flag_gap = pTemp-pDest;  /* the distance between each pixel and it's flag */
   byte flag_mask = (byte)(0x01<<(8-bits));
   byte error_mask = (byte)(0xff>>bits);
   byte pixel_mask = (byte)(0xff<<(8-bits));
   byte *pRand = pTemp;/* for debug results*/

   Array_CopyFrame( pOriginal, pDest ); /* all the processing is done in the Dest frame store */
   Array_BlankFrame( pTemp, FALSE ); /* initialisation of all the flags */

   for(pixel_index=0; pixel_index<most_of_frame; pixel_index++, pRaster++, spreading=TRUE)
   {
      error = (byte)(*pRaster & error_mask);       /* calculate the impending quantisation error
*/
      *pRaster = (byte)(*pRaster & pixel_mask);   /* quantise the pixel */

      candidate_below = ((rand()>>6) & 0x01); /* make the initial random 'spreadee' selection */
      if (candidate_below==FALSE) pSpreadee = (pRaster + 1);
      else {
         if (pixel_index<last_line) pSpreadee = (pRaster + frame_cols);
         else spreading = FALSE;
      }
      if (*(pSpreadee+flag_gap)==TRUE) {
         if (pixel_index<last_line) pSpreadee = (pRaster + frame_cols);
         else spreading = FALSE;
      }
      if ((spreading==TRUE) && (*pSpreadee < pixel_mask))
      {
         old_bit = (byte)(*pSpreadee & flag_mask);
         *pSpreadee = (byte)(*pSpreadee + error);
         if ( (*pSpreadee & flag_mask) != old_bit ) *(pSpreadee+flag_gap) = TRUE;
      }

   }
   *pRaster = (byte)(*pRaster & pixel_mask);      /* quanitse the last pixel without spreading */
}

/***************************************************************************
 *                        NAME:  Quantise_SafeRandDiffuse2
 *  PURPOSE:  the same as SafeRand..1 except that when a flag is encountered
 *            spread is abandoned rather than always spreading to the pixel
 *            below.
 ***************************************************************************/
void Quantise_SafeRandDiffuse2( byte *pSource, byte *pDest, byte *pTemp, byte bits )
{
   dword pixel_index, end_of_frame=(dword)(frame_rows*frame_rows);
   dword most_of_frame=(dword)(end_of_frame-1), last_line=(end_of_frame-frame_rows);
   int candidate_below, spreading=TRUE;
   byte *pRaster=pDest, *pSpreadee=pDest;
   byte error=0;
   byte old_bit;
   long flag_gap = pTemp-pDest;  /* the distance between each pixel and it's flag */
   byte flag_mask = (byte)(0x01<<(8-bits));
   byte error_mask = (byte)(0xff>>bits);
   byte pixel_mask = (byte)(0xff<<(8-bits));
   byte *pRand = pTemp;/* for debug results */

   Array_CopyFrame( pSource, pDest );    /* all the processing is done in the Dest frame store */
   Array_BlankFrame( pTemp, FALSE ); /* initialisation of all the flags */

   for(pixel_index=0; pixel_index<most_of_frame; pixel_index++, pRaster++, spreading=TRUE) {
      error = (byte)(*pRaster & error_mask);         /* calculate the impending quantisation
error */
      *pRaster = (byte)(*pRaster & pixel_mask);      /* quantise the pixel */

      candidate_below = ((rand()>>6) & 0x01);/* make the initial random 'spreadee' selection */
      if (candidate_below==FALSE) pSpreadee = (pRaster + 1);
```

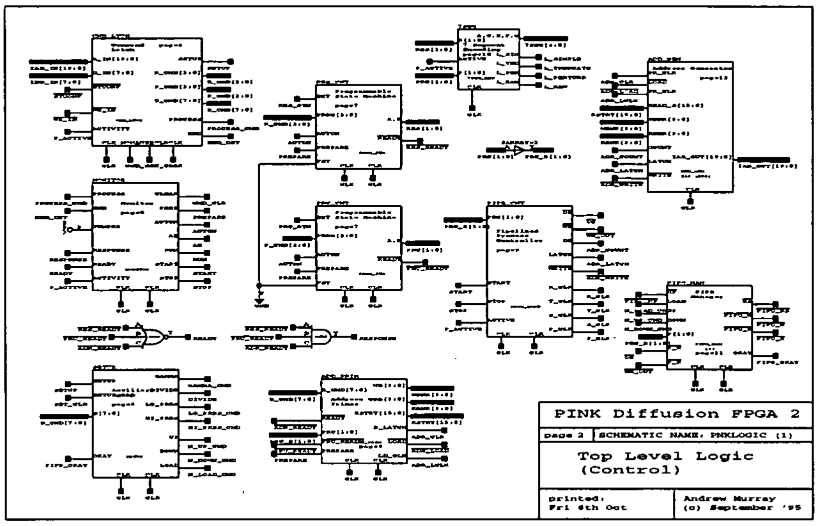
```
        else {
           if (pixel_index<last_line) pSpreadee = (pRaster + frame_cols);
           else spreading = FALSE; /* to stop 'spreading' outside the framestore */
        }
        if ((spreading==TRUE) && (*pSpreadee < pixel_mask) && (*(pSpreadee+flag_gap)==FALSE)) {
           old_bit = (byte)(*pSpreadee & flag_mask);
           *pSpreadee = (byte)(*pSpreadee + error);
           if ( (*pSpreadee & flag_mask) != old_bit ) *(pSpreadee+flag_gap) = TRUE;
        }
        if (*(pSpreadee+flag_gap)==FALSE) *pRand++ = GREEN;
        else *pRand++ = RED;
     }
     *pRaster = (byte)(*pRaster & pixel_mask);       /* quanitse the last pixel without spreading */
  }

  /*****************************************************************************
   *                        NAME: Quantise_FloydS
   * PURPOSE: A variable depth output implementation of the original error
   *    diffusion algorithm (Floyd-Steinberg). It spreads the quantisation
   *    errors over the four remaining immediate unquantised neighbours,
   *    sharing the error according to the 'diffusion filter' shown below
   *    filter:     . Awhere A=7/16, B=1/16, C=5/16, D=3/16
   *              D C B and '.' represents the pixel being quantised.
   *    This version is variable in output depth from 1 to 6 bits/pixel.
   * History: original written 28th Jan 96, Andrew Murray. (working)
   *****************************************************************************/
  int Quantise_FloydS( byte *pSrc, byte *pDest, byte bits )
  {
     unsigned short fifo_index, fifo_len = frame_cols+1;
     short fifo[FRAME_COLS+1];
     long SpreadPixel, QuantError;
     byte QuantisedPixel;
     dword pixel, end_of_frame = frame_rows*frame_cols;
     dword a_limit = end_of_frame - 1, b_limit = a_limit - frame_cols;
     byte num_thresholds= (byte)pow(2,bits), threshold;
     short thresholds[64];
     byte colours[64];

     /* test validity of 'bits' */
     if (bits>6) return FALSE;

     /* create the arrya of thresholds and corresponding array of colours */
     for(threshold=0; threshold<num_thresholds; threshold++) {
        thresholds[threshold] = (short)(255*(2*threshold-1))/(2*(num_thresholds-1));
        colours[threshold] = (threshold*255)/(num_thresholds-1);
     }
     /* intialise the spreading array */
     for(fifo_index=0; fifo_index<fifo_len; fifo_index++) fifo[fifo_index]=0;
     fifo_index=0; /* - probably not necessary as the fifo buffer is circular */

     for(pixel=0; pixel<end_of_frame; pixel++){
        /* calculate the spread pixel, quantised version and error */
        SpreadPixel = ( (((long)*pSrc++)<<4) + fifo[(fifo_index++)%fifo_len] )>>4;
        threshold=num_thresholds-1;
        while (SpreadPixel<thresholds[threshold]) threshold--;
        QuantisedPixel=colours[threshold];
        QuantError = SpreadPixel - (long)QuantisedPixel;

        /* store the results (errors*16)*/
        *pDest++ = QuantisedPixel;
        if (pixel<a_limit)
           fifo[fifo_index%fifo_len] += QuantError*7; /* filter element A */
        if (pixel<b_limit){
           fifo[(fifo_index+frame_cols)%fifo_len] = QuantError; /* B */
           fifo[(fifo_index+frame_cols-1)%fifo_len] += QuantError*5; /* C */
           fifo[(fifo_index+frame_cols-2)%fifo_len] += QuantError*3; /* D */
        }
     }
     return TRUE;
  }

  /*****************************************************************************
   *                        NAME: Quantise_PrescaleFrame
   *     PURPOSE: used to solve dynamic range problems of the simple truncation
   *              quantiser used in Quantise_Diffuse, RandDiffuse etc. The
   *              value of bits passed should be the same as that passed to the
   *              coding function.
   *    Notes: 1) must be used in conjunction with Quantise_RescaleFrame,
 *                   which is applied after the quantisation function, prior to
 *                   image display.
 *****************************************************************************/
void Quantise_PrescaleFrame( byte *pSource, byte *pDest, byte bits )
{
   dword pixel, end_of_frame=(dword)(frame_cols*frame_rows);
   byte subtractant;
   byte sub_mask = (byte)(0xff << (8-bits)); /* ie. 11110000 for 4 bits, 11100000 for 3 etc. */
   byte rep, reps = (byte)((8/bits)-1); /* the number of complete subtractions */

   if ((8%bits)!=0) reps++; /* if 8 doesn't divide exactly by 'bits' a partial sub is req'd */

   Array_CopyFrame( pSource, pDest );

   if (bits!=1){
      for (pixel=0; pixel<end_of_frame; pixel++){
         subtractant = (byte)(*pDest & sub_mask);
         for (rep=0; rep<reps; rep++){
            subtractant = (byte)(subtractant >> bits);
            *pDest = (byte)(*pDest - subtractant);
         }
         pDest++;
      }
   }
   else{
      for (pixel=0; pixel<end_of_frame; pixel++){
         *pDest = (byte)(*pDest>>1);
         pDest++;
      }
   }
}

/*****************************************************************************
 *                          NAME:  Quantise_RescaleFrame
 *       PURPOSE: see prescale frame above
 *****************************************************************************/
void Quantise_RescaleFrame( byte *pImage, byte bits )
{
   dword pixel, end_of_frame=(dword)(frame_cols*frame_rows);
   byte rescaler;
   byte rep, reps = (byte)((8/bits)-1); /* the number of complete subtractions */

   if ((8%bits)!=0) reps++; /* if 8 doesn't divide exactly by 'bits' a partial sub is req'd */

   for (pixel=0; pixel<end_of_frame; pixel++){
      rescaler = *pImage;
      for (rep=0; rep<reps; rep++){
         rescaler = (byte)(rescaler >> bits);
         *pImage = (byte)(*pImage + rescaler);
      }
      pImage++;
}

   }
/*****************************************************************************
 *                          NAME:  Quantise_RescaleTruncate

 *****************************************************************************/
void Quantise_RescaleTruncate( byte *pSrc, byte *pDest, byte bits )
{
   dword pixel, end_of_frame=(dword)(frame_rows*frame_cols);
   byte shift = (byte)128/pow(2,bits);

   for(pixel=0; pixel<end_of_frame; pixel++)
      *pDest++ = *pSrc++ + shift;

   return;
}

/* ############## *
 * quantise.c end *
 * ############## */
```

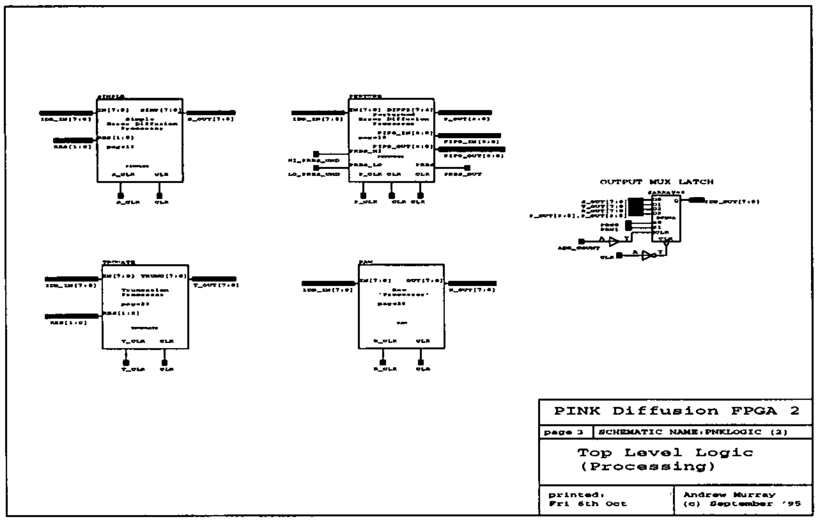
PINK FPGA schematics

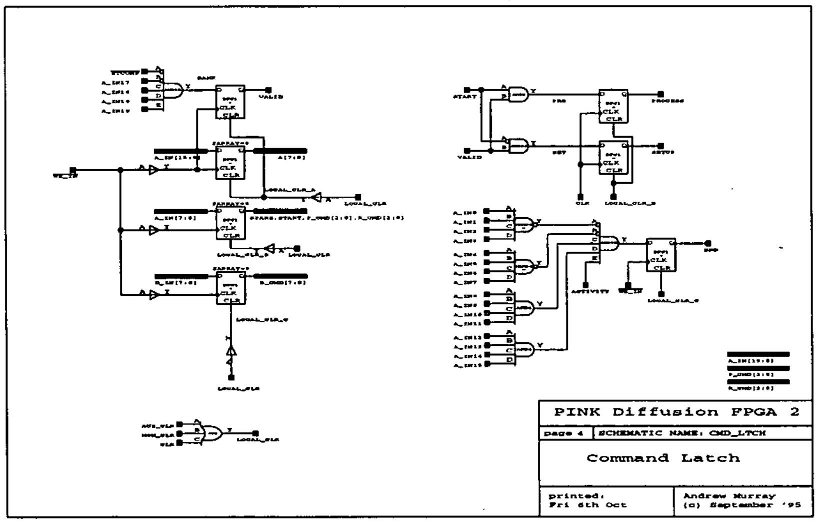
The full design of the PINK2 FPGA (covered in chapter four) is shown in the design schematics below.

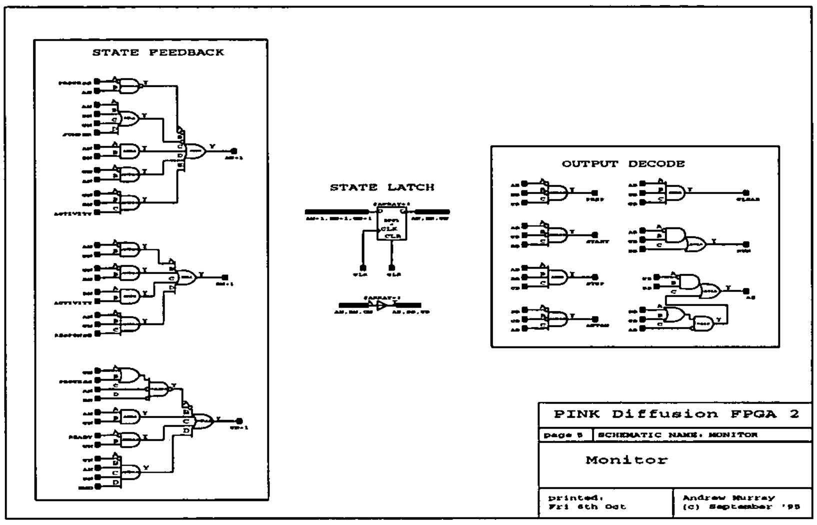


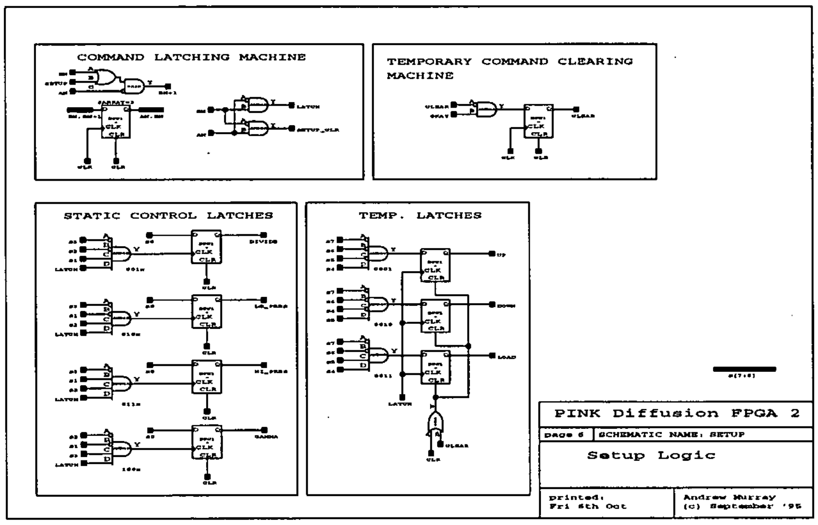


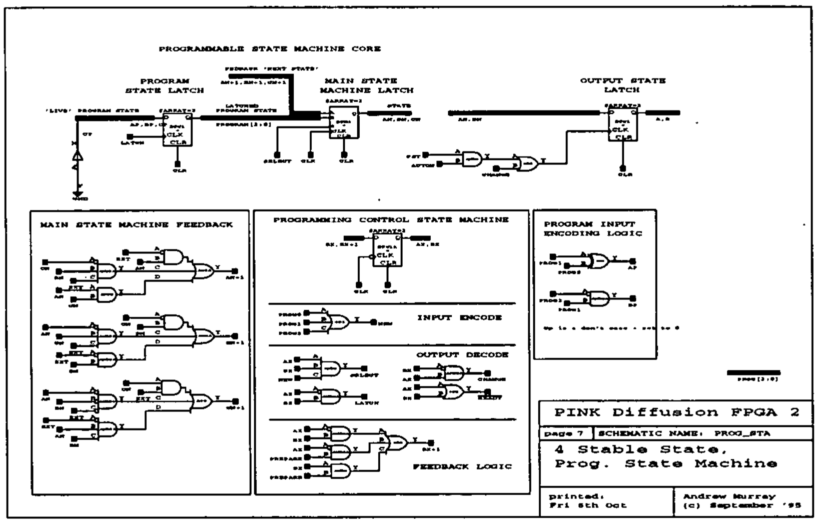

PINK Diffusion FPGA 2
page 2
SCHEMATIC NAME: PNKLOGIC (1)
Top Level Logic (Control)
printed: Fri 6th Oct
Andrew Murray (c) September '95

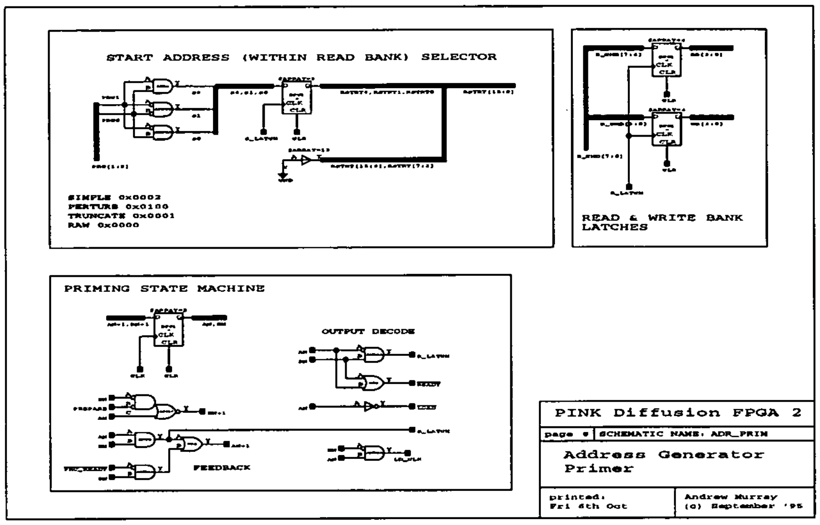

OUTPUT MUX LATCH
PINK Diffusion FPGA 2
page 3
SCHEMATIC NAME: PNKLOGIC (2)
Top Level Logic (Processing)
printed: Fri 6th Oct
Andrew Murray (c) September '95

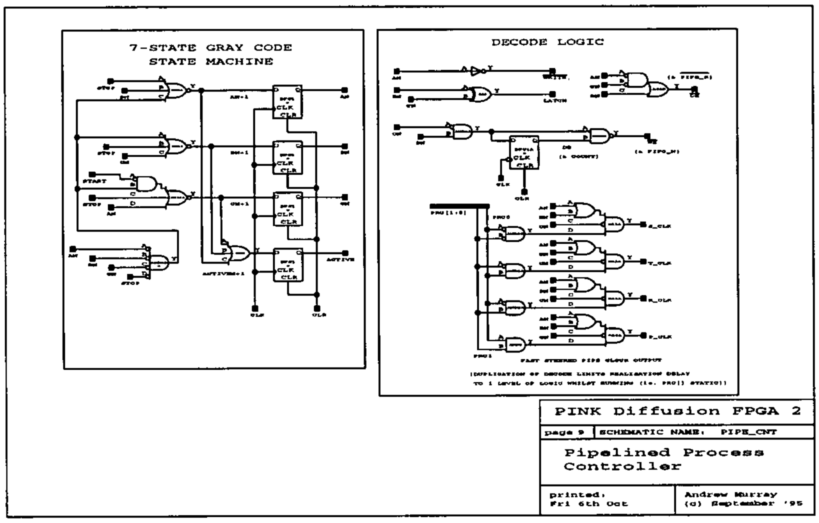

PINK Diffusion FPGA 2
page 4
SCHEMATIC NAME: CMD_LTCH
Command Latch
printed: Fri 6th Oct
Andrew Murray (c) September '95

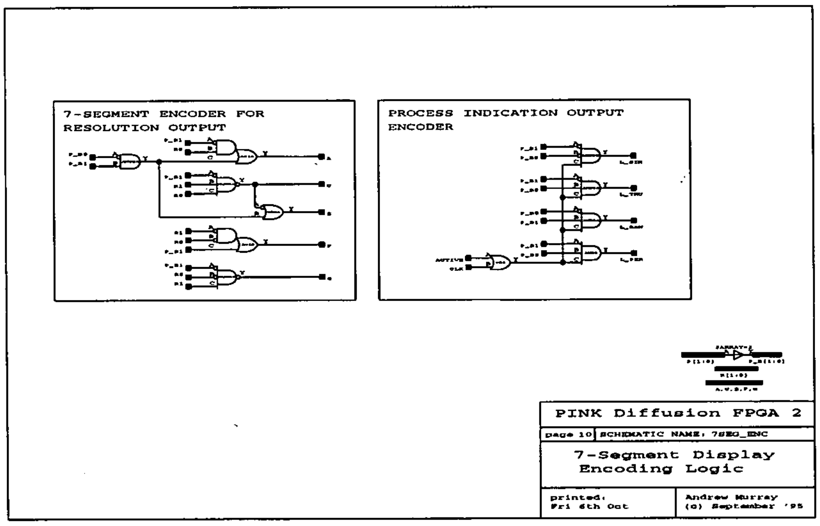

STATE FEEDBACK
STATE LATCH
OUTPUT DECODE
PINK Diffusion FPGA 2
page 5
SCHEMATIC NAME: MONITOR
Monitor
printed: Fri 6th Oct
Andrew Murray (c) September '95

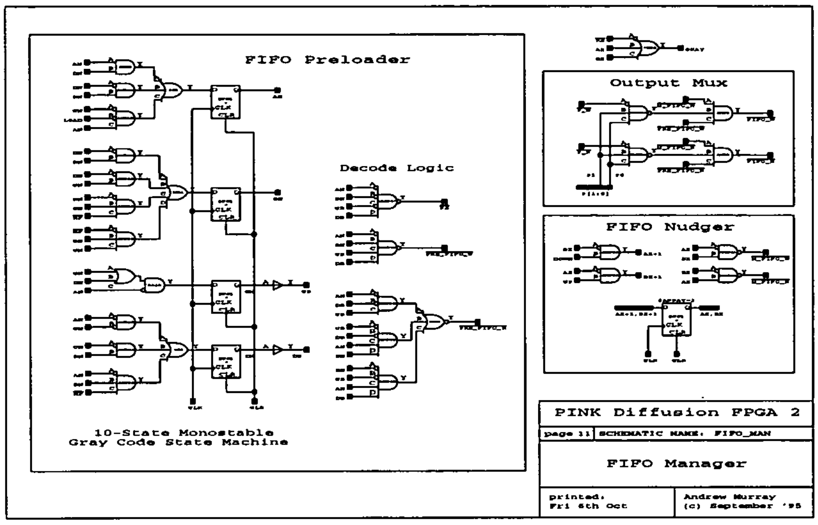

COMMAND LATCHING MACHINE
TEMPORARY COMMAND CLEARING MACHINE
STATIC CONTROL LATCHES
TEMP. LATCHES
PINK Diffusion FPGA 2
page 6
SCHEMATIC NAME: SETUP
Setup Logic
printed: Fri 6th Oct
Andrew Murray (c) September '95

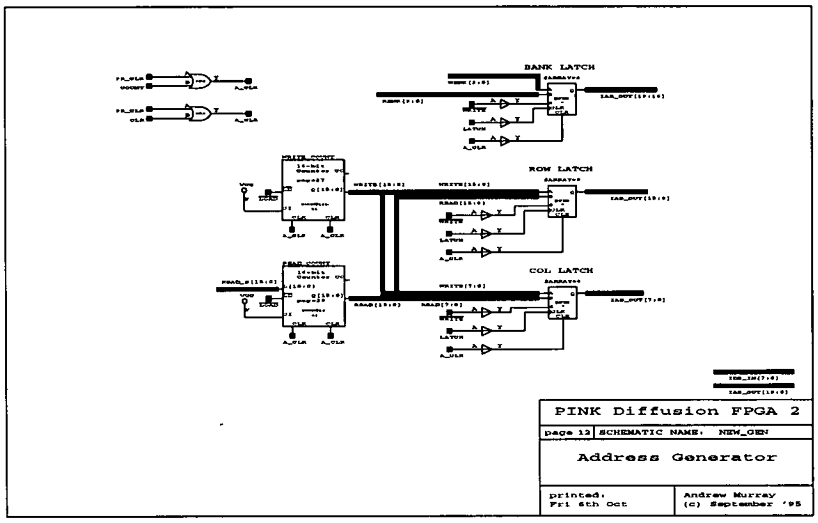

PROGRAMMABLE STATE MACHINE CORE
PROGRAM STATE LATCH
MAIN STATE MACHINE LATCH
OUTPUT STATE LATCH
MAIN STATE MACHINE FEEDBACK
PROGRAMMING CONTROL STATE MACHINE
PROGRAM INPUT ENCODING LOGIC
INPUT ENCODE
OUTPUT DECODE
FEEDBACK LOGIC
PINK Diffusion FPGA 2
page 7
SCHEMATIC NAME: PROG_STA
4 Stable State, Prog. State Machine
printed: Fri 6th Oct
Andrew Murray (c) September '95

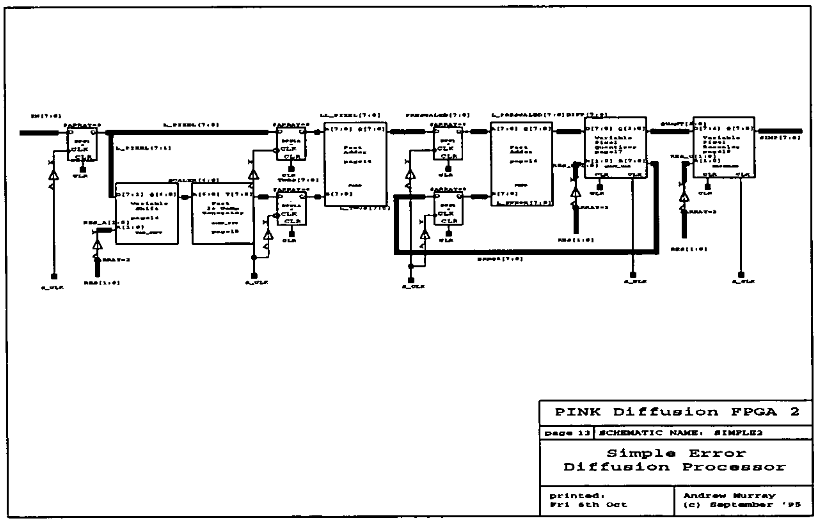

START ADDRESS (WITHIN READ BANK) SELECTOR
SIMPLE 0x0002
PERTURB 0x0100
TRUNCATE 0x0001
RAW 0x0000
READ & WRITE BANK LATCHES
PRIMING STATE MACHINE
OUTPUT DECODE
FEEDBACK
PINK Diffusion FPGA 2
page 8 SCHEMATIC NAME: ADR_PRIM
Address Generator Primer
printed: Fri 6th Oct
Andrew Murray (c) September '95

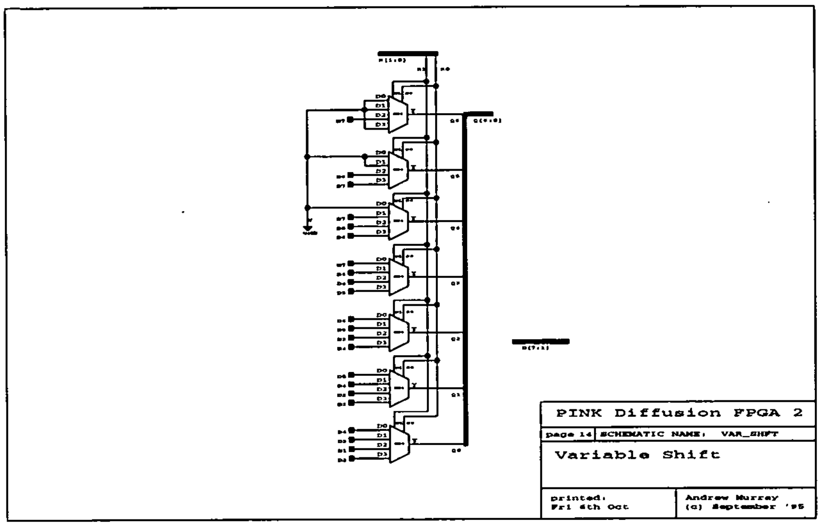

7-STATE GRAY CODE STATE MACHINE
DECODE LOGIC
PINK Diffusion FPGA 2
page 9 SCHEMATIC NAME: PIPE_CNT
Pipelined Process Controller
printed: Fri 6th Oct
Andrew Murray (c) September '95

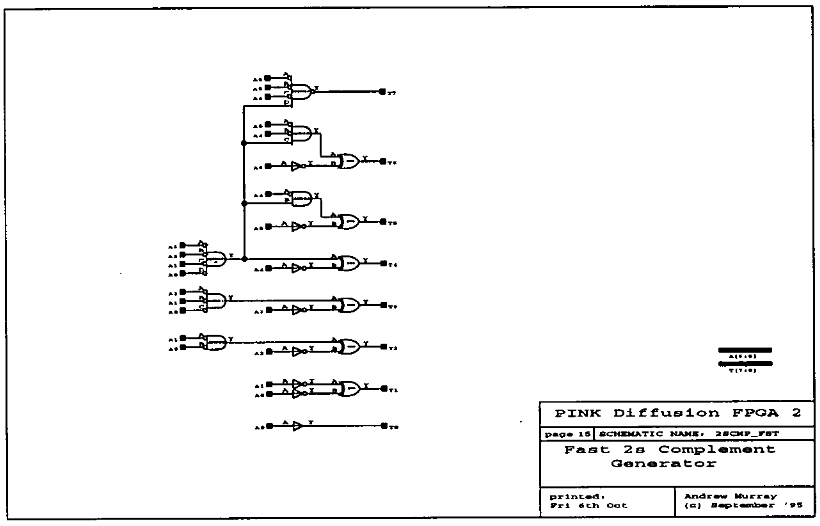

7-SEGMENT ENCODER FOR RESOLUTION OUTPUT
PROCESS INDICATION OUTPUT ENCODER
PINK Diffusion FPGA 2
page 10 SCHEMATIC NAME: 7SEG_ENC
7-Segment Display Encoding Logic
printed: Fri 6th Oct
Andrew Murray (c) September '95

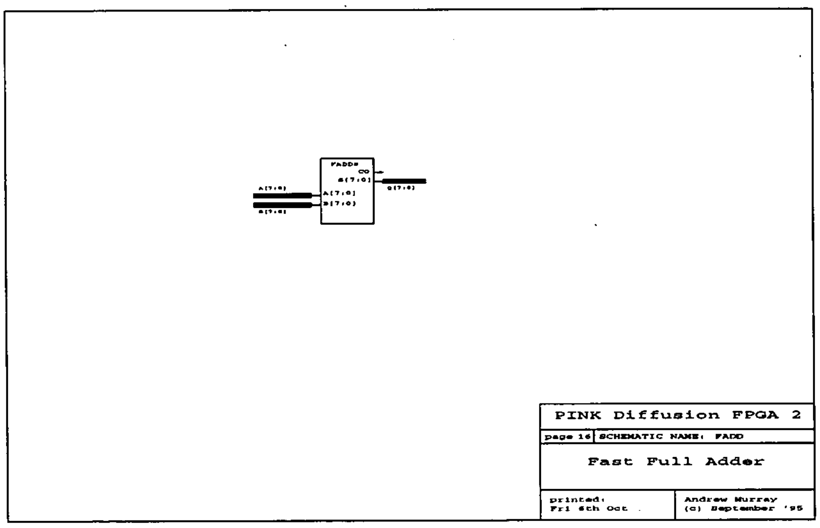

FIFO Preloader
10-State Monostable
Gray Code State Machine
Decode Logic
Output Mux
FIFO Nudger
PINK Diffusion FPGA 2
page 11 SCHEMATIC NAME: FIFO_MAN
FIFO Manager
printed: Fri 6th Oct
Andrew Murray (c) September '95

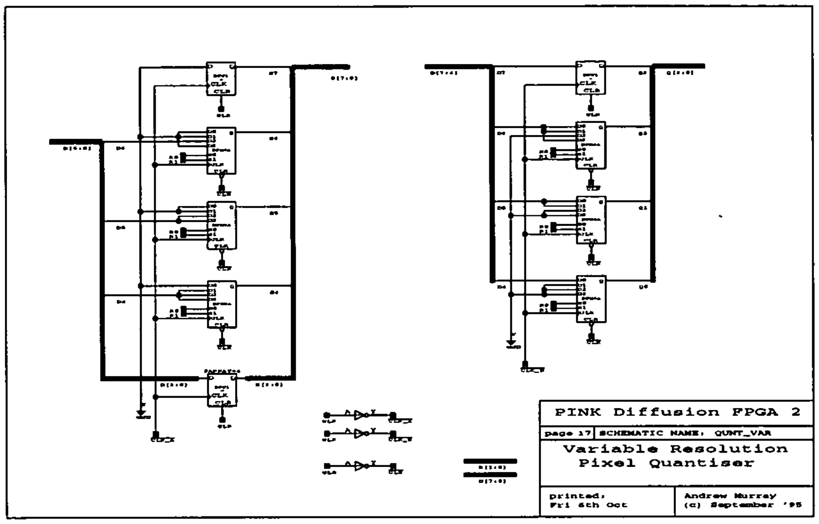

BANK LATCH
ROW LATCH
COL LATCH
PINK Diffusion FPGA 2
page 12 SCHEMATIC NAME: NEW_GEN
Address Generator
printed: Fri 6th Oct
Andrew Murray (c) September '95

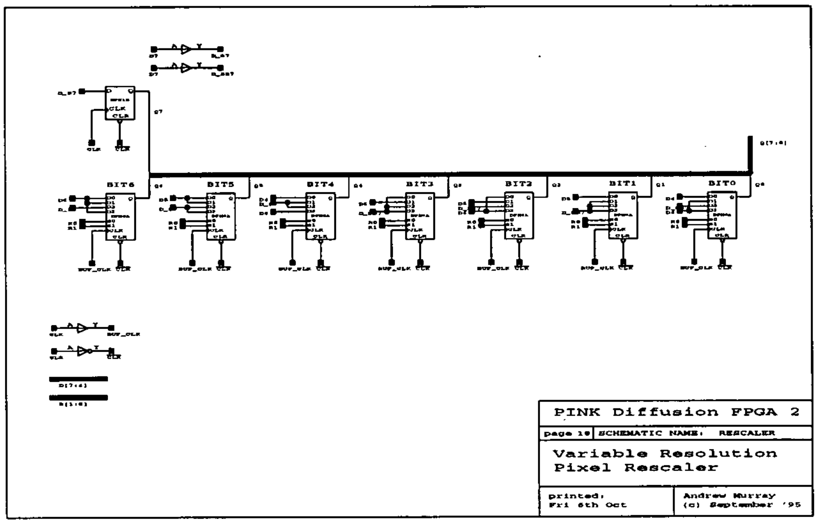

PINK Diffusion FPGA 2
page 13 SCHEMATIC NAME: SIMPLE2
Simple Error Diffusion Processor
printed: Fri 6th Oct
Andrew Murray (c) September '95

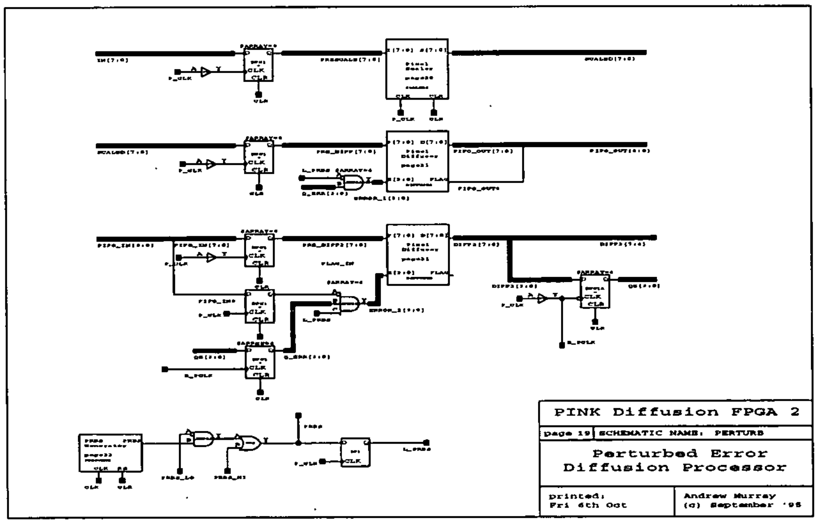

PINK Diffusion FPGA 2
page 14 SCHEMATIC NAME: VAR_SHFT
Variable Shift
printed: Fri 6th Oct
Andrew Murray (c) September '95

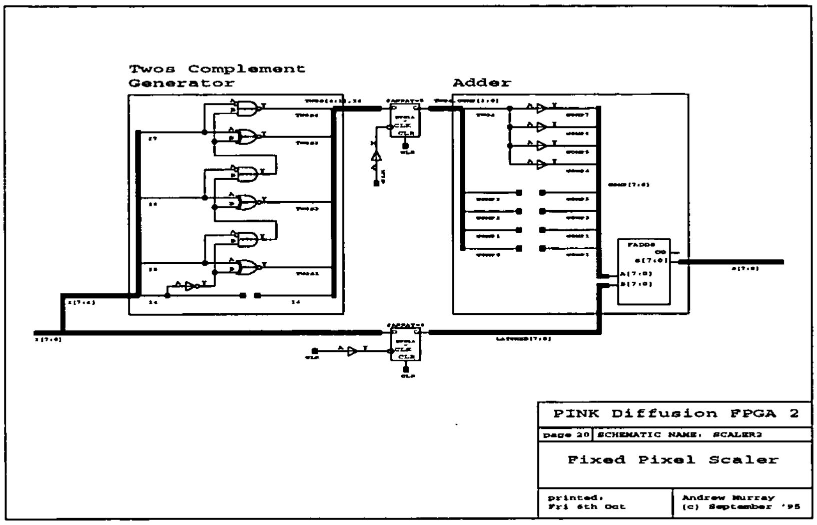

PINK Diffusion FPGA 2
page 15 SCHEMATIC NAME: 2SCMP_FST
Fast 2s Complement Generator
printed: Fri 6th Oct
Andrew Murray (c) September '95

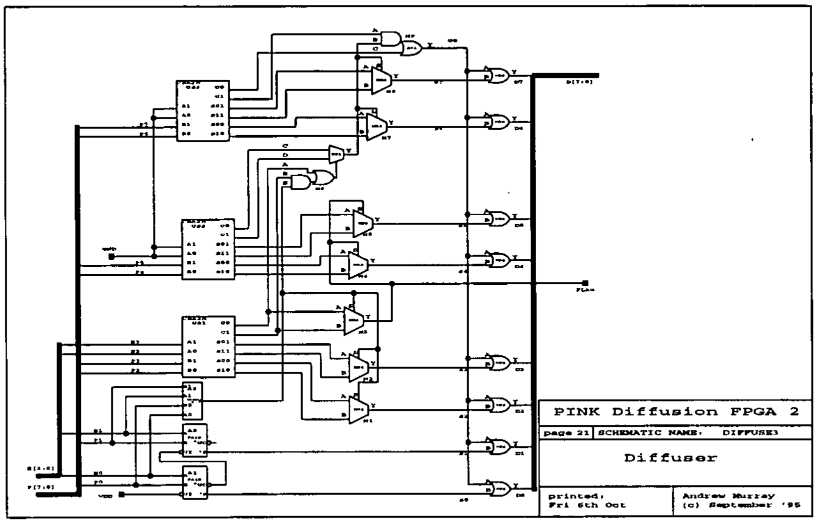

FADD8
CO
S(7:0)
A(7:0)
B(7:0)
PINK Diffusion FPGA 2
page 16 SCHEMATIC NAME: FADD
Fast Full Adder
printed: Fri 6th Oct
Andrew Murray (c) September '95

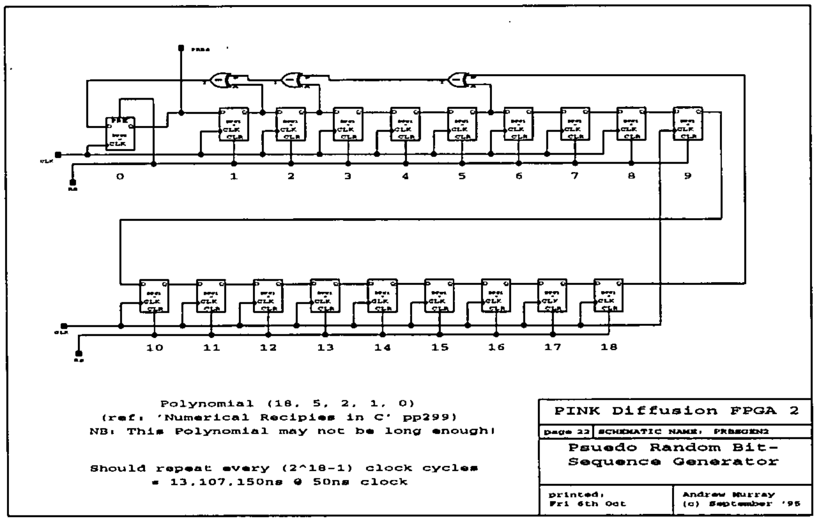

PINK Diffusion FPGA 2
page 17 SCHEMATIC NAME: QUNT_VAR
Variable Resolution Pixel Quantiser
printed: Fri 6th Oct
Andrew Murray (c) September '95

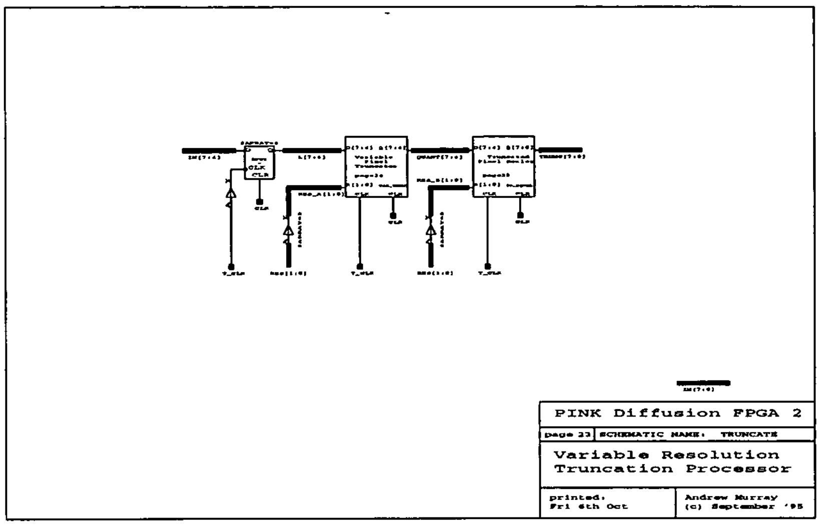

BIT6
BIT5
BIT4
BIT3
BIT2
BIT1
BIT0
PINK Diffusion FPGA 2
page 18 SCHEMATIC NAME: RESCALER
Variable Resolution Pixel Rescaler
printed: Fri 6th Oct
Andrew Murray (c) September '95

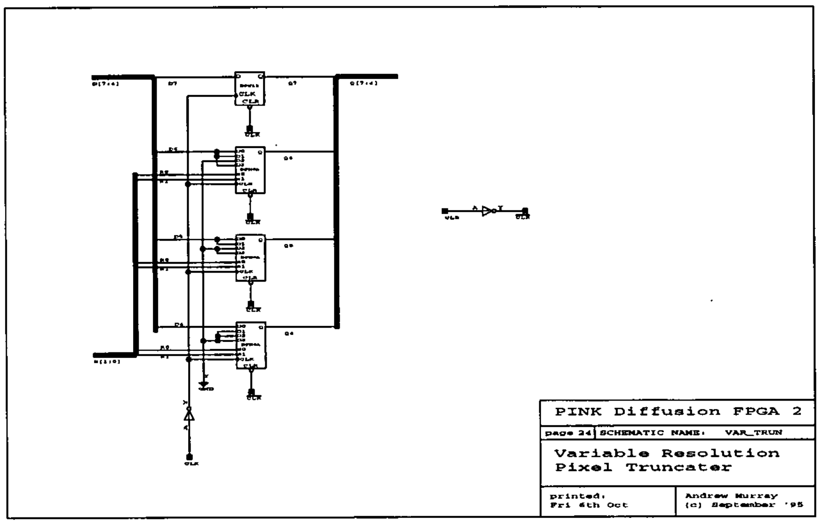

PINK Diffusion FPGA 2
page 19 SCHEMATIC NAME: PERTURB
Perturbed Error Diffusion Processor
printed: Fri 6th Oct
Andrew Murray (c) September '95

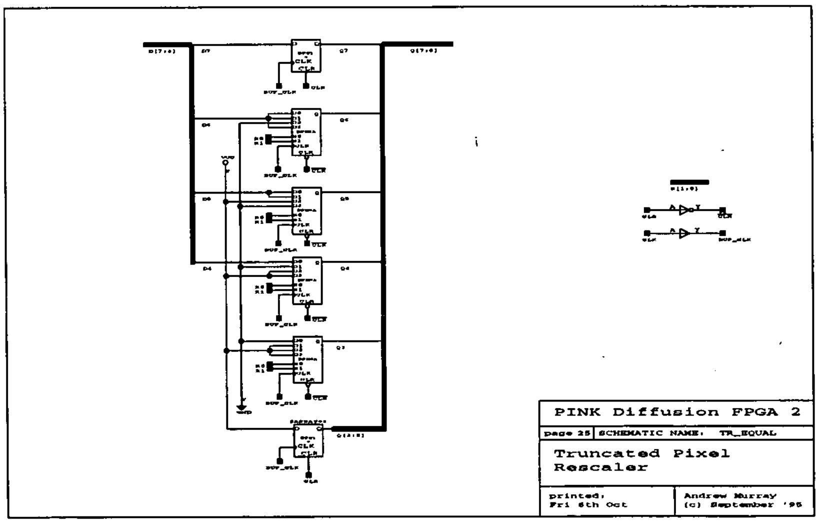

Twos Complement Generator
Adder
FADD8
PINK Diffusion FPGA 2
page 20 SCHEMATIC NAME: SCALER2
Fixed Pixel Scaler
printed: Fri 6th Oct
Andrew Murray (c) September '95

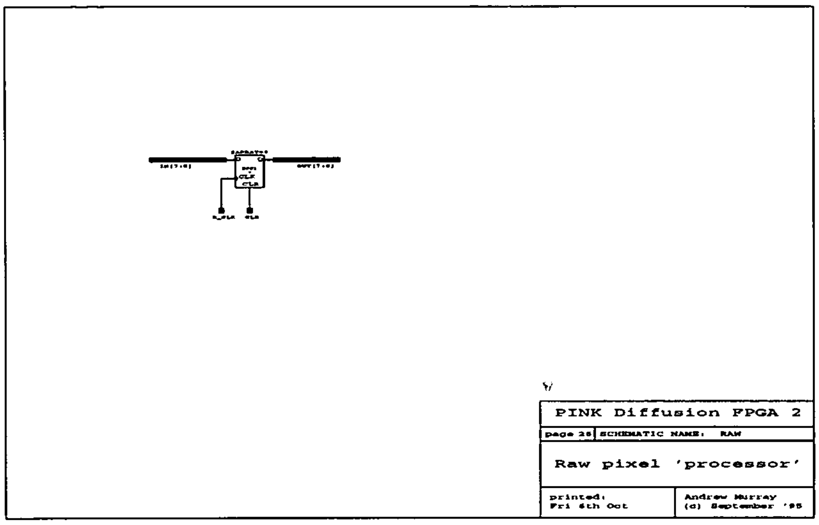

FLAG
PINK Diffusion FPGA 2
page 21 SCHEMATIC NAME: DIFFUSE3
Diffuser
printed: Fri 6th Oct
Andrew Murray (c) September '95

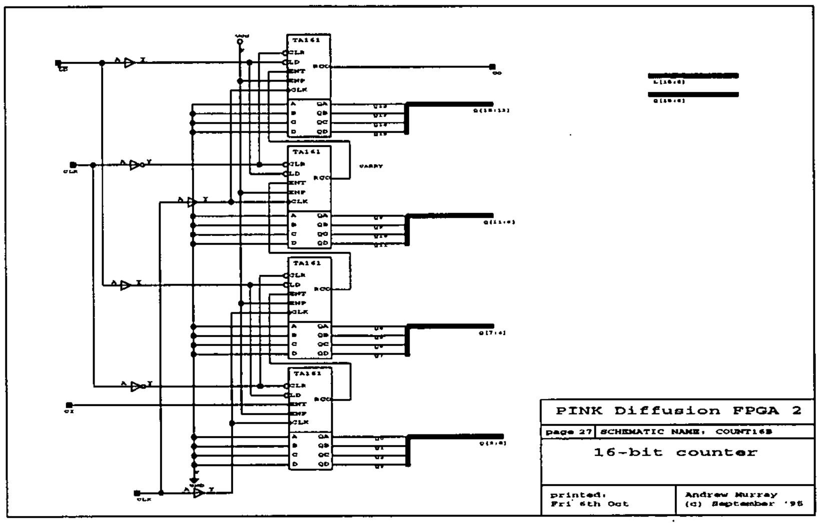

PRBS
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
Polynomial (18, 5, 2, 1, 0)
(ref: 'Numerical Recipies in C' pp299)
NB: This Polynomial may not be long enough!
Should repeat every (2^18-1) clock cycles
= 13,107,150ns @ 50ns clock
PINK Diffusion FPGA 2
page 22 SCHEMATIC NAME: PRBSGEN2
Psuedo Random Bit-Sequence Generator
printed: Fri 6th Oct
Andrew Murray (c) September '95

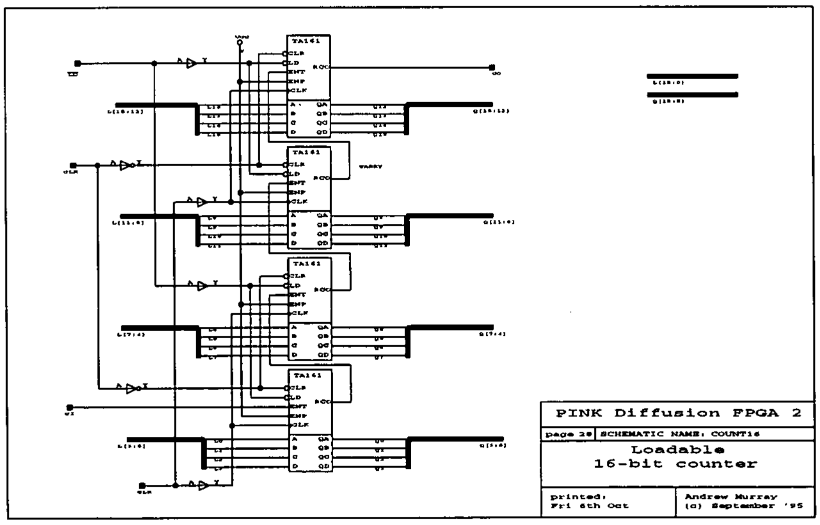

PINK Diffusion FPGA 2
page 23 SCHEMATIC NAME: TRUNCATE
Variable Resolution Truncation Processor
printed: Fri 6th Oct
Andrew Murray (c) September '95


PINK Diffusion FPGA 2
page 24 SCHEMATIC NAME: VAR_TRUN
Variable Resolution Pixel Truncater
printed: Fri 6th Oct
Andrew Murray (c) September '95


PINK Diffusion FPGA 2
page 25 SCHEMATIC NAME: TR_EQUAL
Truncated Pixel Rescaler
printed: Fri 6th Oct
Andrew Murray (c) September '95


PINK Diffusion FPGA 2
page 26 SCHEMATIC NAME: RAW
Raw pixel 'processor'
printed: Fri 6th Oct
Andrew Murray (c) September '95


TA161
CARRY
PINK Diffusion FPGA 2
page 27 SCHEMATIC NAME: COUNT16B
16-bit counter
printed: Fri 6th Oct
Andrew Murray (c) September '95


TA161
CARRY
PINK Diffusion FPGA 2
page 28 SCHEMATIC NAME: COUNT16
Loadable 16-bit counter
printed: Fri 6th Oct
Andrew Murray (c) September '95

appendix three

Imputer FPGA firmware

The source files for the imputer firmware used in testing the FPGA are listed below. The file than contains the menu-driven test program (pinktest.c) is first listed, this is followed by the files that make up the library of commands that can be used to initiate processing of images using the FPGA.

Pinktest.c

This functions sets up the general program environment. Command line flags can be used to start it in simulate, record, demo or test modes.

```
/* ########################################################################### *
 * pinktest.c -  a menu driven test program for the pink.c library and the
 *  PINK Diffusion FPGA
* ########################################################################### */

#include <stdio.h>
#include <stdimp.h>
#include <timer.h>
#include <imputer.h>
#include <math.h>
#include "pink.h"

typedef struct bank_register
{
byte vid_gen;
byte grab;
byte logo;
byte test;
}bank_register;

/* Private function forward declarations */
void RunProcessor( byte process, bank_register reg, int twin );
void RunTestImage( byte process, bank_register reg );
void DisplayMenu();
void DisplayStatus( byte process, int logo_flag, int smooth_flag, bank_register reg
);
void ToggleFlag( int *pFlag );
void GrabFrame( byte dest );
void GenerateTestImage( bank_register reg );
void GammaCorrectDisplay( float *pGamma);
void GammaCorrectImage( byte bank, float gamma );
void RampOnRHS();
void ApplyFixedCorrection( byte bank );

/* ############# *
 * main function *
 * ############# */

void main()
{
   bank_register banks = { 0x1, 0x1, 0xe, 0x0 };
   byte current_process = SIMPLE;
   int quit = FALSE, logo = FALSE, smooth = TRUE;
   char letter;
   int cycles = 100;
   float gamma=2.2;

   imputer_init();

   printf("\nPINK FPGA Test/Demo Software\nVersion 2.01 (pinktst3.c)\nCopyright (c)
nonsense '95\n\n");
   /* initial processing to allow autonomous operation */
   RunProcessor( current_process, banks );

   /* main menu driven processing loop */
   RunProcessor( current_process, banks, smooth );

   while (!quit){
      if (logo) set_video_bank( 0 );
      DisplayStatus( current_process, logo, smooth, banks );
      DisplayMenu();
      letter = getchar();
      printf("\n");
      switch (letter) {
         case 's': current_process = SIMPLE; break;
         case 'p': current_process = PERTURB; break;
         case 'r': RunProcessor( current_process, banks, smooth );   break;
         case 'h': RampOnRHS();
         case 't': RunTestImage( current_process, banks );   break;
         case 'f': GenerateTestImage( banks );      break;
         case 'l': ToggleFlag( &logo );     break;
         case 'm': ToggleFlag( &smooth );   break;
         case 'g': GammaCorrectDisplay( &gamma ); break;
         case 'c': GammaCorrectImage( 14, gamma ); break;
         case 'a': ApplyFixedCorrection( 14 ); break;
         case 'q': quit=TRUE;      break;
         default: printf("\tUnrecognised input '%c'\n", letter);
      }
   }

   printf("\n\tbye.\n");
   reset_imputer();
}

/* ############################ *
 * Private function declarations *
 * ############################ */

/************************************************************************/
void RunProcessor( byte process, bank_register reg, int toggling )
{
   if (!toggling){
      set_video_bank( reg.vid_gen );
      SetPinkBanks( reg.grab, (reg.vid_gen+14) );
      printf("\n\tEntering the processor loop, press the STOP button to exit...");
      while ( !halt() ){
         GrabFrame( reg.grab );
         RunPinkProcessor( process );
      }
   }
   else{
      printf("\n\tEntering the smooth processor loop, press the STOP button to
exit...");
      while ( !halt() ){
      SetPinkBanks( reg.grab, 14 );
      GrabFrame( reg.grab );
      RunPinkProcessor( process );
      set_video_bank( 0 );
      SetPinkBanks( reg.grab, 15 );
      GrabFrame( reg.grab );
      RunPinkProcessor( process );
      set_video_bank( 1 );
      }
   }
   printf(" stopped.\n");
}

/***************************************************************************/
void RunTestImage( byte process, bank_register reg )
{
   set_video_bank( 0 );
   SetPinkBanks( reg.test, 14 );
   RunPinkProcessor( process );
}

/***************************************************************************/
void RampOnRHS()
{
   byte row, col;

   set_main_xbank( 14 );
   for(row=0; row<255; row++)
      for(col=0; col<128; col++)
         XBYTE[(row*256+col+128)]=row;
}

/***************************************************************************/
void GenerateTestImage( bank_register reg )
{
   word pixel=0, row, col;
   char key;

   set_main_xbank( reg.test );

   do{
      printf("\n\tPlease choose test pattern:\n");
      printf("\t[1] vertical ramp [2] horizontal ramp [3] vertical blocks\n");
      printf("\twaiting... ");
      key = getchar();
      printf("\n");
   }while ( key!='1' && key!='2' && key!='3' );

   printf("\tGenerating test image (bank %d), please wait...", (int)reg.test);
   for( row=0; row<256; row++ )
      for( col=0; col<256; col++ )
         switch (key) {
            case '1': XBYTE[pixel++] = row; break;
            case '2': XBYTE[pixel++] = col; break;
            case '3': XBYTE[pixel++] = (byte)((row&0xf0) | (row>>4)); break;
         }
   printf(" done.\n");
}

/***************************************************************************/
void GammaCorrectDisplay( float *pGamma )
{
   char key;
   word row, col, row_pair, col_pair;
   word pixel;
   byte refresh=TRUE, stop=FALSE;
   byte corrected_grey;
   float inverse_gamma;

   set_main_xbank( 14 );
   set_video_bank( 0 );

   printf("\tFilling bank 14 with background pattern...");
   pixel=0;
   for (row_pair=0; row_pair<64; row_pair++){
      for (col_pair=0; col_pair<128; col_pair++){
         XBYTE[pixel++]=0;
         XBYTE[pixel++]=0;
         XBYTE[pixel++]=255;
         XBYTE[pixel++]=255;
      }
      for (col_pair=0; col_pair<128; col_pair++){
         XBYTE[pixel++]=255;
         XBYTE[pixel++]=255;
```

```
            XBYTE[pixel++]=0;
            XBYTE[pixel++]=0;
        }
    }
    printf("done.\n");

    printf("\tPress [>] or [<] to alter gamma, [R] to refresh display and [Q] to
exit\n");
    do{
        if (refresh) {
            inverse_gamma = 1/(*pGamma);
            corrected_grey = (byte)( (256*pow(0.5, inverse_gamma)-1) );
            for(row=63; row<191; row++)
                for(col=63; col<191; col++)
                    XBYTE[(row*256)+col] = corrected_grey;
            printf("\tcurrent image corrected with a gamma of %.1f (%d)\n", *pGamma,
(int)corrected_grey);
            refresh=FALSE;
        }
        key = getchar();
        printf("\n");
        switch (key) {
            case '.': *pGamma += 0.1F; break;
            case ',': if (*pGamma>0.1F) *pGamma -= 0.1F; break;
            case 'r': refresh=TRUE;   break;
            case 'q': stop=TRUE;      break;
            default: printf("\tUnrecognised input '%c'\n", key);
        }
        printf("\tnext gamma: %.1f\n", *pGamma );
    }while( !stop );

    return;
}

/****************************************************************************/
void GammaCorrectImage( byte bank, float gamma )
{
    dword index;
    float inverse_gamma = 1/gamma;
    byte old_grey;

    set_main_xbank( bank );
    set_video_bank( bank-14 );

    printf("\n\tGammaCorrectImage: starting (gamma = %.1f inv = %.1f)\n", gamma,
inverse_gamma);

    printf("\tGammaCorrectImage: creating look-up table...");
    for (index=0; index<256; index++)
        XBYTE[index] = (byte)( 255*pow( ( (float)index/255), inverse_gamma) );

    printf("done.\n\tGammaCorrectImage: correcting image...");
    for (index=256; index<65535; index++){
        old_grey = XBYTE[index];
        XBYTE[index] = XBYTE[old_grey];
    }
    printf("done.\n");
    return;
}

/****************************************************************************/
void ApplyFixedCorrection( byte bank )
{
    dword pixel;
    word row, col;
    byte old_grey;
    boolean stop=FALSE;

    set_main_xbank( bank );
    set_video_bank( bank-14 );

    printf("\tApplyFixedCorrection: correcting image...");
    for(row=0; row<256; row++)
        for(col=0; col<128; col++){
            pixel=(row*256)+col;
            old_grey = XBYTE[pixel];
            switch (old_grey){
                case 32: XBYTE[pixel]=0; break;
                case 0x00: break;
                case 0x11: XBYTE[pixel]=56; break;
                case 0x22: XBYTE[pixel]=65; break;
                case 0x33: XBYTE[pixel]=75; break;
                case 0x44: XBYTE[pixel]=85; break;
                case 0x55: XBYTE[pixel]=95; break;
                case 0x66: XBYTE[pixel]=107; break;
                case 0x77: XBYTE[pixel]=119; break;
                case 0x88: XBYTE[pixel]=132; break;
                case 0x99: XBYTE[pixel]=144; break;
                case 0xaa: XBYTE[pixel]=156; break;
                case 0xbb: XBYTE[pixel]=170; break;
                case 0xcc: XBYTE[pixel]=188; break;
                case 0xdd: XBYTE[pixel]=208; break;
                case 0xee: XBYTE[pixel]=232; break;
                case 0xff: break;
                default: printf(" error\n\t read '%d' not 4bit/pixel data\n",
(int)old_grey);
            }
        }
    printf("done.\n");
    return;
}

/**************************************************************************/
void DisplayMenu()
{
    printf("\n\tPress:");
    printf(  "\t[R] Run processor    [T] run Test image\n");
    printf("\t\t[M] toggle switching \n");
    printf("\t\t[A] Apply fixed cor.[S] change to Simple [P] change to Perturb\n");
    printf("\t\t[F] Fill test bank   [L] toggle Logo     [H] create ramp on RHS\n");
    printf("\t\t[G] Test Gammas      [C] gamma Correct   [Q] to
Quit\n\twaiting...");
}

/**************************************************************************/
void DisplayStatus( byte process, int logo_flag, int smooth_flag, bank_register reg
)
{
    printf("\n\n\n\tStatus:\tProcess: ");
    switch (process){
        case PERTURB: printf("PERTURB"); break;
        case SIMPLE: printf("SIMPLE"); break;
        default: printf("\n\t\tError - unrecognised process\n");
    }
    printf("\t\tBank Switching ");
    switch (smooth_flag){
        case TRUE: printf("On"); break;
        case FALSE: printf("Off"); break;
        default: printf("\n\t\tError - invalid smooth flag\n");
    }
    printf("\tLogo ");
    switch (logo_flag){
        case TRUE: printf("Enabled\n"); break;
        case FALSE: printf("Disabled\n"); break;
        default: printf("\n\t\tError - invalid logo flag\n");
    }
    printf("\t\tVid_gen:%x, Grab:%x, ", (int)reg.vid_gen, (int)reg.grab);
    printf("Logo:%x, Test:%x\n", (int)reg.logo, (int)reg.test );
}

/**************************************************************************/
void ToggleFlag( int *pFlag )
{
    if (*pFlag) *pFlag=FALSE;
    else *pFlag=TRUE;
}

/**************************************************************************/
void GrabFrame( byte dest )
{
    set_main_xbank( dest );
    capture_image(GB_STANDARD);
}

/* ############## *
 * pinktest.c end *
 * ############## */
```

perturb.c

```
#include <stdio.h>
#include <stdimp.h>
#include "pinkdefs.h"

/***************************************************************************
 *   NAME: RunPinkPerturbProcessor
 * PURPOSE: Used to start the 'Perturbed' Error Diffusion processor.
 * NB: Before using either of the internal processors the Address generator
 *   should be initialised with read and write bank addresses.
 ***************************************************************************/
void RunPinkPerturbProcessor()
{
   set_main_xbank( PINK_BANK );     /* switch to the xbank for PINK FPGA commands */
   STCONF=0;     /* toggle the mem. map to 'processors' space*/
   XBYTE[ PERTURB ]=0; /* command FPGA to start the PERTURB processor */
   STCONF=1;     /* toggle the mem. map back to 'RAM' add. space*/
   while (!RUN);    /* wait for the FPGA to release the 'imp. bus' */
}
```

process.c

```
#include <stdio.h>
#include <stdimp.h>
#include "pinkdefs.h"

/***************************************************************************
 *   NAME: RunPinkProcessor
 *PURPOSE: Used to start either Error Diffusion processor.
 * NB: Before using either of the internal processors the Address generator
 * should be initialised with read and write bank addresses.
 ***************************************************************************/
void RunPinkProcessor( byte process )
{
   set_main_xbank( PINK_BANK );     /* switch to the xbank for PINK FPGA commands */
   STCONF=0;     /* toggle the mem. map to 'processors' space*/
   XBYTE[ process ]=0; /* command FPGA to start the SIMPLE processor */
   STCONF=1;     /* toggle the mem. map back to 'RAM' add. space*/
   while (!RUN);    /* wait for the FPGA to release the 'imp. bus' */
}
```

setbanks.c

```
#include <stdio.h>
#include <stdimp.h>
#include "pinkdefs.h"

/***************************************************************************
 *   NAME: SetPinkBanks
 *PURPOSE: Used to control which external memory banks (xbanks) the FPGA
 * will read from and write to. This command should be used prior
 * to using either of the processors. The FPGAs internal address generator
 * defaults to xbank 0 for both read and write on power-up or hardware or
 * software reset.
 ***************************************************************************/
void SetPinkBanks( byte source, byte dest )
{
   byte combined_banks;

   set_main_xbank( PINK_BANK );
   STCONF=0;
   combined_banks = (dest<<4)|source;
   XBYTE[ BANK_LATCH ]=combined_banks;
   STCONF=1;
}
```

simple.c

```
#include <stdio.h>
#include <stdimp.h>
#include "pinkdefs.h"

/***************************************************************************
 *   NAME: RunPinkSimpleProcessor
 *PURPOSE: Used to start the 'Simple' Error Diffusion processor.
 * NB: Before using either of the internal processors the Address generator
 *   should be initialised with read and write bank addresses.
 ***************************************************************************/
void RunPinkSimpleProcessor()
{
   set_main_xbank( PINK_BANK );     /* switch to the xbank for PINK FPGA commands */
   STCONF=0;     /* toggle the mem. map to 'processors' space*/
   XBYTE[ SIMPLE ]=0;  /* command FPGA to start the SIMPLE processor */
   STCONF=1;     /* toggle the mem. map back to 'RAM' add. space*/
   while (!RUN);    /* wait for the FPGA to release the 'imp. bus' */
}
```

appendix four

Subjective test software

The source files for the software used to present the subjective tests (chapter five) are listed below.

demo01.h

```
#ifndef _demo_
#define _demo_

// macro definitions
#define ORIGINAL 0
#define PROC1 1
#define PROC2 2
#define PROC3 3
#define PROC4 4
#define PROC5 5
#define ORIGINAL 0
#define NONE 6

#define BLACKBOARD_NAME     "Subjective Test Demo v0.1 "

typedef struct pos_and_size
{
  int x;
  int y;
  int     width;
  int height;
}POS_AND_SIZE;

#define DEMO_INI_FILE "demo.ini"

// state functions
void DoIdle();
void DoInitialise();
void DoIterate();
void DoTerminate();

void SequenceControlPanel();         // to be removed
void SequenceControlPanelClose();    // to be removed
void DummyButton( void );
void TurnPageForward();     // to be removed
void TurnPageBack();        // to be removed
void Demo_TurnToPage( int page );
int Demo_GlobalsInit( char *filename, char *section );
int Demo_SetupBlackboard();
void Demo_ActivatePlayer( int new_machine );
int ReadButtonDetails( char *filename, char *section, POS_AND_SIZE *button );
int DetermineDemoType();
#endif
```

demo01.cpp

```
/*****************************************************************************
 *        demo01.cpp
 *
 *        Description:written to display images for subjective tests of compression
 *        algorithms.
 *
 *        History:         -09-96   Created
          aamu
 *
 *****************************************************************************/

/* ######## *
 * includes *
 * ######## */
#include "image.h"

#include "visframe.h"
#include "apheader.h"
#include "controls.h"
#include "demo01.h"
#include "player.h"
#include "fileio.h"
#include "graphic.h"
#include <math.h>

/* ##################### *
 * global variable def's *
 * ##################### */
PLAYER players[6];
char translation[71];
int page_number;
int demo_id;
int max_pages;
int looping;
int verbose;
int show_details;
int active_player;
char temp_buffer[30];
int still_page;
Blackboard         *blackboard = NULL;
BLACKBOARD_ARRAY BbArray;


/* ################################## *
 * Vision Framework 'state' functions *
 * ################################## */

  /*****************************************************************************
 *                NAME: DoInitialise
 *PURPOSE:        called by visual framework the first time it executes its loop, *
                  this function sets up the demo application.
 *****************************************************************************/
void DoInitialise()
{
  int vcr;

  VFAppHide();
  VFAppDialogColours(BLACK, RGB(255,255,232));
  SetDrawColour( 0 );
  SetBackColour( 255 );

  if (!Demo_GlobalsInit( "[globals]", "demo.ini" ))
                  VFMessageBox(MB_OK|MB_ICONEXCLAMATION,"DoInitialise","Demo_Init
returned FALSE");
  Demo_SetupBlackboard();

  /* construct and intialise sequence players */
  for (vcr=0; vcr<6; vcr++){
          sprintf( temp_buffer, "[player%i]", vcr );
          if (!Player_Init( temp_buffer, "demo.ini", &players[vcr] ))
                  VFMessageBox(MB_OK|MB_ICONEXCLAMATION,"DoInitialise","Player_Init
returned FALSE");
          }

  page_number=0;
 active_player=NONE;
 Demo_TurnToPage( page_number );
}

/***************************************************************************
 *        NAME: DoIterate
 *PURPOSE:        called by visual framework each time it executes its loop,
 *                except for the first time or if quit has been pressed.
 *                this function contains the iterative processing cycle of the
 *                demo app.
 ***************************************************************************/
void DoIterate()
{
  if (active_player!=NONE){
        if
(players[active_player].frame_counter<players[active_player].max_frames)
                   Player_DisplayNextFrame( &players[active_player] );
                   else {
                           if (looping) Player_ResetSequence(
&players[active_player], FALSE );
                           else {
                                   Player_ResetSequence( &players[active_player],
TRUE );
                                   active_player = NONE;
                           }
          }
  }

}

/***************************************************************************
 *                NAME: DoTerminate
 *PURPOSE:        called by visual framework the last time it executes its loop,
 *                this function tidies up all the demo app. structures
 ***************************************************************************/
void DoTerminate()
{
  int vcr;

  for (vcr=0; vcr<6; vcr++)
          Player_Destroy( &players[vcr] );
  if (verbose) VFMessageBox(MB_OK|MB_ICONINFORMATION,"DoTerminate()","players
destroyed");
  VFBlackboardDestroy(blackboard, &BbArray);
  if (verbose) VFMessageBox(MB_OK|MB_ICONINFORMATION,"DoTerminate()","blackboard
destroyed");
}

/***************************************************************************
 *                NAME: DoIdle
 *PURPOSE:        the fourth possible VF function - unused in the demo app.
 ***************************************************************************/
void DoIdle()
{
}


/* ############################## *
 * functions used by buttons.cpp *
 * ############################## */


void SequenceControlPanel()
{
  /*VFBlackboardShow(sequencebb);*/
}
void SequenceControlPanelClose()
{
  /*VFBlackboardHide(sequencebb);*/
}
void DummyButton( void )
{
  VFMessageBox(MB_OK|MB_ICONINFORMATION,"DummyButton","dummy button pressed");
}
void TurnPageForward()
{
  if (page_number<max_pages) Demo_TurnToPage( ++page_number );
  else VFMessageBox( MB_OK|MB_ICONEXCLAMATION, BLACKBOARD_NAME, "you are on the last
page" );
}
void TurnPageBack()
{
  if (page_number>0) Demo_TurnToPage( --page_number );
  else VFMessageBox( MB_OK|MB_ICONEXCLAMATION, BLACKBOARD_NAME, "there is no
previous page" );
}

/*************************************************************************
 *        NAME: Globals_Init
 *PURPOSE:        Initialises the global variables from the initialisation file.
 *************************************************************************/
int Demo_GlobalsInit( char *section, char *filename )
{
  FILE *pFile;

  /* read global parameters from the .ini file */
  pFile = FileIO_FindIniSection( filename, section );
  if (pFile == NULL) return FALSE;
  if (!FileIO_ReadIniLineNumber( pFile, "looping", &looping )) goto close;
  if (!FileIO_ReadIniLineNumber( pFile, "verbose", &verbose )) goto close;
  if (!FileIO_ReadIniLineNumber( pFile, "max_pages", &max_pages )) goto close;
  if (!FileIO_ReadIniLineNumber( pFile, "demo_id", &demo_id )) goto close;
  if (!FileIO_ReadIniLineNumber( pFile, "show_details", &show_details )) goto close;
  if (!FileIO_ReadIniLineText( pFile, "t", translation )) goto close;
  fclose(pFile);
  if (verbose) VFMessageBox(MB_OK|MB_ICONINFORMATION,"Demo_GlobalsInit","ended OK");
  return TRUE;

close:
  VFMessageBox(MB_OK|MB_ICONEXCLAMATION,"Demo_GlobalsInit","a 'ReadIniLine...'
failed");
  fclose(pFile);
  return FALSE;
}

/*************************************************************************
 *                                                                       *
 *************************************************************************/
int Demo_SetupBlackboard()
{
  FILE *pFile;
  POS_AND_SIZE temp;

  /* == construct a controls list == */
  BbArray.NumElements = NUMBBELEMENTS;
  VFBlackboardControlArrayConstruct(&BbArray);

  /* == add item onto blackboard == */
  ReadButtonDetails( "[controls_box]", "demo.ini", &temp );
  VFBlackboardAddGroupBox( &DEMOCONTROL, temp.x, temp.y, temp.width, temp.height,
"Demo Controls");
  ReadButtonDetails( "[run_button]", "demo.ini", &temp );
  VFBlackboardAddButton( &RUN, temp.x, temp.y, temp.width, temp.height, "Run");
  ReadButtonDetails( "[step_button]", "demo.ini", &temp );
  VFBlackboardAddButton( &STEP, temp.x, temp.y, temp.width, temp.height, "Step");
  ReadButtonDetails( "[halt_button]", "demo.ini", &temp );
  VFBlackboardAddButton( &HALT, temp.x, temp.y, temp.width, temp.height, "Halt");
  ReadButtonDetails( "[quit_button]", "demo.ini", &temp );
  VFBlackboardAddButton( &QUIT, temp.x, temp.y, temp.width, temp.height, "Quit");
  ReadButtonDetails( "[options_button]", "demo.ini", &temp );
  VFBlackboardAddButton( &DEMOOPTIONS, temp.x, temp.y, temp.width, temp.height,
"Options");
  ReadButtonDetails( "[next_button]", "demo.ini", &temp );
  VFBlackboardAddButton( &NEXT, temp.x, temp.y, temp.width, temp.height, "Next Page
>>");
  ReadButtonDetails( "[previous_button]", "demo.ini", &temp );
  VFBlackboardAddButton( &PREVIOUS, temp.x, temp.y, temp.width, temp.height, "<<
Previous Page");
  ReadButtonDetails( "[play_button0]", "demo.ini", &temp );
  VFBlackboardAddButton( &PLAY0, temp.x, temp.y, temp.width, temp.height, "Play");
  ReadButtonDetails( "[play_button1]", "demo.ini", &temp );
  VFBlackboardAddButton( &PLAY1, temp.x, temp.y, temp.width, temp.height, "Play");
  ReadButtonDetails( "[play_button2]", "demo.ini", &temp );
  VFBlackboardAddButton( &PLAY2, temp.x, temp.y, temp.width, temp.height, "Play");
  ReadButtonDetails( "[play_button3]", "demo.ini", &temp );
  VFBlackboardAddButton( &PLAY3, temp.x, temp.y, temp.width, temp.height, "Play");
  ReadButtonDetails( "[play_button4]", "demo.ini", &temp );
  VFBlackboardAddButton( &PLAY4, temp.x, temp.y, temp.width, temp.height, "Play");
  ReadButtonDetails( "[play_button5]", "demo.ini", &temp );
  VFBlackboardAddButton( &PLAY5, temp.x, temp.y, temp.width, temp.height, "Play");
  ReadButtonDetails( "[stop_button0]", "demo.ini", &temp );
  VFBlackboardAddButton( &STOP0, temp.x, temp.y, temp.width, temp.height, "Stop");
  ReadButtonDetails( "[stop_button1]", "demo.ini", &temp );
  VFBlackboardAddButton( &STOP1, temp.x, temp.y, temp.width, temp.height, "Stop");
  ReadButtonDetails( "[stop_button2]", "demo.ini", &temp );
  VFBlackboardAddButton( &STOP2, temp.x, temp.y, temp.width, temp.height, "Stop");
  ReadButtonDetails( "[stop_button3]", "demo.ini", &temp );
  VFBlackboardAddButton( &STOP3, temp.x, temp.y, temp.width, temp.height, "Stop");
  ReadButtonDetails( "[stop_button4]", "demo.ini", &temp );
  VFBlackboardAddButton( &STOP4, temp.x, temp.y, temp.width, temp.height, "Stop");
  ReadButtonDetails( "[stop_button5]", "demo.ini", &temp );
  VFBlackboardAddButton( &STOP5, temp.x, temp.y, temp.width, temp.height, "Stop");
  ReadButtonDetails( "[demo id window]", "demo.ini", &temp );
  VFBlackboardAddInt( &DEMOID, temp.x, temp.y, temp.width, temp.height, "demo id.",
0, 0, BLACK, RGB(255,255,232) );
  ReadButtonDetails( "[page no. window]", "demo.ini", &temp );
  DEMOID.Current.Int = demo_id;
  VFBlackboardAddInt( &PAGENUMBER, temp.x, temp.y, temp.width, temp.height, "page
no.", 0, 0, BLACK, RGB(255,255,232));

  /* read in the blackboard details and construct */
  pFile = FileIO_FindIniSection( "demo.ini", "[blackboard]" );
  if (pFile == NULL) return FALSE;
  if (!FileIO_ReadIniLineNumber( pFile, "x", &temp.x )) goto close;
  if (!FileIO_ReadIniLineNumber( pFile, "y", &temp.y )) goto close;
  if (!FileIO_ReadIniLineNumber( pFile, "width", &temp.width )) goto close;
  if (!FileIO_ReadIniLineNumber( pFile, "height", &temp.height )) goto close;
  fclose(pFile);
  VFBlackboardConstruct(&blackboard, temp.x, temp.y, temp.width, temp.height,
BLACKBOARD_NAME, &BbArray);

  if (verbose) VFMessageBox(MB_OK|MB_ICONINFORMATION,"Demo_SetupBlackboard","ended
OK");
  return TRUE;

close:
  VFMessageBox(MB_OK|MB_ICONEXCLAMATION,"Demo_SetupBlackboard","a 'ReadIniLine...'
failed");
  fclose(pFile);
  return FALSE;
}

/*************************************************************************
 *                                                                       *
 *************************************************************************/
int ReadButtonDetails( char *section, char *filename, POS_AND_SIZE *button )
{
  FILE *pFile;

  pFile = FileIO_FindIniSection( filename, section );
  if (pFile == NULL) return FALSE;
  if (!FileIO_ReadIniLineNumber( pFile, "x", &button->x )) goto close;
  if (!FileIO_ReadIniLineNumber( pFile, "y", &button->y )) goto close;
  if (!FileIO_ReadIniLineNumber( pFile, "width", &button->width )) goto close;
  if (!FileIO_ReadIniLineNumber( pFile, "height", &button->height )) goto close;
  fclose(pFile);
  if (verbose) {
          sprintf( temp_buffer, "read: x=%i, y=%i, width=%i, height=%i\nfrom %s in
%s", button->x, button->y, button->width, button->height, section, filename );
          VFMessageBox(MB_OK|MB_ICONINFORMATION,"ReadButtonDetails", temp_buffer );
  }
  return TRUE;

close:
  VFMessageBox(MB_OK|MB_ICONEXCLAMATION,"ReadButtonDetails","a 'ReadIniLine...'
failed");
  fclose(pFile);
  return FALSE;
}

/*************************************************************************
 *        NAME: Demo_ActivatePlayer
 *PURPOSE:        this is the function called when a 'play' button is pressed on
 *                 the demo app. blackboard. If there is a player currently
 *                 displaying a sequence then it stops it (by reseting the
 *                 sequence, it then sets the global variable 'active_player' to
 *                 reflect the new active machine.
 ***************************************************************************/
void Demo_ActivatePlayer( int new_machine )
{
  if (active_player!=NONE) Player_ResetSequence( &players[active_player], TRUE );
  active_player = new_machine;
}


/* ################# *
 * private functions *
 * ################# */


/**************************************************************************
 *        NAME: Demo_TurnToPage
 *PURPOSE: this function loads all the sequence details for a page by
 *        calling ..._LoadSequence for each player.
 ***************************************************************************/
void Demo_TurnToPage( int page )
{
  int machine_id;
  FILE *pFile;


  /* stop any active player */
  if (active_player!=NONE) {
          Player_ResetSequence( &players[active_player], TRUE );
          active_player = NONE;
  }

  /* read still/sequence from .ini file */
  sprintf( temp_buffer, "[page %i]", page );
  pFile = FileIO_FindIniSection( DEMO_INI_FILE, temp_buffer );
  if (pFile == NULL){

  VFMessageBox(MB_OK|MB_ICONEXCLAMATION,"Demo_TurnToPage","FileIO_FindIniSection
returned NULL");
          return;
  }
  FileIO_ReadIniLineText( pFile, "page_type", temp_buffer );
  if ((_stricmp( temp_buffer, "still"))==0) still_page=1;
  else still_page=0;
  fclose (pFile);

  for (machine_id=0; machine_id<6; machine_id++){
          sprintf( temp_buffer, "[page%i_player%i]", page, machine_id );
          if (!still_page) {
                  if (!Player_LoadSequence( temp_buffer, "demo.ini",
&players[machine_id] ))

  VFMessageBox(MB_OK|MB_ICONEXCLAMATION,"Demo_TurnToPage","Player_LoadSequence
returned FALSE");
                  if (verbose){
                          sprintf( temp_buffer, "loaded [page%i_player%i]", page,
machine_id );
                          VFMessageBox( MB_OK|MB_ICONINFORMATION,
"Demo_TurnToPage", temp_buffer );
                          /*sprintf( BbArray.Element[8+machine_id].Text, "Play%i
(page%i)", machine_id, page);*/
                  }
          }
          else {
                  pFile = FileIO_FindIniSection( DEMO_INI_FILE, temp_buffer );
                  if (pFile == NULL)

  VFMessageBox(MB_OK|MB_ICONEXCLAMATION,"Demo_TurnToPage","FileIO_FindIniSection
returned NULL");
                  FileIO_ReadIniLineText( pFile, "still", temp_buffer );
                  fclose(pFile);
                  ImageLoadBMP( players[machine_id].image, temp_buffer );
                  VFBlackboardImageShow(blackboard, players[machine_id].image);
          }
    }
```

```
          active_player=NONE;
          PAGENUMBER.Current.Int = page_number;
          VFBlackboardWriteElements(blackboard,&BbArray);
}

/* ### 'player' related functions ### *

 /************************************************************************
 *         NAME: Player_Init
 *PURPOSE:          Initialises a player, by constructing and intialising its
 *                  constituent parts. The details are ,read from an .ini file
 *NOTES:            returns TRUE unless the load fails in any way.
 *                  the BMP grabber still needs intialised before use.
 ************************************************************************/
int Player_Init( char *section, char *filename, PLAYER *machine )
{
  FILE *pFile;
  char name[10];
  int x_coord, y_coord;

  /* construct grabber, stream and image, and intialise the stream */
  VFGrabberConstructBMP( &machine->bmpgrabber );
  VFImageStreamConstruct( &machine->bmpstream );
  if(( machine->image=ImageConstruct(GS1,256,256))==NULL) DiagnoseError(gerr_flag);
  VFImageStreamInitialise( machine->bmpstream, machine->bmpgrabber, SOURCE);

  /* intialise the image, using data from the .ini file */
  pFile = FileIO_FindIniSection( filename, section );
  if (pFile == NULL){

  VFMessageBox(MB_OK|MB_ICONEXCLAMATION,"Player_Init","FileIO_FindIniSection
returned NULL");
          return FALSE;
  }
  if (!FileIO_ReadIniLineText( pFile, "name", name )) goto close;
  if (!FileIO_ReadIniLineNumber( pFile, "x", &x_coord )) goto close;
  if (!FileIO_ReadIniLineNumber( pFile, "y", &y_coord )) goto close;
  fclose(pFile);
          ImageSetName( machine->image, name );
          ImageSetLocation( machine->image, x_coord, y_coord ); /* #### change x&y
type #### */
          ImageSetDisplaySize( machine->image, 256, 256 );

  return TRUE;

close:
  VFMessageBox(MB_OK|MB_ICONEXCLAMATION,"Player_Init","a 'ReadIniLine...' failed");
  fclose(pFile);
  return FALSE;
}

/************************************************************************
 *         NAME: Player_LoadSequence
 *PURPOSE:          loads all the details necessary to play an image sequence into
 *                  a 'player', then intialises the player's bitmap grabber with
 *                  the file details and displays the blank image.
 *NOTES:            returns TRUE unless the load fails in any way.
 ************************************************************************/
int Player_LoadSequence( char *section, char *filename, PLAYER *machine )
{
  FILE *pFile;

  /* read details from .ini file */
  pFile = FileIO_FindIniSection( filename, section );
  if (pFile == NULL){

  VFMessageBox(MB_OK|MB_ICONEXCLAMATION,"Player_LoadSequence","FileIO_FindIniSection
returned NULL");
          return FALSE;
  }
  if (!FileIO_ReadIniLineText( pFile, "sequence", machine->sequence )) goto close;
  if (!FileIO_ReadIniLineText( pFile, "algorithm", machine->algorithm )) goto close;
  if (!FileIO_ReadIniLineNumber( pFile, "colour_resolution", &machine->colour_depth
)) goto close;
  if (!FileIO_ReadIniLineNumber( pFile, "start", &machine->start_frame )) goto
close;
  if (!FileIO_ReadIniLineNumber( pFile, "length", &machine->max_frames )) goto
close;
```

```
  fclose(pFile);

  /* create path for image sequence (using sequence, algorithm and resolution */
  switch( machine->sequence[0] ){
          case 'c':       sprintf( temp_buffer, "seq\\claire\\" ); break;
          case 'd':       sprintf( temp_buffer, "seq\\dummy\\" ); break;
          case 's':       sprintf( temp_buffer, "seq\\sman\\" ); break;
          case 'm':       sprintf( temp_buffer, "seq\\missa\\" ); break;
          default:        sprintf( temp_buffer, "seq\\" ); break;
  }
  switch( machine->algorithm[0] ){
          case 'n':       strcat( temp_buffer, "none" ); break;
          case 's':       strcat( temp_buffer, "simp" ); break;
          case 't':       strcat( temp_buffer, "trnc" ); break;
          case 'p':       strcat( temp_buffer, "pert" ); break;
          case '1':       strcat( temp_buffer, "saf1" ); break;
          case '2':       strcat( temp_buffer, "saf2" ); break;
          case 'f':       strcat( temp_buffer, "flyd" ); break;
          default:        break;
  }
  if (machine->algorithm[0]!='d') sprintf( temp_buffer, "%s%i%c", temp_buffer,
machine->colour_depth, '\\' );
  strcpy( machine->path, temp_buffer );
  /* set blank screen to the new constant value */
  sprintf( machine->blank_screen, "blank.bmp" );
  /* create image filename root (from sequence, colour_depth and algorithm ) */
  sprintf( machine->root, "%c%i%c", machine->sequence[0], machine->colour_depth,
machine->algorithm[0] );

  Player_ResetSequence( machine, TRUE );
  return TRUE;

close:
  VFMessageBox(MB_OK|MB_ICONEXCLAMATION,"Player_LoadSequence","a 'ReadIniLine...'
failed");
  fclose(pFile);
  return FALSE;
}

/*****************************************************************************
 *        NAME: Player_ResetSequence
 *PURPOSE:        displays the player's 'blank' image, resets the grabber and
 *                the frame counter.
 *****************************************************************************/
void Player_ResetSequence( PLAYER *machine, int blank )
{
  char image_name[20];
  int process_id;

  if (blank){
          if (verbose){
                  sprintf( temp_buffer, "about to load %s", machine->blank_screen
);
                  VFMessageBox(MB_OK|MB_ICONINFORMATION,"Player_ResetSequence",
temp_buffer );
          }
          ImageLoadBMP( machine->image, machine->blank_screen );

          /* draw image title */
          if (machine->sequence[0]=='d'){
                  sprintf( image_name, "\n\n%s", machine->sequence );
                  MoveTo( 40, 120 );
                  DisplayText( machine->image, image_name );
          }
          else {
                  switch( machine->algorithm[0] ){
                          case 'n':       process_id = machine->colour_depth;
break;
                          case 's':       process_id = machine->colour_depth+10;
break;
                          case 'p':       process_id = machine->colour_depth+20;
break;
                          case '1':       process_id = machine->colour_depth+30;
break;
                          case '2':       process_id = machine->colour_depth+40;
break;
                          case 'f':       process_id = machine->colour_depth+50;
break;
```

```
                            case 't':         process_id = machine->colour_depth+60;
break;
                            default:          process_id = 0; break;
                   }
                   sprintf( image_name, "%c\n\n%s", translation[process_id],
machine->sequence );
                   MoveTo( 40, 120 );
                   DisplayText( machine->image, image_name );
                   if (show_details){
                            sprintf( temp_buffer, "\n\n\n%s\n%i bits", machine-
>algorithm, machine->colour_depth);
                            DisplayText( machine->image, temp_buffer );
                   }
          }

          VFBlackboardImageShow(blackboard, machine->image);
  }
  VFGrabberInitialiseBMP( machine->bmpgrabber, machine->path, machine->root,
machine->start_frame, machine->max_frames);
  machine->frame_counter = machine->start_frame;
}

/****************************************************************************
 *        NAME: Player_DisplayNextFrame
 *PURPOSE:         Displays the next frame of the sequence currently loaded into
 *                 the player's.
 ****************************************************************************/
int Player_DisplayNextFrame( PLAYER *machine )
{
  if(!VFImageStreamGrab(machine->bmpstream, machine->image)){
          VFMessageBox(MB_OK|MB_ICONEXCLAMATION,"Player_DisplayNextFrame","Grab
error");
          return FALSE;
  }

  VFBlackboardImageShow(blackboard, machine->image);
  machine->frame_counter++;
  return TRUE;
}

/****************************************************************************
 *        NAME: Player_Destroy
 ****************************************************************************/
void Player_Destroy( PLAYER *machine )
{
  ImageDestroy( machine->image );
  VFGrabberDestroyBMP( machine->bmpgrabber );
  VFImageStreamDestroy( machine->bmpstream );
}


/* ######## */


/****************************************************************************
 *                 NAME: FileIO_FindIniSection
 *PURPOSE: returns a pointer to the line after the specified header in the
 *        specified '.ini' file.
 *NOTE: this function opens the file that is passed to it, the calling
 *        function is resposible for closing it.
 ****************************************************************************/
FILE *FileIO_FindIniSection( char *pFilename, char *pSection )
{
  FILE *pIni_file;
  int end = FALSE;
  char temp_buffer[80];

  /*if (g_diagnose && !g_graphics){
          printf(" FileIO_FindIniSection: called...\n");
          printf(" FileIO_FindIniSection: searchine for '%s' in '%s'\n", pSection,
pFilename);
  }*/

  pIni_file = fopen( pFilename, "r" );
  if (pIni_file==NULL) {
          sprintf( temp_buffer, "ini file '%s' not found", pFilename );
          VFMessageBox( MB_OK|MB_ICONEXCLAMATION, "FileIO_FindIniSection",
temp_buffer );
```

```
          pIni_file = NULL;
          return pIni_file;
  }
  /*if (g_diagnose && !g_graphics) printf(" FileIO_FindIniSection: opened file
okay\n");*/

  while ( (!end) && (_stricmp( temp_buffer, pSection )!=0) ){
          if (FileIO_ReadNextLine( temp_buffer, pIni_file )){
                  /*sprintf( buffer, " FileIO_FindIniSection failed - section %s
not found\n", pSection );*/
                  end = TRUE;
          }
  }

  if (end) {
          pIni_file = NULL;
          sprintf( temp_buffer, "couldn't find %s in %s", pSection, pFilename );
          VFMessageBox(MB_OK|MB_ICONEXCLAMATION,"FileIO_FindIniSection",
temp_buffer );
          }
  /*else if (g_diagnose && !g_graphics) printf(" FileIO_FindIniSection: found
it\n");*/
  return( pIni_file );
}

/**************************************************************************
 *        NAME: FileIO_ReadIniLineNumber
 *PURPOSE: checks the name at the start of the line and if its is correct
 *        assigns the int then returns TRUE, else it returns FALSE.
 *************************************************************************/
int FileIO_ReadIniLineNumber( FILE *pFile, char *name, int *pVariable )
{
  char temp_line[81]="";
  char seps[]       = " =";
  char *token;

  /*if (g_diagnose) printf(" ReadIniLine: called... (looking for %s)\n", name);*/
  if (FileIO_ReadNextLine( temp_line, pFile )) return FALSE;
  token = strtok( temp_line, seps );
  if ( _stricmp( token, name)==0 ){
          token = strtok( NULL, seps );
          sscanf( token, "%u", pVariable );
          /*sprintf( temp_line, "read: %s = %u", name, *pVariable );
          VFMessageBox(MB_OK|MB_ICONINFORMATION, "FileIO_ReadIniLineNumber",
temp_line);*/
          return TRUE;
  }

  sprintf( temp_line, "couldn't find '%s'", name );
  VFMessageBox(MB_OK|MB_ICONEXCLAMATION, "FileIO_ReadIniLineNumber", temp_line);
  return FALSE;
}

/**************************************************************************
 *        NAME: FileIO_ReadIniLineText
 *PURPOSE: checks the name at the start of the line and if its is correct
 *         assigns the string then returns TRUE, else it returns FALSE.
 *************************************************************************/
int FileIO_ReadIniLineText( FILE *pFile, char *name, char *dest )
{
  char temp_line[81]="";
  char seps[]       = " =";
  char *token;

  /*if (g_diagnose) printf(" ReadIniLine: called... (looking for %s)\n", name);*/
  if (FileIO_ReadNextLine( temp_line, pFile )) return FALSE;
  token = strtok( temp_line, seps );
  if ( _stricmp( token, name)==0 ){
          token = strtok( NULL, seps );
          sscanf( token, "%s", dest );
          return TRUE;
  }

  sprintf( temp_line, "couldn't find '%s'", name );
  VFMessageBox(MB_OK|MB_ICONEXCLAMATION, "FileIO_ReadIniLineText", temp_line);
  return FALSE;
}
```

```
/*************************************************************************
 *         NAME:    FileIO_ReadNextLine
 *PURPOSE: reads the next non-empty line of text from a file. The line is
 *         read from whatever file the passed pointer is pointing to, and
 *         from the position of that pointer within the file. If there is another
 *         non-empty line in the file the function copies (minus any carriage
 *         returns) to the passed string and returns FALSE, if there are no more
 *         lines in the file containing text the function returns TRUE.
 *************************************************************************/
int FileIO_ReadNextLine( char *buffer, FILE *pFile )
{
  char line[81]="", temp_letter;
  int end_of_line=FALSE, found_text=FALSE;
  byte bytes_read;

  while (!end_of_line) {
          bytes_read = (byte)fread( &temp_letter, 1, 1, pFile );
          if ( bytes_read != 1 ) {
                  strcpy( buffer, line );
                  /*if (g_diagnose && (!g_graphics)) printf(" FileIO_ReadNextLine:
read line '%s', returning TRUE\n", line );*/

  VFMessageBox(MB_OK|MB_ICONEXCLAMATION,"FileIO_ReadNextLine","found unexpected
EOF");
                  return TRUE;
          }
          else {
                  if ( temp_letter != 10 ) {
                          sprintf( line, "%s%c", line, temp_letter );
                          found_text=TRUE;
                  }
                  else if ( found_text ) end_of_line = TRUE;
          }
  }

  strcpy( buffer, line );
  /*if (g_diagnose && (!g_graphics)) printf(" FileIO_ReadNextLine: read line '%s',
returning FALSE\n", line );*/
  return FALSE;
}


/* ################# *
 * end of demo01.cpp *
 * ################# */
```