UNIVERSITY OF HERTFORDSHIRE

School of Information Sciences

BACHELOR OF SCIENCE IN COMPUTER SCIENCE
(Information Systems)

Project Report

MEASURING IMAGE QUALITY

M. G. Ross

April 1996

Abstract

The current methods for measuring the quality of computer stored digital images are subjective. There are a multitude of different file formats available for the storage of such images, each with its own unique features which may work for or against it. For an average user or system developer, to decide on the most suitable format for their purpose requires a knowledge of the available file formats, their features and how they affect the quality of images, as well as the kind of image data they will be storing.

This places undue pressures on the user which my lead to a format unsuitable for them and the application. Therefore it is important to make the right choice first time, while the opportunity is still open.

In this project I have set out to identify any methods currently used in related industries to measure the quality of an image stored in a wide variety of these file formats and how they can be implemented successfully. From this information, and details on the specifics of popular file formats and their compression methods, I have carried through the ideas, incorporating my own opinions, to formulate suggestions on how this could be done on a wider general level.

To fortify my understanding of the problems associated with file formats and how their compression and storage methods affect image quality, a software component to this project has involved writing a graphics library to allow the conversion between a number of the most popular graphics formats.

Acknowledgements

I would like to thank the following people for their help in the production of this project:

Alan Boyd, project supervisor, without whose help and support throughout, this project would not have been possible.

Jeffrey Glover, of ASAP Inc., for his opinions on the measurement of image quality.

Nik Sutherland, of the National Remote Sensing Centre, for information on the image conversion problems encountered by the NRSC as well as opinions on quality measurement.

Nick Efford, of the University of Leeds, for further information regarding image quality measurement relating to medical imaging and motion tracking.

Andy Wells, of ERDAS, for providing contacts in the industry and opinions on image quality measurement.

Simon Boden and Neil Dudman for their assistance in testing the software application developed in the project and providing feedback.

TABLE OF CONTENTS

CONTENTS PAGE

Appendices

TABLE OF FIGURES

PAGE

1 Introduction

1.1 Project Motivation

As a user of graphics file formats and conversion applications I have been interested in this field since my interest in computing began. My own experiences of using graphic images for course-work has led me to ponder many questions as to why there are so many formats and methods for storing these images. This project has given me the opportunity to explore the world of graphics files to find out the answers to my questions.

My knowledge of this field at the start of the project was casual. I knew generally about bitmaps without knowing anything specific about the formats, compression techniques and overall structure of the graphic images I was using. As this is a subject I am interested in making my career in, measuring the 'quality' of images and how this can be affected by the right or wrong choice of a file format seemed a natural choice of study which I knew would be both challenging and interesting.

The learning curve embarked on has been considerably steeper than previous work I have undertaken. The software component constitutes my first true software development culminating in a final product. My previous knowledge of the C language did not cater for the scale of this work, and my skills in Pascal, as used in Borland Delphi, were only of a basic level. Through the development I have learnt everything necessary about these languages and how they can be applied to creating file conversion software.

From the theory aspect, I have done much research into the principles of image storage and its related areas including compression and decompression, colour spaces and conversion between colour systems, image displaying, conversion between file formats and some advanced techniques used to enhance compression ratios and allow such features as real-time full-motion video.

1.2 Aims And Objectives

The core objectives which have been designated as fundamental to the project are:

- *Identify, understand and describe a range of industry-based methods for quantitatively measuring the quality of an image represented in various graphic file formats.*

  Information gathered from related industries as well as from other image processing sources will be described with its relevance to this study.

- *Suggest methods for measuring an image's quality in varying graphic file formats.*

  Using the information gathered as a base, I will build up my own ideas on ways 'quality' can be identified and measured fairly between different formats and techniques.

- *Research, understand and describe current popular static graphic file formats, the compression methods utilised as well as colour spaces etc.*

  Emphasis will be on the common compression and decompression techniques used widely, and how their use impacts the quality of the image representation, not just in visual terms, but overall efficiency and suitability.

- *Gain an understanding of relevant advanced algorithm concepts, such as JPEG, MPEG, and Fractal compression.*

  Although not covered in great detail, an understanding of these advanced representation methods is useful in the context of the project.

- *Research Windows API programming.*

  Although the software will involve little direct API programming, it is useful to know about the facilities and restrictions I will be working with.

- *Learn Borland Delphi and ObjectPascal.*

  To be learnt specifically for the project.

- *Use shareware JPEG and GIF encoding/decoding routines to create routines which allow transfer to and from the Microsoft Windows BMP format.*

  The BMP format will be used as the central format by which the other supported formats will be converted to and manipulated.

- *Write ZSoft PCX encoding/decoding routines to and from Microsoft Windows BMP format.*

  Along with the JPEG, GIF and BMP routines, a 16-bit Dynamic Link Library compatible with Microsoft Windows 3.1 or greater will be constructed with high-level format conversion routines accessible to external software.

- *Design and implement a user-interface with Borland Delphi which makes use of the routines.*

  This will provide a front-end to the graphics library created in the objectives above. This application will allow the conversion between JPEG, GIF, PCX and BMP formats.

In addition, the advanced aims which are desirable if time is permitting are:

- *Implement tools for clipboard transfer of image selections, as well as simple manipulation tools covering fixed rotation (i.e. 90, 180 or 270 degrees), scaling, horizontal and vertical axis flipping.*

  Of these extra utilities the ability to use the clipboard will increase the compatibility of the application. Therefore, it is more important than magnification, rotation and axis-flipping, which are not essential, but enhance the functionality of the software.

- *Construct an online help system within the software package.*

  Although this will mainly contain procedural information on how to use the application, it would provide software testers with an instant information source if problems are encountered using the system.

1.3 Report Structure

Chapter 2 introduces the major factors which bias the measuring of image quality, as well as listing the industry sources used to collect information. My opinions on the information described is contained in Chapter 3. File formats are discussed in Chapter 4, in general terms with examples from file formats. In Chapter 5 I follow-up the work from the previous chapters by suggesting methods in which image quality could be measured whilst avoiding the bias factors mentioned in Chapter 2. Chapter 6 is described below. Finally, in Chapter 7, I conclude by evaluating the work I have done, the problems I have encountered, the areas of future work which could be done, and a self-appraisal of my success in attaining the objectives and aims and overall management of the project.

The technical documentation for the software component of this project is contained in Chapter 6. This includes the design principles, structure of the application, problems encountered and details of how they were overcome. Specific details on how to use the application can be found in the on-line help system available through the software. An evaluation of my success in writing this software is contained in Chapter 7, as are future improvements which could be made. Appendix 2 contains the source code of the application written by myself. The entire source code is not included, as a majority of the library low-level functions were taken from the previously mentioned shareware packages.

A project plan, in the form of a Gantt Chart can be found in Appendix 1. This plan outlines the initial plan at the offset of the project. The evaluation in Chapter 7 discusses how reality has matched up to the plan.

2 Description Of Current Image Quality Measures

2.1 Background

In order to fully appreciate the requirements of an accurate file format measuring system, it is important to have details on the following:

- Current methods used in industry for performing image quality measurement.
- An understanding of the formats available.
- Implementation details of the main formats used in representing an image.

The latter two will be covered in the following Chapters. This Chapter is concerned with procedures used by organisations in industry which deal with the difficulty of file format selection.

The first thing to ask is why is it necessary to measure them so accurately? Looking at most images, one can usually tell which provides the best quality just by looking. The clear answer here relates to perception. One person looking at a set of images in different formats may think format *A* to be better than formats *B* or *C* because they can see the colour definition better. Another person may disagree on the grounds that *B* is of a higher resolution, and is therefore 'better'. Yet another person could be colour blind, making the results even less accurate and reliable. This is the first problem encountered: each individual has his or her own unique perception. We cannot rely on a method whereby everyone involved could have differing opinions. This does not help judge formats scientifically and fairly. Many factors which are beyond our control affect the way we view image representations. Some of the more distinct ones include:

- Equipment – Using a low-quality monitor with a poor graphics card which can only display, say, 16 colours at 320x200 pixels will place an unfair disadvantage on all the formats involved in the test. Most importantly, however, will be the effects on formats which have the test image stored as 24-bit and in a resolution of 1024x768 (format *A* in this example) or higher. The scaling down of the colours to those available will give undesirable results and is likely to result in an unsuitable display. Now if the image was displayed again from a format which can only store 256 colours at 640x480 pixels (format *B*), the down-sizing and down-sampling required is less drastic and hence, the displayed image will be closer to the actual file stored representation. This example would give the second format *B* a clear and unfair disadvantage. If state-of-the-art equipment was available for the test, the results would obviously be turned around with the 24-bit high resolution format *A* utilised to the full with the lower standard format *B* distinguished as a poor format for high-quality representation.

- Human Vision – Many people require man-made aids to help their vision nowadays. As we all are unique, the vision quality we each possess varies widely. This means that we cannot rely on our own vision to systematically judge image representations. Many of the formats of today can produce qualities so high that the human eye cannot appreciate the detail level. As an example, experiments have shown that humans can discriminate about 200 colours across the spectrum if placed next to each other (Jackson, MacDonald and Freeman, 1994). This difference can be exploited by these formats without decreasing the visual quality to the naked eye. Just looking at an image will not necessarily enable us to notice differences which, being so insignificant to the eye, are not identified by our visual system.
- Environmental Conditions – Lighting is the main factor in this group. Our perception of an image representation will be swayed to some degree by the lighting in the room where the viewing is taking place. If it is a bright room and we have entered from a dark room previously, it is likely our eyes will take a while to adjust to the new lighting. This will play a big part when looking at the pictures on-screen. Other factors such as noise and smell could also play a rôle, to a lesser extent, in that they may affect the concentration of the viewer.
- Viewer Bias – For one reason or another, an individual may have pre-conceived ideas about which format they believe will perform better. This already places bias towards the format before they have even been seen. Ensuring objectivity would be important and difficult in such tests.

So if a fair method for measuring such quality is to be found, it cannot involve the use of viewing the image with the naked eye. A scientific approach is required which filters out the subjective, bias-factors described above.

2.2 Information Sources

To get an understanding of how these image format problems are circumnavigated in industry, I have selected a range of relevant companies to approach and request information from. To gain as wide a viewpoint as possible I have not restricted my information requests to any particular type of industry. The organisations and individuals I have requested information from are:

- ASAP Inc. – Jeffrey Glover
- Atlas Image Factory
- BBC Television: ‘Sky At Night’ and Weather Centre
- Centre of Medical Imaging Research (CoMIR) – University of Leeds – Nick Efford
- Erdas
- Imaging Systems Lab; Centre for Imaging and Pharmaceutical Research (CIPR); Magnetic Resonance Imaging Group (MRIG); Teleradiology & Medical Imaging, Nynex
- Laser-Scan Limited; Visioneering Research Limited (VRL)
- NASA Information Services; NASA Jet Propulsion Laboratory (JPL); NASA Remote Sensing Unit (RSU)
- National Remote Sensing Centre (NRSC)
- Silicon Graphics, UK
- United States Naval Observatory (USNO)
- WXP Weather Project; National Center for Atmospheric Research (NCAR)

I have requested information regarding the necessity of image quality and storage using the more popular formats (or, indeed, any others used) with respect to their application. The purpose of this is to paint a picture of the current state of the industry so I am able to form my own suggestions as to how this could be done.

2.3 Feedback

Three responses were received with regards to my information request.

1. *Jeffrey Glover, of ASAP Inc.*, stated that if speed was more important for an application (for example World Wide Web graphics) then a format is chosen on this basis. As 24-bit colour at 1600x1200 resolution is rarely required for this application, only disadvantages would result in its use. A majority of the World Wide Web user-base would not appreciate the large graphic files and have no requirement for such high quality.

   For quality-critical applications, only lossless compression will do. The possibility of losing some detail, even if it is too small to see with the naked eye, gives rise to problems if the image is then processed and enhanced by a computer. Disparities and noise could then be amplified to the extent of affecting the image visually.

2. *The National Remote Sensing Centre*, a company involved in the production of maps generated from remote sensing scans taken by orbiting satellites stated that user judgement is predominant. For the majority of their applications, colour plays a major rôle. The example quoted involves infra-red scans of an area, whereby an experienced user can map the outputted false colours from the scan to bands of infra-red intensities using *Erdas Imagine* or *ER Mapper*. No scientific method is utilised to judge the accuracy of the user's decisions, or provide assistance along the way.

   The file formats used by the NRSC also provided further information. As pointed out, all formats which can handle 24-bits per pixel of colour information should be on a par when representing the colour, in whatever form. The problem arises, however, when the file needs to be interpreted by many applications on different platforms. Most, if not all, applications of this type include their own built-in proprietary image format. Transferring from one type to another can raise problems. Of course, plug-in filters are available for most of these which allow the transfer to a common format suitable for all involved platforms and software. In the experience of the NRSC these often fail to sustain the quality required, and so are not used. As an aside, I too have noticed this with certain pieces of software, such as early versions of Microsoft Word, which includes a low-quality GIF import/export filter which is of little practical use for most purposes. Instead, the images are stored in the application's built-in format at the NRSC which can be guaranteed to maintain the detail. As a consequence of this, if the image is required in all the varying application-specific formats, the image processing steps have to be carried out separately on each application. This is not a viable option, due to the cost and time resources required, so is rarely undertaken.

3. *Nick Efford, of the School of Computer Studies at the University of Leeds*, is involved with *The Centre of Medical Imaging Research (CoMIR)*. For storing their image databases, they use lossless compression formats. They have not concerned themselves on the issues of file formats as they feel it is convenient to purchase further storage devices as required. In this case, because all of the images are stored lossless, there is little point in analysing the differences between the formats as they will be almost identical (except in the technicalities of the format layout). It must be noted at this point that not all lossless formats are capable of storing 24-bit colour, GIF being an example (see Chapter 4).

   They also do research into object-tracking. Moving images are stored as lossy, as it is more important to gain higher compression than better quality. All the concern with this application is focused towards identifying the objects. Subtle differences in the quality of the image from frame-to-frame is not important as only the object outlines need to be recognised. The advantage of not requiring a lossless format for this application is the large savings on storage space, and the use of a simple lossy compression algorithm ensures the motion can be tracked in real-time.

   When dealing specifically with medical images, the quality of the image is not paramount. He says they are aware of the significance of high quality in this area, but do little to ensure their images are of the highest quality as they feel it is not necessary. From their point of view, the medical profession is very wary of imaging in general, especially in the United States where concerns of lossy compression affecting patient diagnoses are higher. In England, however, the majority are content with the quality currently attained.

3 Personal Opinion On Image Quality

The best way to approach this is to analyse the responses I have received and described in Chapter 2.3. From there, I can build up my own ideas and paint a bigger picture of how I feel about the factors for and against this constantly developing field.

3.1 ASAP Inc.

I feel that Mr. Glover failed to answer the questions I put to him directly. His response was somewhat vague and did not take into consideration the necessity for a scientific measuring system which could be used and justified by any user, expert or otherwise. To some extent, I agree with his notion that one can sometimes tell which format is most suitable for a certain image and application. However, for this to be so requires knowledge of the available formats and the application in-hand. For someone unfamiliar with either or both of these, visual perception and knowledge of available formats is not a satisfactory method.

What is required is an unbiased system which can be used by anyone that allows the various file formats to be graded against each other for a particular type of application. No knowledge of any file formats or the application in-hand would be necessary, other than the basic requirements of the system (such as medical, recreational and so on).

We cannot assume that the user will have any knowledge of the formats available, and therefore most suitable for the application. In fact, the final results may be better if the user has no understanding of such formats, as the removal of preference for one format may lead to a better choice being made. Only a scientific, quantitative method can guarantee better results every time. It is important to make the correct decision at an early stage, as it could be too late if software for the application has been developed with a certain format in mind. If, for example, it is later discovered that the chosen format will consume too much disk space per image, or requires more processing power than can be harnessed in real-time, the system will be severely restricted the longer it is used. Tough decisions would need to be made as to whether or not it should be redesigned with a new format in mind, better equipped for the tasks ahead. The system updates could be costly and time-consuming, especially if it has been redistributed to many customers. All of these problems should never need to occur.

3.2 National Remote Sensing Centre (NRSC)

In this response the issue did not lie so much with ensuring image quality is of the optimum level when stored, but converting between the myriad formats efficiently whilst maintaining the quality. Until better quality plug-ins are made available which allow easier transference, this is likely to remain a problem. Due to the wide range of facilities offered by different software applications, to convert from one to another would usually require a filter to be specifically built for that purpose. In some cases, it may not be possible to convert at all, if the features of one package are not supported in another.

This is a fundamental problem in the image storage dilemma which can only practically be solved through collaboration between the software manufacturers, ensuring file formats are interchangeable. Currently there are already some alliances between players in this market, but it is a far cry from the co-operation required to eliminate the problem. While this continues, the pool of formats grows larger and the possibilities for conversion become endless.

In relation to the quality of an image, the NRSC usually relies on the capacity of its employees to judge its quality. As the people involved with these remote sensing images are all well experienced with the application, there would be little advantage to formally representing this system. Situations like these demonstrate that a quantitative method for grading formats is not always necessary. In some cases it would only cause problems, as all the relevant employees would need training under the new system. Clearly, the kind of system I am suggesting is more suited to scenarios where the users will have less knowledge of the overall system, or those systems where the quality of an image is paramount and cannot be accurately judged by visual perception alone.

3.3 Centre Of Medical Imaging Research (CoMIR)

This case highlights differing viewpoints. On the one hand, Mr. Efford is involved in the Computer Science aspect, as a member of the academic staff at the University of Leeds. Generally, the technical aspects of the technology used is of more importance, and how it can best be improved and utilised to the full. On the other hand, his links to medical imaging provide a perspective from the medical point of view which are more concerned with the contents of the image, rather than how it was captured.

However you look at it, there will be a merging to some degree between them. From the Computer Science stance, it is important to know what kind of things the images are representing so suitable hardware and software can be used which copes with the demands. Inversely, the medical staff must also know of limitations with the hardware and software which may give rise the noise and artefacts on the images. The definition of this division depends on the particular example, as it will vary with many factors, but in an ideal world, the computer scientists should be aware of the application as much as the medical staff are of technical capabilities.

With regards to the quality of medical images, the next step depends on what is to happen to the images after they are acquired. If a doctor is to look at them and make a judgement, the results would be less erroneous than if a computer was used to post-process and highlight object outlines. The computer is much more likely to pick up on disparities and noise generated by the compression technique, especially as it has little or no knowledge of the human anatomy. In the design of the system this will be accounted for and acted on accordingly, but the distinction must be made that this could vary widely. With images only being viewed by the doctor, it is highly likely that lossy compression may suffice. Most of these techniques are capable of significant compression ratios without affecting visual quality. This may be all that is required, so lossless algorithms would be wasted, as well as storage space. If the images were parsed through an image processing system to extract important features, it would be vital that lossless techniques were used. After this stage, the doctor then views the result, and a lossy method used for the final output.

On a different subject, motion tracking enables much of the detail in a set of images to be disregarded. As we are only interested in the shape of the object, the cheapest possibility would be binary thresholding. This would allow extensive compression which would make the hardware and software requirements of a motion-tracking system less advanced. All the processing power could then be handed to the artificial intelligence engine which processes the motion. Image format choices would be far wider in this case, as virtually all of them are capable of storing binary image data (although some are better suited).

4 File Formats And Compression Methods

With the proliferation of new computer systems and applications to run on them, a multitude of file formats has also been introduced by manufacturers. Each of these has been designed for either a specific software application or for a certain type of use.

In this chapter I will discuss the main types of file formats, how and what they are capable of storing, and any methods used for compressing the represented data. The information contained herein can then be used as a reference for the suggestions and recommendations I have made in Chapter 5. This discusses quantitative methods for fairly measuring an image format's ability to represent certain types of data as accurately as possible. It should be noted that I have concentrated primarily on the file formats specifically involved with this project, but other types are mentioned to provide a broader view of what is available.

4.1 Format Types

Image formats can be broken down into three main categories: *Vector*-based, *Bitmap*-based, and *Metafile*-based. Each of these makes a different approach in representing an image, and stores different types of data accordingly. In addition, and less importantly to this project, there are also *Scene Description* and *Animation* types.

4.1.1 Vector

Representation is performed by retrieving vector information for the image as *objects*. Initially, only lines could be represented, but now the capability is there to build up complex objects from different kinds of curves and polygons. All the simple vectors objects can be stored as a set of mathematical statements with attached colours and other information which can build up more complex objects.

The primary use of such formats is in Computer-Aided Design. Vector formats are well-suited to storing mechanical and other technical drawings as, generally, they are constructed from simple vectors and curves. 3-D, labelling and hatching of objects is also possible within some of the more advanced vector formats.

Due to the highly compact method of storage, compression would yield little on most images on this type of format. For example, all of the compression techniques discussed in this chapter (see Chapter 4.2) are unsuitable for vector compression. However, if it were used on complex files, an effective method would involve a dictionary-based algorithm storing frequently used mathematical statements, making references as required in the same way as a colour palette look-up table (CLUT).

This type of format would be unsuitable for storing real-world scenes, as each item of the scene would need to be broken down into a set of vectors, even before colour has been considered. This would require much processing and artificial intelligence to identify possible object outlines. In addition, as the image is described as a set of formulae, exact display cannot be guaranteed to be identical on different machines. Different machines may interpret the commands in slightly different ways, as with a non-procedural programming language. Usually, vectors are drawn in the same original order they were created in, so complex drawings may give rise to a 'building block' effect as the drawing is created before your eyes. Fig. 4.1 demonstrates a vector-representation of a chair.

4.1.2 Bitmap

This, by far the most widely used type, represents an image at a bit-by-bit level. A mapping of an image is made and transformed into pixel values which can then each be assigned a colour value. Bitmap formats are ideal for storing real-world photographic or other complex images, as there is no requirement to classify objects, and as each pixel can be easily converted to an array value, they are ideally suited to programming methods. These reasons have made them popular with many differing applications. The opportunity for storing photographic images digitally has been utilised by many applications in both industry and recreation, and is becoming ever more present as a part of everyday life.

The first thing to note about bitmaps is the large increase in data required to represent a scene. For an accurate depiction of, say a photograph, a resolution of at least 640x480 (307,200 pixels) would be required. At least 256 colours (8-bit) would be necessary for realism, so the minimum storage required for the raw data is 307,200 bytes without even considering the colour palette entries. Currently, resolutions up to 1600x1200 are possible, at up to 128-bit true colour: requiring 30,720,000 bytes to store the image uncompressed!

Storage space and image display memory is of a much higher concern with bitmap data than it was with vector data. Due to this, numerous sophisticated methods have been devised over the years since the inception of bitmaps, which can drastically reduce the storage required for such images. It is the improvements in computer technology which has made bitmap formats possible. As it advances, and prices fall, the specifications are on the rise. The resources required to display high resolution images is increasing, as is the need for effective reduction in storage size. A detailed discussion of some of the common types of bitmap compression can be found in Chapter 4.2.

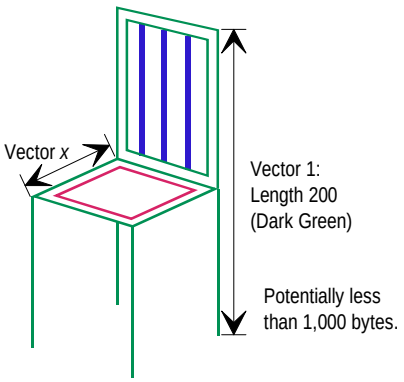

Fig. 4.1: Vector Representation Of A Chair

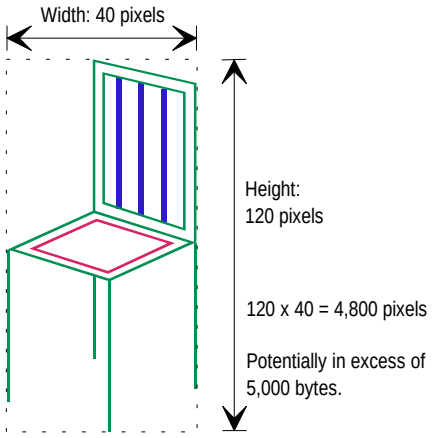

Fig. 4.2: Bitmap Representation Of A Chair

Figs. 4.1 and 4.2 shows vector- and bitmap-oriented representations of a chair object. An interesting point to note is that the vector version is concerned only with the chair itself, whereas the bitmapped version does not identify objects and thus the white background is also stored as bitmap information. In this case, if the chair were to be overlaid on a background image of, say a kitchen, the bitmap version would require further manipulation to prevent the background interfering with the output.

This remains a significant disadvantage of bitmaps. Some of the industries where bitmaps are used require manipulations of such data to identify object outlines and perform other operations to extract certain information from an image. Quite how this is done varies a great deal, depending on the necessity for accuracy. The image processing techniques used can be done either by an operator, or by the computer itself. Of course, relying on the computer to make judgements on an image it has little knowledge of can be risky. The context of an object could be misconstrued by a machine without detailed information being programmed in to tell it of, for example, the human anatomy. Some very complex systems have been developed to try and make this as effective as possible, but at the end of the day, computers are unlikely to come close to the effectiveness of an experienced user's perception.

In a lot more cases, the image processor has been provided with a multitude of tools which allow every conceivable mathematical alteration on an image, so that the information required can be tweaked manually, using the computer as a 'slave'. Details which may not be visible from the standard picture can be gleaned by manipulating the bits in some uniform, or non-uniform way. All of this takes time, and that is why projects are trying to involve the computer more in understanding its subject, and applying its knowledge in automating the enhancement.

The whole requirement for image processing is centred around the original quality of an image. If it is very poor, much more work will be required to remove noise and other unwanted artefacts before it can be properly analysed. If the original image could be guaranteed to be of optimal quality, the need for image processing would be greatly reduced. The purpose of this project is to identify ways in which this could be done, through the use of currently available file formats. Often, an unsuitable format will render the quality low, a headache for the image processor. If better decisions were made at the design stage, many of these problems could be avoided.

4.1.3 Metafile

Beyond the vector and bitmap formats now available, this type fills a niche which bridges the gap between them. Metafiles have the capability of handling both bitmap and object vector data within the same file. So, a representation of a chair (as in Figs. 4.1 and 4.2) could be stored as both vector and bitmap data in a single file.

The advantage of doing this is that some machines may only be able to display bitmap information, while others can only cope with vector data. Both of these can be catered for with a metafile. Users of both vector- and bitmap-based systems can then display and manipulate a version of the image (assuming the metafile format is supported by both), dramatically increasing the accessibility of the format type. In addition to this greater platform-independence, a reduction in required file storage may be in effect (only one file is needed for bitmap and vector representations of the same scene).

The cost for this flexibility is transferred into the construction and specification of the formats. Metafile formats are generally more complex than bitmap or vector ones as they have to account for many more possibilities in content, encoding, and other storage dilemmas. The knock-on effect could be to decrease the speed of encoding or decoding a complex file which utilises the full range of facilities offered by the format.

However, the fact that metafiles are sometimes stored as ASCII representations brings on some new advantages: the little- and big-endian byte ordering systems of different platforms no longer plays a part, as well as an even larger potential user base.

The manual editing of metafiles is also made possible through the use of a standard text editor, although the implied complexity of such files may warrant this a pointless exercise except for the experienced few.

Finally, the high redundancy in ASCII files makes for high compression ratios. The implication of this is that standard text compression algorithms could be used with a particular format relatively easily, without the requirement for a proprietary compression and decompression library.

A widely known example of a metafile format is that used by Microsoft Windows. Image descriptions are sub-divided into device-independent GDI (Graphics Device Interface) function calls which can later be played back on any compatible device.

4.1.4 Scene Description

Formats of this type are almost indistinguishable from vector types. The only difference is that scene description files describe how to reconstruct the image as a whole, not as individual objects. The methods of applying this are the same as vector and often it can be difficult to classify the two as separate groups.

4.1.5 Animation

When a static image is no longer enough for displaying graphical information, the next step is animation. Initially, such formats stored multiple static images in the same file which, when displayed in quick succession, gave the appearance of movement. Nowadays this has been superseded by more complex and effective methods which can radically reduce the storage requirements of animation. A commonly used example of one such format, MPEG, is discussed in further detail in Chapter 4.4: Advanced Image Formats.

4.2 Bitmap Compression Algorithms

Before looking directly at the various styles of compression used in image formats today, it is necessary to lay some groundwork.

4.2.1 Symmetric And Asymmetric

Image compression and decompression is composed of two distinct types: *symmetric* and *asymmetric*. Symmetric codecs are those which perform the reverse operation of a compression for decompression. That is, in the same way we can reverse a mathematical formulae to extract a different value, we can reverse the procedures used to compress an image leaving an identical copy of the original data.

As the name suggests, asymmetric means the compression algorithm differs from the decompression one. This may mean the compression is more complicated, or the decompression, according to the system in use. The importance of asymmetric compression is that we can make a decision when selecting a type of compression for use in a format exactly what we desire of it. If concerned with obtaining the smallest (or best compressed) file available, we can afford to spend more time on the compression to make sure it is as efficient as possible. Decompressing is likely to be quicker as a great deal of time was spent on compression to ensure it.

A good example is the MPEG animation format (described in more detail in Chapter 4.4.2). Sometimes a Full-Motion Video (FMV) of a feature film can take weeks to compress correctly using multiple high-end workstations and many trained staff providing the system with guidance. Yet it is possible to decompress the data in real-time so it can be displayed with as little disruption as possible on an ordinary home computer. Clearly, the emphasis here is that decompression should be quick, and storage kept to a minimum.

On the other hand, if we wanted to compress many images in a small amount of time, say for real-time object tracking, speed far outweighs the necessity for higher compression Using extensive techniques to maximise compression would mean many frames would have to be skipped for the computer system to keep up. This example is more likely to be symmetric, as decompression should be fairly straightforward in most cases. However, if we do not expect to decompress the target too often, it may be convenient to simplify the decompression to an asymmetric form.

4.2.2 Non-Adaptive, Semi-Adaptive, And Adaptive Encoding

When using a dictionary-based system for encoding and decoding an image, as described later in Chapter 4.2.6, we have three methods by which to identify the dictionary:

- NON-ADAPTIVE: if the compression is to be performed on a specific type of data, we can decide beforehand on a dictionary to use. A good example is compressing English sentences from a novel: we can safely assume words such as ‘and’, ‘will’, and ‘was’ will be present many hundreds of times, and so ensure they are pre-defined in the dictionary. This provides quick compression, but leads to problems if the data changes significantly. For instance, if we translate the novel into Polish and then attempt to compress using the English dictionary, it will not perform nearly as well (if at all!) It also relies on the contents of the novel containing many words which can be referenced, which of course, cannot be guaranteed.
- SEMI-ADAPTIVE: The next progression on from a pre-defined dictionary is to have some idea of the contents which are to be compressed. With this variation, two passes over the data are required. In the first pass, common ‘phrases’ are identified, and the most regular ones are stored in the dictionary. Then, on the final pass the phrases are compressed as with non-adaptive methods. It is still not fully adaptive as the dictionary is pre-defined when the data comes to be compressed, even though the computer has ‘cheated’, in a sense, and taken an advanced look at the contents. It could also mean significant delays when making dual passes of a very large file, decreasing the efficiency of the method.

- ADAPTIVE: The most efficient of the three adapts the dictionary entries as necessary whilst it moves through the data. So, as a new phrase is encountered, it will be entered as an entry. Not only is the best compression attained, in dictionary terms, but only one pass of the data is required.

4.2.3 Lossless V. Lossy

Another consideration we must make when assessing compression techniques is the output from decompression. Is it the same as the original, before being compressed? If it is, then the technique is lossless. That is, the decompressed data is identical to the data that was passed as input to the compressor. If the data has been amended in any way, it is termed lossy.

This method primarily sets out to exploit the restrictions of the human eye, and remove detail which we cannot identify. This provides significantly improved compression ratios at the cost of reduced detail and quality.

How this is done can vary widely, but a crude method may be to remove the least significant part from each pixel colour entry. The effects of this may vary although generally it is seen as an unsuitable lossy approach as even small colour differences may be noticeable. Some advanced procedures have been devised which are capable of removing 'invisible' detail, one of which, JPEG, is discussed in further detail in Chapter 4.4.1.

The idea of using lossy compression is to remove as much unnecessary detail as possible without affecting the naked eye's view of the image. The level of detail which can successfully be removed will vary according to many factors; including image contents, hardware and software quality, and individual perception.

4.2.4 Pixel Packing

Although, strictly speaking, this is not a compression method, it is worth mentioning as it is commonly used to improve the efficiency of storage. The principle behind it is to ensure no storage is wasted which can be used for further data.

As an example, let us assume that we wish to store 4-bit pixel values in a file. There are two approaches we can take:

1. One pixel can be stored in each 8-bit byte. This would mean that 4-bits in every byte would be unused. (See Fig. 4.3).
2. We could double the efficiency of this storage by storing two pixel values in each byte. That is, one pixel in the lower 4-bits, and another in the high 4-bits (as in Fig. 4.4). This prevents valuable space being wasted, and effectively compresses the file by 50%.

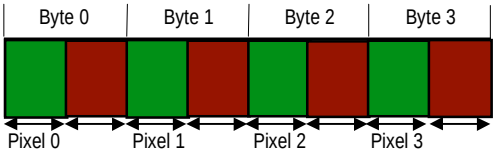

Fig. 4.3: 4-bit Unpacked Pixels

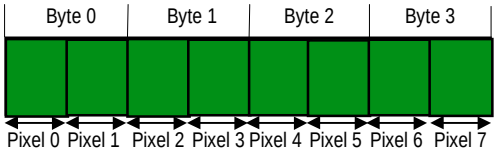

Fig. 4.4: 4-bit Packed Pixels

When it was first devised it was considered necessary in all formats because of the expense of storage. Nowadays, when deciding if this will be included in a format specification, the decision hinges on a trade-off between speed and file size. However, because of the method's simplicity, variations on the theme can be found in many of the popular formats available today.

4.2.5 Run-Length Encoding (RLE)

This, one of the simplest of the techniques in use, can be used to compress any kind of data. However, the compression attained is dependant on the content type. It has become very popular as it is easy to implement and provides a quick method of compressing data.

It works by reducing repeating strings of characters into *runs* of, typically, two bytes (although the atomic RLE base can also be bit- or pixel-based). The first byte represents the number of characters in the run and is called the *run count*. The second byte is the value of the encoded character string and is called the *run value*. Fig. 4.5 demonstrates how strings of repeated characters would be encoded under this scheme.

Uncompressed String:	Run-Length Encoded:
"AAAABCCCCCCCCCDDDDDD" = 20 bytes	4A 1B 9C 6D = 8 bytes (40%)
"MISSISSIPPI" = 11 bytes	1M 1I 2S 1I 2S 1I 2P 1I = 16 bytes (145%)
" WED WE WEE WEB WET" = 19 bytes	1" " 1W 1E 1D 1" " 1W 1E 1" " 1W 2E 1" " 1W 1E 1B 1" " 1W 1E 1T = 36 bytes (189%)

Fig. 4.5: Byte-Level Run-Length Encoding Of Character Strings

Each time the run character changes or the run count exceeds the limit (255 for one byte), a new *RLE Packet* is generated. However, for some types of data, runs are rarely this size. Clearly this type of compression, in terms of images, would be well suited to hand drawn pictures or anything else which contain large areas of the same colours. Real-World scenes would struggle under this technique, and could even result in a larger compressed file than the original (as demonstrated above).

A number of variants of this scheme have been created. Mostly, these involve scanning the data in a different sequence or a different style: such as grids of 4x4 pixels. The exact way this is done is dependant on the format and the type of data being compressed. Two interesting slants on RLE have formulated.

The first is a lossy method which involves removing some of the least significant bits from each byte value. This would be of use in real-world scenes which have many subtle differences in pixel colour which could be exploited with little affect on the decompressed output.

Another point to consider is whether or not scan lines are treated separately. If they are, a run terminates at the end of a scan line regardless of what the first character on the next line is. If this is not the case, *cross-coding* is in effect. Without special attention, problems could arise on decompression when identifying scan line boundaries. Of course this can be avoided, and the result would be a slightly more efficient compression ratio, but generally the extra calculation required on decompression adds to the overall time required.

Encoding scan lines individually has advantages when an application needs to use only part of an image. Knowing where each scan line begins and ends means we can easily display, say, lines 100-120. This would require significantly more work if the image had been cross-coded.

This is where the second of the variations steps in. It provides an extension to this which allows whole scan lines to be encoded in the same manner as bytes (or bits or pixels).

4.2.6 Lempel-Ziv Welch (LZW)

This lossless method for compressing data can be found in several of the popular image file formats, including GIF, TIFF and PostScript Level 2. It is based around substitution, or dictionary-based LZ77 and LZ78 algorithms devised by Abraham Lempel and Jakob Ziv in 1977-78. This was extended in 1984 by Terry Welch, thus Lempel-Ziv Welch was born.

It provides a fast symmetric way of compressing and decompressing any type of data without the need for floating point operations. Another reason for its popularity is that it works as well on little- and big-endian machines because information is written as bytes and not words (although bit-order and fill-order problems could still be encountered).

The system works on an input stream of data and builds up a list of dictionary entries (a translation table) of 'phrases' which appear in the stream. When a sub-string from the stream is identified as already being in the dictionary (that is, it has already occurred previously) it is replaced in the compressed output by a reference to the dictionary entry. If it is not present, a new entry is placed in the table, and the reference sent to the compressed stream. These references are generally smaller in size than the uncompressed phrases and thus compression is attained.

To decompress, a compressed stream is read and references are added to a dictionary if not present. The phrases can then be restored building up the dictionary as it progresses. The advantage of this procedure is that it is not necessary to store the table within the compressed output as it is built up as required when decompression takes place. When initialising the dictionary before compression, the first 256 entries are set to values 00_{16} through FF_{16} (all possible byte values) from which all sub-strings can be built. As both the encoder and decoder are aware of this, there is no need to keep the dictionary stored with the data.

Variants on the way data is passed in to the compressor are based on whether the data is in byte or pixel values. For example, TIFF packs pixel data into bytes before compression depending on the image's bit depth and number of colours. So, a byte could represent a pixel, less than a pixel, or more than one pixel. With the GIF format, each input symbol must be a pixel value. As 1- to 8-bit depths are supported, there are between 2 and 256 input symbols to initialise. It is irrelevant with GIF how the pixels may have been packed originally as they are treated as a sequence of symbols regardless.

With a set of data that has odd-size pixels, packing into bytes will obscure patterns making the compression less efficient. If they agree, such as two 4-bit pixels per byte or one 16-bit pixel every two bytes, then the byte packing will work well with LZW. On the other hand, odd-size bit depths work well in the GIF approach but make it unwieldy having greater depths than 8-bit (as the dictionary initialisation will be much larger: 65,535 entries for 16-bit; 16,777,215 for 24-bit).

Fig. 4.6 shows how a simple string of text characters would be compressed by LZW. The first 256 entries in the table (starting from zero) are initialised to the possible single-byte values (ASCII character codes are used for alphabet clarity). On the first pass, a check is performed to see if the string " W" is in the table. Since it is not, the code for " " is output, and the string " W" is added to the table (entry 256). After the third character, "E", has been read in, the second string code, "WE", is added to the table and the code for letter "W" is output. In the second word, the characters " " and "W" are read in, matching sub-string 256. So, code 256 is output, and a three-character string is added to the table: " WE". This process continues until the string is exhausted and all codes have been output. With a 9-bit code for each (the minimum required) the 19 character input (or 19 bytes) could be packed into 108-bits, or 13.5 bytes: 71% of the original size (assuming a pack bits algorithm was employed as described in 4.2.4).

Input String: " WED WE WEE WEB WET"

Characters Input	Code Output	Translation Table Values
" W"	<32> [' ']	<256> = " W"
"E"	<87> ['W']	<257> = "WE"
"D"	<69> ['E']	<258> = "ED"
" "	<68> ['D']	<259> = "D "
"WE"	<256> [" W"]	<260> = " WE"
" "	<69> ['E']	<261> = "E "
"WEE"	<260> [" WE"]	<262> = " WEE"
" W"	<261> ["E "]	<263> = "E W"
"EB"	<257> ["WE"]	<264> = "WEB"
" "	<66> ['B']	<265> = "B "
"WET"	<260> [" WE"]	<266> = " WET"
<EOF>	<84> ['T']	

Fig. 4.6: Lempel-Ziv Welch Compression Of A Text String

On decompression, as shown in Fig. 4.7 below, the dictionary does not need to be kept after the compression stage, as the decompressor is able to calculate the table values from the code stream. The first four output values can be looked up from the initialised dictionary (which should be identical to that used by the compressor). When tag 256 is encountered, the entry has been created as " W" in the table (the first sub-string not found in the dictionary). This continues in an identical manner as seen in Fig. 4.6 until the original stream is restored.

Input Codes: "<32><87><69><68><256><69><260><261><257><66><260><84>"

Code Input	Last Code	Stream Output	Character	Translation Table Values
<32>		<32> [' ']		
<87>	<32>	<87> ['W']	'W'	<256> = " W"
<69>	<87>	<69> ['E']	'E'	<257> = "WE"
<68>	<69>	<68> ['D']	'D'	<258> = "ED"
<256>	<68>	<256> [" W"]	' '	<259> = "D "
<69>	<256>	<69> ['E']	'E'	<260> = " WE"
<260>	<69>	<260> [" WE"]	' '	<261> = "E "
<261>	<260>	<261> ["E "]	'E'	<262> = " WEE"
<257>	<261>	<257> ["WE"]	'W'	<263> = "E W"
<66>	<257>	<66> ['B']	'B'	<264> = "WEB"
<260>	<66>	<260> [" WE"]	' '	<265> = "B "
<84>	<260>	<84> ['T']	'T'	<266> = " WET"

Fig. 4.7: Lempel-Ziv Welch Decompression Of A Code Stream

4.2.7 Huffmann Coding

This algorithm produces variable-length codes according to a symbol's probability within a stream. These codes can then replace the symbols in the compressed stream, thus producing compression. The two important things to note about this are:

- Shorter bit-codes represent the symbols most likely to occur. Thus a 1-bit code can be assigned to the most probable symbol in the stream and the largest amount of bits per symbol represent the least likely.
- The codes have a *unique prefix attribute*, which allows variable-length codes to be identified and decoded even though they are not uniform.

To illustrate this point, consider how the ASCII data string "BBAAABCBDEEBBE-BBBAEBDAEDBEBEBBDAB" would be compressed. The first step we would undertake is to merge the frequency values of the symbols into parent nodes until only two parents remain. Refer to Fig. 4.8 for the example. On the first pass, we take the lowest two frequencies, 1 and 4 (C and D) and merge them into one parent node equal to their sum (5) and temporarily pretend C and D no longer exist. This is repeated at the next level (with values 5 and 6) and continued until only two parents are left (15 and 18). The underlined values signify those which have been designated as parents.

On the second pass we work in reverse building up the bit-codes for the values, which can now be more efficiently assigned according to their probability. The highest value of the two root parents is assigned bit 0 (to value 18), the lower assigned 1 (value 15). The next level back value 18 is broken down into 11 and 7. So, the bit-codes for these are an extension of the parent codes. In this instance 11 is given 00 and 7 is given 01. Value 15 remains unchanged as it is already in its lowest denomination. This procedure continues until all the frequency probabilities have been assigned a bit-code. Note that the highest frequency values have the smallest number of bits. The mapping table would also need storing but the compression attained is still significant. Fig. 4.9 yields the results of these operations.

Symbol	Frequency
A	6
B	15
C	1
D	4
E	7

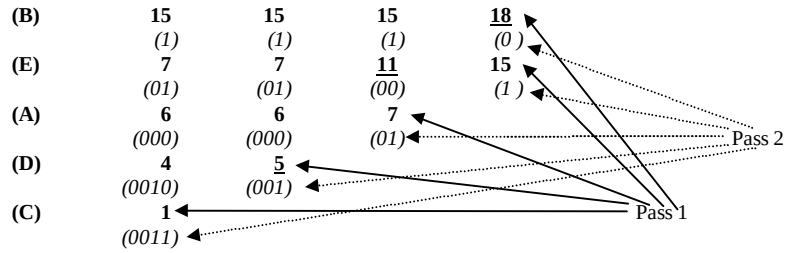

Fig. 4.8: Huffmann Coding – Symbol Frequency And Bit-Code Representation

Symbol	Frequency	Bit-Code	Total bits
A	6	000	18
B	15	1	15
C	1	0011	4
D	4	0010	16
E	7	01	14
	33 (bytes)		67 (bits)

$$\left(\frac{67(bits)}{33*8(bits)}\right)*100 = 25.38\% \text{ (of original size)}$$

Fig. 4.9: Huffmann Coding – Results

This example does not demonstrate the application of the unique prefix attribute to simplify the concept, although minor changes to the algorithm could ensure each bit-code has a unique prefix. In this example, the bit codes would be (B⇒0; E⇒100; A⇒101; D⇒110; C⇒111) as there are five unique codes required and three bits for four of the values is the best that can be offered. The impact this has on the compression is not significant in this case (69 bits ⇒ 26.00%) although it could be more of a problem with larger data sets. As with all forms of compression, speed and size balance the scales.

When certain symbol probabilities are being converted into bit codes by a Huffmann encoder, redundancy is taking place. For example, to encode a symbol with a 1/3 chance of appearing, Huffmann Coding will most likely assign 2-bits even though the optimal storage required would be 1.33-bits. Although this does not seem significant with this example, consider an image which contains a symbol with a very high probability of appearing, say 90%. Optimally, 0.15-bits could handle this even though 1-bit will be assigned. In effect, more than 6 times the storage required is being used due to the inefficiency of the bit-code assignment.

4.2.8 Arithmetic Coding

Only within the last ten years has a new method come to light which bypasses this problem with the Huffmann algorithm. Arithmetic Coding has the capability of optimal storage for any string through the use of arithmetic. More specifically, a string is represented as a floating point number between 0 and 1. This number can be uniquely decoded to give the stream of symbols that it is constructed from.

To do this we must first calculate the probability of each of the symbols contained in the stream. With these values we need to assign a range within 0-1 for each of the symbols according to their likelihood. For example, if we wish to store the phrase 'BILL GATES' the distribution would be as in Fig. 4.10. Note that for ' ' the actual range available is 0.0-0.99 (as 0.1 is the beginning of the range for 'A'). This applies to all symbol ranges similarly. The ordering of the symbols and the method for assigning ranges is not fixed, but the encoder and decoder must use the same set of rules. Using this table we can then begin creating our floating point number which will store the phrase.

Symbol	Probability	Range
' '	1	0.0-0.1
'A'	1	0.1-0.2
'B'	1	0.2-0.3
'E'	1	0.3-0.4
'G'	1	0.4-0.5
'I'	1	0.5-0.6
'L'	2	0.6-0.8
'S'	1	0.8-0.9
'T'	1	0.9-1.0

Fig. 4.10: Arithmetic Coding Probability Distribution

At each step we will proportionately sub-divide the ranges of the high/low values so that the next character is within the correct sub-range. The full calculation is given in Fig. 4.11. Before the first symbol, we know the range will be from 0.0-1.0. A 'B' will restrict the range to 0.2-0.3. 'I' will further restrict this to 0.25-0.26. We continue adding precision to the number using the low value (as further symbols will add greater precision and increase the value of the number within its available range) until all the symbols are accounted for. This should leave the value 0.2572167752 which is the encoded version of our string 'BILL GATES'.

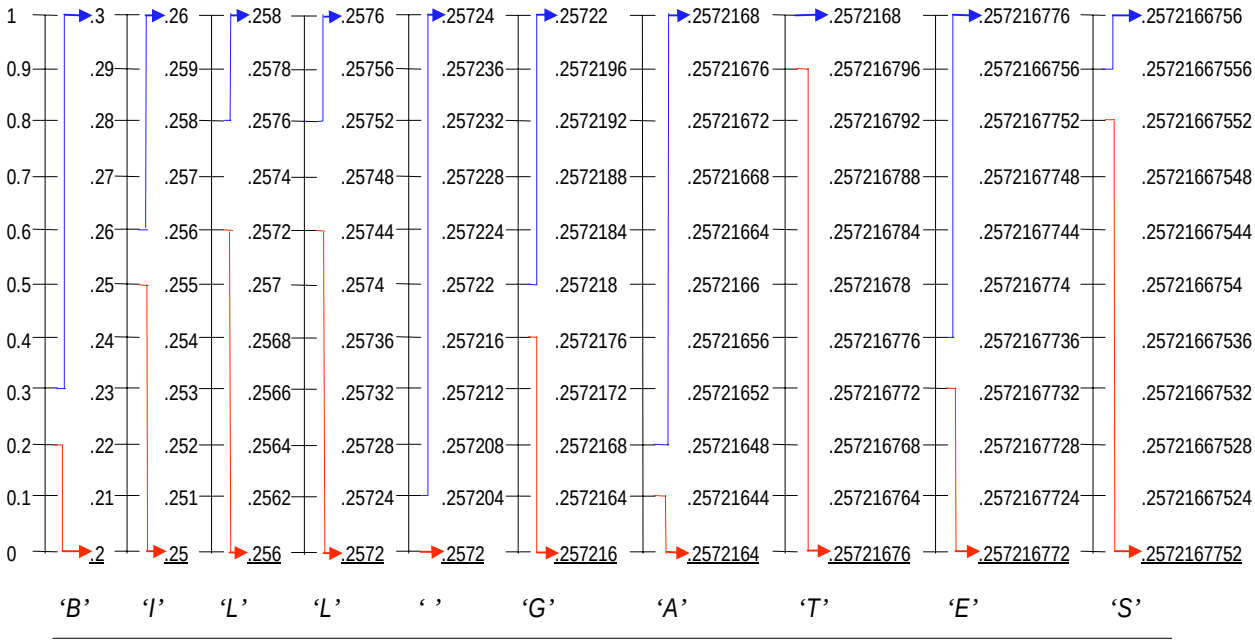

Fig. 4.11: Arithmetic Encoding Of A String

When the decompressor wishes to obtain the encoded message, it looks at the most significant digit of the value, in this case 0.2, and then knows the whole value lies between 0.2 and 0.3. Looking up this in its string table it will know the first character is 'B'. It continues this by narrowing the range, identifying characters along the way in the reverse manner of the compressor.

4.3 Colour Spaces And Other Considerations

Beyond the compression techniques available, other topics must be examined before we can fully understand the nuances of file formats. While these are usually specific to a particular format, the general principles remain the same.

4.3.1 Colour Space

The methods possible for storing colour information are many. In the same way that graphic file formats have flourished, so have the colour space schemes. Each one has been designed to model colour accurately with the intention of providing output as close to the original as possible. Of course, due to the small differences between monitor phosphor colouring and other conditions, this is unlikely to be achievable with total success. While some colour spaces have been developed scientifically to correlate closely to the human vision system, others were created to ease computer colour modelling.

Nowadays a small subset of these are commonly used, mainly due to their ease of use and fair results. In particular I refer to the RGB and HLS models (see Figs. 4.12 and 4.13).

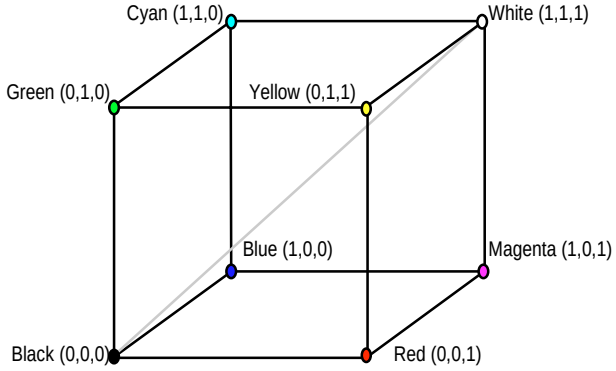

Fig. 4.12: The RGB Cube

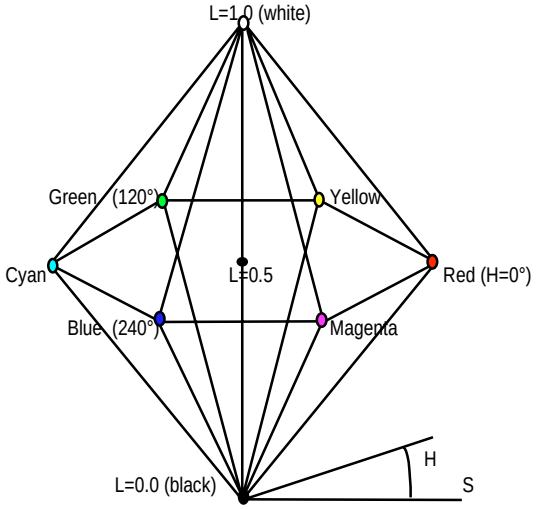

Fig. 4.13: The HLS Double Hexcone

RGB allows the *red*, *green* and *blue* components of each pixel to be mapped onto an intensity of the monitor's cathode-ray tube for each of the three electrode guns. It is then possible to calibrate the monitor to attain accurate output (although this can be difficult). The second advantage of this system is that it is conceivable to visualise the colour resulting from the three values. For example, an RGB value of (200, 0, 0) has a high red factor (assuming a scale of 0-255 for each) whilst green and blue components are not present: naturally one would assume the resultant colour to be a shade of red.

The HLS scheme models colour according to its *hue*, *lightness* and *saturation* values. Hue is described as being 'similar to one, or to proportions of two of the perceived colours red, yellow, green and blue'. Lightness is the '…brightness of an area judged relative to an apparently equally illuminated white or highly transmitting area'. Saturation is the 'colourfulness of an area judged in relation to its brightness' (Jackson, MacDonald et al., 1994). As can be seen from these definitions, visualising the output from a typical set of HLS values is not as easy as with RGB, but due to its ease of programming implementation it has also remained at the forefront of commonly used colour spaces.

This model is perceived as being more accurate than RGB at modelling the real-world. To illustrate this point, if we assume that lightness (L) is modelled on a scale of 0-1, with a mid-point at 0.5, all colours are possible down the 0.5 lightness with whiteness tapering towards 1 and black to 0. As a perfect white will reflect all the illumination falling on it (L=1), any area exhibiting a colour must have a lightness of less than 1; as modelled in this system.

4.3.2 Other Considerations

Some formats have added to the standard features of image formats by allowing extra information to be stored. Some of these features include:

- Support for multiple compression algorithms. This could be to enhance the compression attained, to provide greater compatibility with other formats, or to improve the suitability of the format for certain types of images, such as binary (black and white only). An example might be an image which was first run-length encoded and then Huffmann encoded to allocate more efficient storage for the run-length symbols.
- Multiple colour spaces can be included to provide compatibility with other formats or output devices. If this is built in to the format, it is not necessary for the application programmer to be concerned with it, she can merely use the provided tools. Many of today's formats provide support for more than one of these, due to the ease of implementation.
- Transparency of pixels allows the overlaying of an image on another in such a way that we can see through the top image to the back, as if it were transparent. This is closely tied to the colour space used, as the combination of semi-transparent pixel colours will be determined by the method for representing colour. An example is GIF89a which uses the RGB additive colour system in correlation with Alpha channels (which store the transparency at each pixel).
- To enable the storage of multiple images in each file. Images which are related can be stored together in one file forming an image library. The major advantage of this is the reduction in overheads of storing each image in a separate file. Good examples are the TIFF and GIF89a formats which provide features for this.
- A further refinement of multiple image formats is animation. Primitive forms may just involve displaying many images every second to give the impression of animation. More sophisticated methods have been designed, however, which reduce the storage required whilst still allowing the real-time display at up to VHS quality. A more detailed description of one of these, MPEG, can be found in Chapter 4.4.3.

4.4 Advanced Image Formats

In this chapter I will discuss three of the more complex and advanced methods by which images can be compressed and stored: JPEG, MPEG and Fractal. They are described in overview only as a full study of them is beyond the scope of this project.

4.4.1 JPEG

One of the problems with lossy compression is knowing what information can be safely removed from an image without affecting the overall picture to the naked eye. Crude methods involving the removal of precision of pixel values does not suffice as its results will vary according to the image contents. This is where JPEG comes into play. The Joint Photographic Experts Group wanted a way to maximise compression without affecting the output, and thus the JPEG JFIF (file interchange) format was born. This format supports both lossless and lossy compression, but we are only interested in the lossy component here.

Three stages are undertaken to perform this compression, as shown in Fig. 4.14.

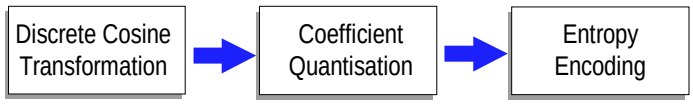

Fig. 4.14: The Three Stages of JPEG Lossy Compression

i. *Discrete Cosine Transformation.* The first stage of the JPEG compression procedure involves a mathematical concept known as the DCT. This takes as input a block of pixel values (8x8 with JPEG) and converts its spatial representation into a frequency range. This is performed as a 3-dimensional operation, as the *x* and *y* axes represent the pixel location, and *z* represents the pixel intensity at location *x*, *y*. The result of this operation is a set of values which contain the same data only in terms of frequency and magnitude. The reason this is so important over spatial representation is that the low frequencies (as more commonly displayed on-screen) can be distinguished from the less important high frequencies. This would not be possible with the spatial representation, as there is no concrete way of deciding which pixels are more important than others. The top left corner value of the block is the lowest frequency value, and is the most valuable. As we move further from this point the values become less important to the image.

   However, it is important to note that at this stage, no loss of information has taken place (we have just altered the way we represent the data). In fact, the output of this step requires more storage than the input (the frequency value range is $-1024 \leq x \leq 1023$ requiring 11-bits per pixel). The 'lossy' aspect takes place in the next step.

ii. *Quantisation.* Now that we can differentiate between important and less important frequencies, we are able to discriminate against those below a certain threshold. The higher the threshold, the more lowest frequencies are discarded (or converted to 0). This is how the quality factor of JPEG functions so it can be amended according to the quality required in the output.

iii. *Entropy Encoding.* Three sub-stages complete the JPEG encoding process. Firstly, the coefficients in the top left of each block are converted from absolute to relative values. As the differences from block to block is likely to be small, using relative values allows smaller values to be stored.

   Next, the zero and non-zero values are treated separately. The zeroed values can be converted to run-length encoding pairs reducing the storage requirements drastically. Due to the nature of the frequency distribution (most important, or low, in the top left corner) the run-length encoding does not take the usual path across and down an image. Instead, a zig-zag sequence is used which exploits the features of the frequency coefficient distribution. Fig. 4.15 demonstrates such a sequence which gives a higher probability of efficient run-length encoding.

   Non-zero values are Huffmann or Arithmetic encoded in the manner described earlier.

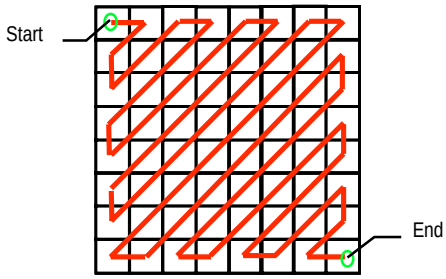

Fig. 4.15: Zig-Zag Sequence

The time taken to compress an image with the JPEG process is significantly longer than with the more simple methods discussed in the previous sub-section. The added complexity provides the capability for impressive compression ratios at the expense of image detail. The advantage of this is that JPEG stored images can be compressed according to the requirement for storage space against quality. If an application requires images to outline objects only, the detail required is low, meaning a low quality factor can be used. In this example the ratios possible can be anything from 30:1 upwards. On the other hand, high quality images can be attained by increasing the value. Although the stored image has lost some of its detail (even at maximum quality) to the naked eye the differences are invisible. Both of these extremes can be catered for by the one format. As JPEG is symmetric, the time to decompress is roughly similar to compression (depending on the quality factor).

On the downside, the DCT block size of 8x8 can cause disparities on the borders of the blocks. This is because the processing for each block may average the values to significant differences which generate a blocky effect. Admittedly this is rarely a problem on high quality factor images, but as quality is decreased so the blocks will become more evident. Fig. 4.16 demonstrates the blocky effect caused by quantisation. Image (a) has been stored using low compression, does not show any blockiness and is 6,357 bytes in size. Image (b) uses 10 times more compression and the entire image is visibly blocky (8x8 pixel blocks). Its file size is 2,093 bytes.

(a) (b)

Fig. 4.16: The Effect Of Quantisation

One way around this problem could be to amend the size of the block to, say 64x64. However, research shows that connections between pixels tend to diminish quickly, such that pixels even fifteen or twenty positions away are of very little use as predictors (Nelson, 1992). Furthermore, the processing and memory power required to work with blocks this size make it unattractive in terms of time.

Similarly, if we made the blocks smaller many more iterations of the procedure would be required, again drastically increasing the time required for compression.

When this format was first devised, circa 1991, it was decided that it should be possible to use JPEG compression/decompression on a wide range of the available computer systems, hence the block size of 8x8.

4.4.2 MPEG

All the compression methods described thus far have dealt with static images only. MPEG is one of the methods for handling full-motion video as created by the ISO. Originally, it was merged with the JPEG team and this explains the similarities between the two.

MPEG-1, the first standard, is based around the concept of three frame types:

- *Intra-frames*. These are the closest relatives of JPEG images, as they represent one frame in a video stream (the entire frame is stored). Compression is based on the DCT method.
- *Predictive frames*. In contrast, inter-frames store only the differences in the current frame with reference to the closest preceding *I* (intra) or *P* (predictive) frame. This delta information (the difference values) can then be DCT encoded.
- *Bi-directional frames*. These frames are constructed from the nearest two *I* or *P* frames, although it must be between the two references (that is, *IBI* or *IBP* or *PBI* or *PBP*). Theoretically, there can be a limitless number of *B* frames between two reference frames, although in practice there are typically twelve *B* and *P* frames between each *I* frame, or an *I* frame each 0.5 seconds (assuming 25 frames per second) (Murray and vanRyper, 1994). Again, the data is DCT compressed.

On decompression, it is necessary, before decoding a *B* or *P* frame, to have any necessary reference frames in memory. Delta information is useless without data to compare it against. For this reason, a stream which is encoded as *IBBPBBPBBP* (0123456789) would be decoded in the order *IPBBPBBPBB* (0312645978) to ensure the necessary references can be accessed as required.

The characteristics of full-motion video suggest it is likely that the scene in the stream will change significantly fairly regularly. When deciding where to place *I* frames, a balance must be made to decide whether or not a *B* or *P* frame might provide better performance (in terms of compression). If we were to place a *B* frame at the beginning of a new scene, the differences from that to the reference *I* frame would be larger than storing a new *I* frame, hence less compression and slower decompression (as the reference *I* and *P* frames would need to be in memory for the *B* frame data to be quantified).

The complexity involved in the compression process of a video stream in this manner is very extensive, and compression for a typical feature film encoding can take days or weeks to complete. Naturally, this makes obvious the point that MPEG is asymmetric. Nowadays it is possible to view a VHS quality MPEG encoded video in real-time. It is also important to note that trained users usually decide where the varying frame types are to be represented, to ensure optimum playback quality and efficient compression concurrently. Furthermore, we have not even considered the necessities of encoding the audio and synchronising it with the video stream, adding further complications.

Compression ratios attainable with MPEG depend on the quality factor used at the quantisation stage. As with JPEG the range is wide, depending on the quality required for the application and the time spent on choosing the correct use of frame types. Typically, if the application involves VHS quality video playback, the ratio will be in the 16-40:1 range. This means it is possible to store 50-60 minutes of video onto a 780 Mb CD-ROM. The Philips VideoCD range is a good example of MPEG being put to use.

Currently, work is under way to extend this standard to incorporate new advanced features which will improve the overall performance of video playback through MPEG whilst still being backwardly compatible. This new standard, MPEG-2, removes the 1.5 Mbps data throughput imposed by the MPEG-1 standard, allowing the future use of MPEG for more than just CD-ROM and home computers. Plans are under way to incorporate it into the next generation of cable, satellite and home entertainment systems with features such as higher data throughput, improved picture quality, support for multiple display types (examples being PAL/NTSC/SECAM) as well as interlacing and backward MPEG-1 support.

4.4.3 Fractal

In 1977, Benoit Mandelbrot published a book, The Fractal Geometry Of Nature, which put forward the hypothesis that traditional geometry with its straight lines and smooth surfaces does not resemble the geometry of trees and clouds and mountains. Fractal geometry, with its convoluted coastlines and detail ad infinitum, does (Kominek, 1995).

By 1981 a unity within the diverse world of fractals was presented by John Hutchinson: Iterated Function Theory. Later work by Michael Barnsley reinforced this by providing the mathematics required to demonstrate what an iterated function would look like for a given image (the Collage Theorem). This highlighted the possibility that if fractal geometry was well-suited for generating natural-looking images, could it not be reverse-engineered to compress images? The *inverse problem* has never been solved, and although Barnsley believed he had cracked it, he later admitted it took 100 hours guided by a human to compress a typical image, and 30-minutes for decompression.

It was one of his students, Arnaud Jacquin, who produced a workable method of fractal compression, known as Partitioned Iterated Function Systems. The method involves breaking the input image into domains and ranges from which an affine transformation matrix, (involving combinations of scaling, rotation, shearing and translation), can be applied to link range blocks to a larger domain block through the application of the affine transformation on the range block. The result of this produces the equation: range block = domain block + affine transformation.

To illustrate this concept, refer to Fig. 4.17 which shows how a range block can be represented as a domain block plus a transformation in a simple image.

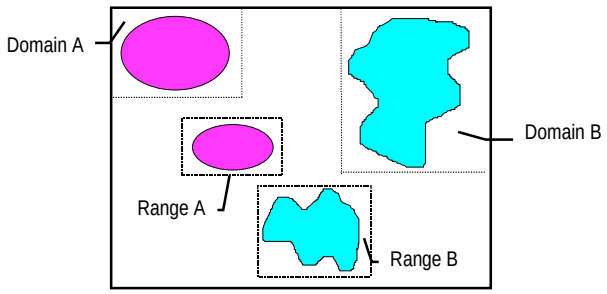

Fig. 4.17: Domain And Range Blocks In Fractal PIFS

Range A can be represented as Domain A + Transformation (Scale 0.5). Range B can be Domain B + Transformation (Rotation 270º). This means that whole images can be approximated by selecting domain blocks and the necessary transformations required to provide the finer details. This gives a typical compression ratio in the range 4-100:1, which although a far cry from the claims of 10,000:1 compression by Barnsley, was still a step forward.

As an example of how we calculate the compression possible, consider a 256x256x8 (bits-per-pixel) image which is divided into regular 8x8 partition range blocks. If we assume, for simplicity, that we will only need one affine transformation for each, we have 1024 transformations to store. In most implementations, the domain blocks are twice the size of the range blocks, so the spatial contraction is constant and can be hard-coded into the decompressor. This means we require:

x position of domain block	8
y position of domain block	8
luminance scaling	8
luminance offset	8
symmetry indicator	3
	35 bits

Or $\frac{8*8*8}{35} = 14.63\%$ (8*8*8 = *x* position, *y* position, luminance scaling).

Example taken from Kominek, 1995. This is not impressive, especially when compared to JPEG. More advanced systems are also in use which can improve this (for the same image) to 35-40% and involve the use, among others, of entropy encoding as described previously in 4.4.3(iii). As a general rule, through experimentation and general experience in industry, it has been found that for low compression (<35-40%), the fractal method works best, and for higher than this JPEG is better suited.

The major drawback to this system is the time required to perform the procedure. The decisions made as to which areas are defined as domain and range blocks take a lot of searching time, and provide no guarantee of the resulting compression ratio. This also does not account for the fact that the image contents will rarely contain uniform objects which can easily be mapped to rectangular blocks. If the algorithm were very thorough, a more efficient set of blocks may be gathered producing a more efficient ratio. Ultimately, if an infinite time were allowed to calculate the blocks and transformations we could attain perfect fractal compression (we have solved the inverse problem for the image). On the other side of the coin, the less time spent on this, the poorer the representation and compression. The balance hinges on the requirements of the system and resources available.

On the upside, if we zoom in to the image, the blockiness of enlarged pixels is not present, as with all the other image description methods. As the image is represented as formulae, zooming in generates detail from the formulae. No matter how close we look at a portion, blockiness will not arise, and it will give the impression that we really can see the smallest of detail. Of course, the added detail we see is only generated from provided information, so, for example, zooming in on an area of skin would not give us an image of skin pores or hair roots unless the detail was contained in the original. Nevertheless, this feature can be very useful. Fig. 4.18 shows a fractally compressed image of a touring car, before and after zooming. As can be seen, the zoomed image appears the same 'quality' as the whole image. Ironically, these images have been converted to a more conventional bitmap format, Windows BMP, so they can be included in this report.

Fig. 4.18: Fractally Compressed Image Before And After Zooming

5. Measuring Image Quality

In the same way as in any mathematical problem, measuring image quality involves a combination of many factors which affect the output quality of an image. Exactly how they co-exist will make a large difference to the final results. For example, certain compression algorithms (such as Run-Length encoding) may not be suitable for true colour data, whilst others thrive on it. Quite apart from the image quality, there is also the question of speed and efficiency. Although one could assume 'quality' refers solely to the output resolution and colour depth I feel it goes deeper than that, incorporating the suitability for its application in all areas. We do not wish to measure the quality of a bitmap on-screen and disregard the fact that it took thirty minutes to compress unless time is not a critical factor.

The aim of this chapter is to address the issues over measuring an image's quality, in all respects. Starting off with a description of the factors I feel affect the quality of bitmaps, I shall move on to some suggestions as to how popular image formats can be measured for a particular application and rated accordingly. It is important to note at this point that the information contained in this chapter is based on my own opinions.

5.1 Factors Affecting Image Quality

Such factors can broadly be categorised under two headings: the more tangible image format factors, such as colour depth, resolution and so on; and the higher level factors, such as speed, system resource requirements and so on.

5.1.1 Image Format Factors

With any image format there are features and restrictions. The author of a new format must balance off the advantages and disadvantages of every perceivable option and make the best judgement that will leave an all-round suitable format for the application in-hand. The main features available in any format can be summarised as:

- *Pixel Resolution*. In the early days of image formats, the supported maximum pixel resolution was decided more by the available hardware than designer decisions. Nowadays, possible resolutions can be very large. ZSoft PCX, for example, has a maximum resolution of 65,535x65,535 pixels, a great deal more than is required by the average user. This is generally agreed in the computing community to be one of the most important factors. An image of high resolution but few colours is more discernible than one of low resolution and high (or true) colour.
- *Colour Depth*. Older formats allow for 1-bit black and white images only, and were generally designed for fax use only where it was well-suited. Other possible depths can be greyscale (usually palette-based), palette indexed colour with 4, 8, 16, or 256 entries, high-colour of 15- or 16-bit colour values for each pixel (32,767 or 65,535 colours), 24-bit true-colour (over sixteen million colours), 32-bit (true-colour with 8-bits of alpha-channel transparency per pixel) and now in excess of 128-bits per pixel! Many of the modern formats support up to 32-bit, while some, such as GIF89, include a hybrid which in this case is a 256-entry colour palette with alpha transparency. In order to achieve photo-realism in images, it is essential for a format to support at least the mid-range of this scale. Typically, the high-end values are used in professional systems only.

- *Compression and Decompression*. Some of the major codecs available in popular image formats have been described in Chapter 4. The prime concern with compression is whether it is lossy or lossless. While lossless schemes ensure the image is preserved, they cannot attain the kind of compression ratios provided by lossy methods. Large true-colour images can be shrunk to sizes smaller than many lossless compressed lower resolution and colour-depth images, whilst looking identical to the average naked eye. On the other hand, this process can take significantly longer and may require more resources, especially on larger images. Codecs may be suited to a particular kind of data. A good example of this is Run-Length encoding which does not cope well with true-colour data stored as Red, Green and Blue byte values. These values will rarely be identical unless it is a 24-bit greyscale image, and as the average human cannot distinguish between this amount of grey levels the effort would be wasted. For this reason, formats such as BMP, which have optional compression, do not generally compress 24-bit data for fear of generating a larger file than the original, the antithesis of compression.
- *Colour Space*. There are many different ways colour can be modelled in computer generated images, of which the two most common, RGB and HLS, have been described in Chapter 4.3.1. Many of these were designed for a specific type of image content. For example, artists might choose one colour model as they feel it best synthesises light and the subtleties of water colour ink (and some have been designed for this purpose). The RGB and HLS models are satisfactory for most users needs, so choosing a colour-space may not be an issue. However, some applications require a more accurate model which will be able to cope with the task at hand. Very often the need to change this is due to colour mapping from one system to the image format, where using a compatible colour system makes faster collation and storage of data possible, whilst ensuring accuracy.
- *Multiple Image Storage*. This, less obvious feature is important in situations where a library of related images should be kept together. Storing them as separate files is likely to increase redundancy in the data. For example, a colour palette could be used for many images in the same file. Furthermore, the cataloguing of the images is practically complete (and is often built into the format), and there is less likelihood of the related images being separated.

5.1.2 Higher Level Factors

Quite apart from the image format itself, other, less tangible factors influence the quality of an image. This is subjective as it is entirely dependent on the application:

- *Speed*. While some formats provide great speed at representing and storing images, others require more time for processing. Assuming the code has been written efficiently, reasons for slow image format storage and retrieval are usually directly linked to the compression algorithm in use. Uncompressed data has the advantage of low processing overheads for fast storage and retrieval. As the codec becomes more complex, so the time to perform the operations grow. The speed of the system used can also vary this, as an image compressed with the same algorithm should generally decompress quicker on a faster machine. While ten seconds may be an acceptable time to compress the average user's image needs, with an application such as real-time object tracking and analysis, a ten second lag between each frame could be disastrous. Another way to look at it is to compare compression and decompression times where the format is asymmetrical, so it can be viewed in real-time, as with full-motion video formats.

- *File Size*. Inversely proportional to speed is the resulting file size on output. An uncompressed file is likely to be large, especially if it is in 24-bit colour. As the compression becomes more efficient, so the file size will decrease. The extreme end of this scale could be MPEG encoding which can take weeks to compress a full-motion feature film but provides optimal compression. Applications requiring fast storage could provide little or no compression to start with, and then convert the format of the saved images at a later time to make better use of the storage space.
- *System Resources*. The kind of system running the software will play a significant rôle in the overall performance and quality of a chosen format. Attempting to compress an MPEG full motion feature film using an Intel 80286 with 512Kb of RAM and a 20Mb hard disk is not sensible. If the chosen format is to be sensible, consideration must be made on the system resources that will be available to perform the operation. Features like floating-point maths can make a big improvement in the quality of an image, in terms of speed and visual quality, but if the system used cannot perform it, the format will be useless at best.
- *Application*. The most important decisions to be made when choosing suitable image formats relate to whether or not they can cope with the application they will be used for. Applications that will generate mass storage of image files, say a graphics database, would benefit from any format that can provide good compression from a typical image. Conversely, applications such as motion tracking have bigger problems to be concerned about: like can the format generate and store twenty-five static frames per second at high resolution? In a case like this, storage would probably have to be sacrificed to maintain such a rate (although some top of the range machines can do this with reasonable compression as well). Each image format can be regarded as good for an application, providing it is the right application.

5.2 Suggestions On Measuring Image Quality

5.2.1 Exhaustive Testing

For any application, the combination of factors, described above, which identify the best image format to use will be different. Unfortunately, I have not been able to locate a book which describes the formats best suited for certain applications. Although such a book would be useful, in my opinion, being this specific is not necessary to find the right match.

Each application has a set of requirements for an image format, describing the colour depth, resolution, colour type, speed of compression and decompression, and so on for all the factors already discussed. What is required is a benchmark system which can be applied to the popular formats, typifying a wide range of application needs.

One method for doing this would involve first generating a set of test images which could represent a wide range of application images, as in Fig. 5.1.

(a) (b)

Fig. 5.1: Test Images For Measuring Image Quality

Image (a) is a 4-bit (only 8 out of 16 colours used) 240x160 pixel test pattern used to take advantage of Run-Length encoding algorithms. Image (b) is a 24-bit Kodak Photo-CD picture, 1478x966 pixels in size, used to take advantage of more complex compression schemes and its large size. I have saved copies of each of these images in BMP (compressed and uncompressed), GIF, JPEG (low quality and maximum quality) and PCX. For simplicity, the images in Fig. 5.1 have been scaled to fit the page. The results of this test can be seen in Fig. 5.2. As an aside, Adobe Photoshop provides four levels of JPEG compression only: maximum, high, medium and low. While a direct comparison of these values to a 0-100 scaling is not easy, the lowest setting still maintains a reasonable quality and is consistent with a level in excess of 30 on such a scale.

Image	*Format*	*File Size (bytes)*	*Compression (%)*		
a	BMP	19,320	*benchmark*		
	BMP (+RLE)	760	3.93		
	GIF	947	4.90		
	JPEG (max.)	5,550	28.73		
	JPEG (low)	3,671	19.00		
	PCX	2,177	11.27	*Load (secs)*	*Save (secs)*
b	BMP	4,285,232	*benchmark*	.21	.21
	BMP (+RLE)	*N/A* *	*N/A*	*N/A*	*N/A*
	GIF	768,934 ‡	17.94	.14	.16
	JPEG (max.)	348,408	8.13	.32	.27
	JPEG (low)	74,811	1.75	.25	.20
	PCX	3,915,808	91.38	.28	.26

* 24-bit RLE compression is not generally supported in BMP

‡ Quantised to 256-entry colour palette

Fig. 5.2: File Sizes And Compression Of Test Images

Clearly image (a) is well suited to Run-Length encoding, although the GIF Lempel-Ziv Welch compression also gains a good ratio. JPEG high quality, even though not suited to this kind of image still comes out under 29% of the benchmark size. This type of image is suited to palette-based formats which perform Run-Length encoding.

Looking at the image (b) results, the story changes completely. Now the JPEG dominates, the low quality version storing the picture at under 2% of its original size. The quality of this image is still good enough to be used in Fig. 5.1 (b) instead of the BMP version. At full resolution, close inspection would be required to see any degradation. It is interesting to note that PCX, which always compresses data, regardless of colour depth, manages a 10% reduction. This highlights the poor results Run-Length encoding 24-bit images exhibits. As the image increases in size, so effort on the PCX encoding is wasted. This is the main reason why many applications do not support compression of 24-bit BMP files.

For image (b), I have also detailed the loading and saving times for each type under Adobe Photoshop on a 486DX-50MHz processor running in 32-bit mode. As one might expect, BMP load and save times are identical. After all, it does not need to perform any procedures to get at the data and so compression is the exact reverse of decompression. The GIF version has the quickest loading and saving times, due to it referencing a palette for each pixel. JPEG, in general, takes longer to compress data as there is much more processing required, although I was surprised to see the low quality save performing seven seconds quicker than high quality. My understanding of JPEG compression lead me to believe that the time to perform the process would be comparable whatever the quality factor, whereas clearly it is not. As already discussed, PCX loses out with RLE losing precious seconds on a compression ratio close to the uncompressed bitmap.

With only these two example image types, we have been able to identify some interesting characteristics on each of the formats. The uncompressed BMP is not suited to either type of test image, and so has been used purely as a benchmark for uncompressed data. BMP with RLE outperforms all the other formats with the simple test pattern, but cannot be used with 24-bit data. The GIF output is a good size on both types of image, but the fact remains that it only contains 256 colours so the 24-bit image has become lossy. JPEG, at any quality factor, is not suitable for representing simple solid-filled areas, as it is designed to work well with small pixel discontinuities such as in photo-realistic images like (b). However, with photographic images it is unsurpassed in this company, but at high resolutions how much data has been lost? The low quality JPEG has taken over 4Mb of image data and compressed it into 74,811 bytes. Many users may find it hard to accept there is so much redundant data.

If this study were to be extended to a wider range of formats, and possibly more varying test images, a clear picture could be painted as to which provide the best quality for certain types of application. Users with little knowledge of the suitability of such formats could then browse through this reference until they find the format right for them.

5.2.2 Quality Rating

Another way of looking at the problem of measuring image quality is to produce a set of ratings which can be applied to a test image and format to identify if the match rates as satisfactory to the developer of the system. Implemented correctly, it would provide a less subjective method than that of exhaustive testing described above. The tests could be carried out by the developer on a machine typical of the end-user's technology. The results would give an accurate measure as to the suitability of the tested formats. To consider how such a set of ratings might be constructed, we must identify the relationships between the factors already discussed and how they correlate with each other towards the final output of the image.

GIF images, for instance, have a maximum of 256 colours, which are represented by an internal colour palette. When a true-colour image is converted to GIF, quantisation must take place to reduce the 24-bits of colour information to a match, or close match, in the palette. This process has knock-on effects which alter the general behaviour of the image. Reading an image which uses colour palette indexing is significantly quicker than 24-bit reading, as the palette could be stored wholly in memory and as each index is read, so the RGB values can be ascertained; plus a palette index is 1-byte as opposed to 3 with true-colour. Therefore, GIF encoded files can be read and written significantly quicker than true-colour file formats, as has been proved in Chapter 5.2.1.

A further implication of GIF encoding is the lossy nature of quantisation. The scaling down of the colour information is subjective. The encoder must decide on the most suitable colour entries for the palette, and is unlikely to do a perfect job. In most cases, one can see elements of dithering on images which have been quantised in this manner. Whilst this dithering does help the overall quality of the image to the naked eye, the preservation of original image data is gone forever.

JPEG image quality casts another important shadow on devising image quality ratings. At what point can it be assumed the quality factor will start degrading the visual quality? Practically speaking, this depends on the image being compressed. Using too high a factor could result in a large output file and longer time for compression. Using too low a factor will result in the blocky effect, as demonstrated in Fig. 4.16. Either way, the original data is not preserved. Exactly how the quality factor affects the output quality can be analysed by dividing the image into blocks equal to the DCT size, 8x8 in standard JPEG, and looking at the pixel intensities of pixels 8 pixels apart across the whole image. If the differences are large, a low quality factor will cause blockiness. With the intensities closer, a low factor will not have such an adverse effect. The 'stripes' example image, Fig. 5.1(a), used in the previous section is a good example. The low quality factor version copes well horizontally, equalling the quality of the high factor one, whereas vertically, across the colour borders, the differences are more evident. To illustrate this further, refer to Fig. 5.3 below, which uses contiguous areas of colour in addition to low quality:

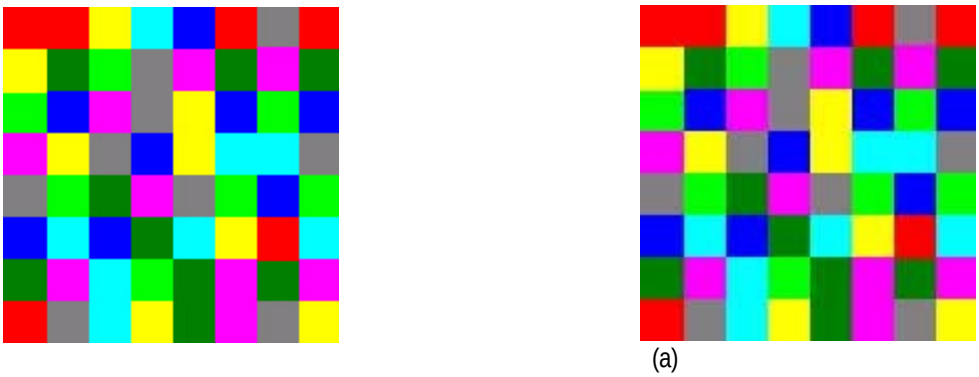

Fig. 5.3: Pixel Discontinuity Caused By Low Quality JPEG

In this version, image (a) has a high quality, whereas image (b) is of low quality. On both images, the borders of the colours highlight shadowing, darker areas on colour boundaries. However, this is much more prominent on the low quality image (b).

These relationships must be analysed in the same way for a set of format types. Only when this has been done can we begin work on a rating system to measure this. Although it can be argued that each format has its own unique way of affecting an image, there are general traits associated with lossless, lossy, palette indexed, JPEG encoded and all the other variations which can be approximated well enough to carry out this work. The only input that would be needed is a list of the features of any given format and the image itself. Our quantitative method for measuring the predicted output quality can gauge whether the format is suitable or not.

As a simple example, consider an input image of hand drawn cartoon strips involving significant areas of contiguous colour. If applied with JPEG, the score would instantly highlight the fact that JPEG compression does not mix with contiguous areas of colour, giving a low rating not only for the quality of the output, but also the file size and processing required to perform the operation. If GIF were used, however, the fact that it is palette-based will work in its favour, and as dictionary compression techniques such as LZW work well with contiguously coloured areas, the GIF format will gain a higher score (perhaps even the highest). These scores would be gauged by the resulting file size, the time taken to read and write the image, and most importantly, the visual quality of the image in the different formats in comparison to other tested formats.

An average user, armed only with a file format reference, such as Murray and vanRyper (1994), and some knowledge about the typical image contents for their application, can then set to work on finding the format best suited for the application. All without knowing about the formats or other subjective details which can cloud their judgement.

The specifics of such a rating or scaling system will not be discussed in this report as there is insufficient scope or time. To give this idea full justice would require much more time for carrying out the testing on many file formats, even before the analysis of their relationships can begin. As each format is analysed, so the system could be refined to take account of new features provided by a wider range of formats.

For example, if the first format chosen was BMP the features covered in the scale would be Run-Length or uncompressed encoding and the RGB colour cube. Extending this to the JPEG lossy format would incorporate the features of the JPEG compression stages (refer to Fig. 4.14) and possibly differing colour cubes. At each stage, if we were to apply this to an image the results would allow the comparison of more file formats. The more comprehensive the database, the better choice for the user.

The down-side to this approach is the unwieldy nature of all these format details. The complexity in involving more than a few formats may make it difficult and time-consuming to use, even for the experienced user. To avoid this the details could be implemented into a software system automating the analysis of formats for any given image. This reduces much of the burden on the user having to know anything about the process, and is likely to give more consistent results (as conditions should be identical every time). With relative ease, formats could be added or removed from the test results, or certain tests could be removed to save time, such as loading and saving times. To identify some features of the given input image may require artificial intelligence, or perhaps more simply, user input. Details such as whether the image is photo-realistic, line art, a cartoon-strip and so forth.

While it was my wish to tackle this problem, time has proved to be the overcoming factor, and as such, this is probably suited to a further project. As the idea of this kind of system has been formulated as I have worked on the project, it is not a task that can be added, due to its size and need for extensive development. Moreover, it highlights this as an area of future work.

6. Imagica Technical Documentation

Imagica is a graphic file viewer and converter which has been written with both Borland C++ and Borland Delphi, supporting Windows Bitmaps (.BMP), CompuServe Graphics Interchange Format (.GIF), Joint Photographic Experts Group JFIF (.JPG) and ZSoft PCX (.PCX).

Borland C++ was used to construct a Dynamic Link Library of graphics file format conversion functions in C. To do this, readily available freeware libraries (source code) has been utilised and adapted for this purpose by myself. The JPEG, GIF and BMP routines are taken from the IJPEG5b release (Lane, Gladstone et al, 1995). This is designed to be as portable as possible, so came with many platform-dependant build options. The one I have used is DOS (as Windows is not directly supported).

6.1 Design Principles

The design strategy of the graphics library is to convert the files to a common format which can be displayed and manipulated easily. The chosen format was BMP, as it is already fully supported by Windows and is a lossless method capable of storing 24-bit output. When a non-BMP file is opened, it is first converted to BMP and then opened as a BMP (but still treated as a foreign format). Each open file has two filenames associated with it: the name of the real file where the BMP version is stored, and the true filename the user has requested (in whatever format they wish). To the user, no indication of this method is known, so this process occurs transparently.

It has been done in this manner as bitmaps displayed under Windows are converted to the BMP format anyway. The only difference in this instance is that the library converts files to BMP files which can then be easily opened by Windows. Normally, these bitmaps would be stored in memory. It is a disk space versus memory trade-off, and as the conversion routines would already be using significant portions of the available memory storage, I decided that on machines with less memory (say, 4Mb or less) this may invoke heavy use of the swap file.

The User-Interface to this dynamic link library was constructed in Borland Delphi, an object-based visual development system which runs on the base Pascal language (or ObjectPascal in this case). My problems in designing and constructing this interface have been relatively minor compared to the graphics library. The main difficulty arose over the function calls and data-types used in them. As the library was constructed in C, there was the issue of data passing from the interface to the library and back again. While using the `pascal` keyword in the library functions enabled the interface to treat them as Pascal code, it did not overcome the problem of differing data widths between C and Pascal. Further research was required (from both books and the Internet) before I solved the problems. The documentation I had access to was ambiguous as to which data types were compatible between the languages. It was the assistance of other usenet newsgroup users that provided the clarity I needed.

6.2 Problems Encountered And How They Were Overcome

Initially, I was unable to get the first drafts of the compiled library to work within my application interface. It would compile with no errors, but when it was listed in the interface code as a DLL referenced function it would not allow the compilation of the interface. The only way I could get it to accept the library was to create procedures which took in no input parameters and returned no output. This meant that I could no longer test the actual routines as I was not able to receive any output. Convinced that there was nothing wrong in the code of the routines, I began to search for alternative reasons that might stop the integration. After much experimentation, I tracked down the problem to a switch in the definition file for the library. Including 'multiple' as a switch in this file meant it should be possible to load multiple instances of the library simultaneously. However, the point of dynamic link libraries is that they only require loading once and any further software which requires it can use the already open one. This was not reported as an error, or even a warning, and it was only after studying each of the options in turn that I found this to be the cause. Once removed, the DLL was accepted by Delphi and I was able to move on.

Dealing with the GIF portion of the library, the IJPEG software raised a problem. The source code had been designed with the JPEG central to the system. Converting JPEG files to BMP or GIF or converting BMP or GIF to JPEG was a fairly straightforward task which presented no problems. However, converting from BMP to GIF, or GIF to BMP was a different matter. The data structure used to store an instance of a JPEG decompression (to BMP or GIF) was different than that of a compression instance (from BMP or GIF). The crossover in the middle meant that the structures would be incompatible when converting BMP to GIF and vice versa. This proved to be insurmountable, although I did attempt to bridge the gap between the two by manually converting from the decompression structure to the compression, with no success.

Due to a lack of time, this has been bypassed by performing such conversions as a two-step process. Firstly, convert the BMP or GIF file to a JPEG one (using the already written routines) and then convert the JPEG to a BMP or GIF file. The obvious point to this is that JPEG being a lossy method will not preserve all the data from the original, and thus is not maintaining a true conversion (which should not lose detail, except with the GIF colour palette limit of 256 entries). However, with the JPEG quality factor set reasonably high (at a cost to speed) the results of these conversions is still good and unnoticeable to the naked eye.

In addition, I have unsuccessfully attempted to incorporated the PCX format into the graphics library. This has been taken from a different source (Murray, vanRyper, 1994 – CD-ROM) and has brought about the following problems:

- The data structure used in the IJPEG software is different to that of the PCX structure. This has meant that I needed to code a transfer routine to convert the PCX header values to the IJPEG compatible one for conversion to those formats. The same applies vice versa.

- The scanlines in a PCX file are read from the top-left pixel to the bottom-right. BMP (the central format in this application) scans bottom-left to top-right. The significance of this is that two buffers are required in main memory. Firstly, a smaller buffer to hold one uncompressed scanline. Secondly, a much larger buffer capable of storing the whole image uncompressed in memory. When the first PCX scanline is read in, it can be placed as the last scanline in the output BMP file (the RGB values, as stored in PCX, must also be reversed to BGR for BMP output). Although this seemed to be a straightforward task, it proved to be otherwise. Firstly, using C `far` pointers raises problems when the segment address of the pointer exceeds its limit and resets to the beginning of the segment. This means that the segment is not incremented as expected and important data is overwritten. The effect this had on the output was to split the image into bands of 50-100 horizontal scanlines, duplicating similar copies of the whole, squashed image in each band. After further research into the cause of this, I discovered the use of huge pointers, which automatically normalise and increment the pointer addresses in the manner I expected. This led to a further problem: the memory allocation function `farcalloc()` returns a `void far *`. I could not directly create a huge pointer and assign the `farcalloc()` to it. Fortunately this problem can be avoided by using two pointers. The first, a `far *`, is used to allocate the memory for the huge array. The second, a `huge *`, is assigned the pointer value of the `far *`, typecast to a `huge *`. Please refer to the source code in Appendix 2 for exact details.
- The third, and ultimate problem, lies in the way the PCX image data is structured. All the references from the books I consulted state that the PCX format stores image data either pixel- or plane-oriented. I was encountering problems with the colour in my output images, everything appearing greyscale (and duplicated three times across the image) as in the example below.

Fig. 6.1: Plane-Oriented PCX Data Misinterpreted As Pixel-Oriented

This, as I eventually found out was due to the fact that the data for each scanline was stored in the plane-oriented fashion (all the red values first, then Green and Blue) and I was attempting to read it in pixel-orientation (Red, Green and Blue values for a pixel stored contiguously). Thus I had to re-structure the routines for the PCX format to cater for this.

As an aside, I am still not aware how it is possible to tell if a PCX file is stored pixel- or plane-oriented (and the books are vague on the matter). The only method I could think of for determining which to use when decoding involves reading the first few values and if they are all similar (the intensity difference is small) assume it is plane-oriented. If the intensities in a pixel-oriented image were similar (i.e. a greyscale image) the decoder would incorrectly assume plane-orientation. I considered implementing a user-choice into the interface to allow the user to see both methods and then decide which was correct, but dropped the idea, deciding to note it in this report instead.

I have since found out from users of the newsgroup `comp.graphics.misc` that the orientation can be assumed by the number of bit planes per pixels. With true-colour, three planes are used for the Red, Green and Blue pixel values. For this, assume plane-orientation. When using palette indexes, usually the number of planes will be one, in which case the data is pixel-oriented.

Despite this important information, I have still been unable to get the PCX to BMP routine to work correctly. My attempts at re-orienting the data have been summarised in Fig. 6.2:

(a)	(b)	(c)
RGBRGBRGBRGBRGB	RRRRRRRRRRRRRRR	RRRRRGGGGGBBBBB
RGBRGBRGBRGBRGB	RRRRRGGGGGGGGGG	RRRRRGGGGGBBBBB
RGBRGBRGBRGBRGB	GGGGGGGGGGBBBBB	RRRRRGGGGGBBBBB
RGBRGBRGBRGBRGB	BBBBBBBBBBBBBBB	RRRRRGGGGGBBBBB

Fig. 6.2: Attempted Data Orientations For PCX Images

Pixel-oriented is (a), plane-oriented by image is (b), and plane-oriented by scanline is (c). Each of these methods has given different output results, none of which correctly match the original image. The closest of the attempted orientations has been (c), with the colour data working better than the other versions, although the offsetting of each scanline appears incorrect and the whole image is skewed. Fig. 6.3 demonstrates how a sample palette-indexed PCX image is affected by the conversion to BMP format.

(a)

Fig. 6.3: The Effect Of Imagica PCX Scanline Orientation

Image (a) is a PCX image converted to BMP format by Windows Paintbrush. Image (b) is the equivalent converted by Imagica

7. Conclusion And Evaluation

7.1 Evaluation Of Objectives And Aims

This project has turned out to be challenging in many ways. Each stage has presented its own problems to be overcome.

When collecting information from industrial sources, as covered by the first objective, I expected a lack of response. Previous experience had taught me that only a small percentage of sources are likely to respond at all. I tried to compensate this by applying to many varied sources, to improve the chance of receiving varied answers which could typify a cross-section of the graphics community. What I was not prepared for, however, was the lack of ideas for measuring quality used in industry. The general consensus of opinion of those who replied seems to be that a measurement system is not required for the most part, as the people working in this field already have experience with graphics file formats. In hindsight, it may have been better to also apply to some personnel not directly involved with the file formats, who would have less knowledge of the technicalities whilst still having an interest in their use. This could have highlighted more of a need for a quality measurement system such as discussed in this report. The responses I have received, however, have introduced some other problems I had not envisaged, such as proprietary formats supplied with each new application. When running more than a few of these simultaneously, difficulties can arise when there is a need to move data between different software systems. Although OLE (object linking and embedding) has been designed with this in mind, it is not always possible to transfer data if the structure is foreign to that recognised by other systems.

My suggestions on how a measurement could be made on these graphics file types has been made possible through my research into the popular formats and techniques already in use. This second objective, for me, has been the most difficult. Working from a set of techniques towards a generalised method of rating formats has not been easy. Throughout my research on the available formats I have been formulating my ideas into ways this could be done. While I am pleased with the ideas that have come to light, it is unfortunate that time has run out without me being able to give this section as much attention as I feel it deserves. If the workload had been shifted from the research towards the suggestions I may have been able to develop my ideas further, although doing this may have restricted my understanding of the current facilities available to the graphics programmer. Whichever may have been best, I believe I could have made good use of extra time to develop the concepts introduced in Chapter 5. The challenge put forward by this section has identified new ideas as to how this might be carried out. Furthermore, I have been able to see first hand the difficulties encountered when attempting to quantify the enormous range of graphics formats.

An area which has proved to be straightforward is finding research information on the file formats and their compression. The diversity of resources is very large, and I have had very little problem in locating details of the topics. In some respects, there has been too much information, especially from the Internet, requiring the identification of the most reliable sources for my needs. My third objective, researching and understanding these formats and their techniques, has been accomplished successfully due to this wealth of information.

With some of the advanced techniques for compression discussed in Chapter 4.4, finding readable details which do not dwell on mathematics too heavily has not been easy. As most readers interested in the complexities of these are likely to have a knowledge of the mathematics, I have had to understand what the purpose of the techniques are without involving the low-level composition. This work has given me sufficient understanding to argue for and against the merits of such advanced formats, and I am satisfied that the fourth objective for this has been used well.

The remaining core objectives relate to the software component, although the reason there appears to be more emphasis on the software is for clarity only. Separating the tasks into distinct objectives seemed most natural in terms of dividing the available time. With regards to the software, the only item which has been short of the target is the inclusion of the PCX format. Despite all the references I have used to help me with this, I have found the information to be vague about true-colour PCX images. As the PCX format was designed before true-colour images were possible, much of the documentation is based on the original specification which uses palette colouring only. This was not a problem I anticipated. I had read about PCX beforehand, but was not aware of the bias of the information available. Although I was able to get further details from other users of the Internet, I still could not complete the routines.

In this respect, considerable effort has been expended on a portion of the software component not present in the final version. The upside of this is that it has given me the chance to see the typical kinds of problem which can be encountered as a beginner. The fact that a single format can have more than one internal representation has caused this problem. The software writer has the burden of supporting all the possible structures so the routines are fully format-compliant. The attention to detail required in doing this can make this process slow and meticulous.

Furthermore, as graphics are reliant on large data structures, I have had to overcome the problems Microsoft DOS (and Windows) places on the programmer, relating to the memory system. For structures greater than 64Kb, these extra necessities have raised some unforeseen problems which have hindered development. Please refer to Chapter 6 for further details.

The objectives which involved learning Borland Delphi and the Windows API (application programming interface) have been undertaken well. This has been an ongoing process with new features being learnt as and when I have thought they were necessary. In reality, there are a lot of similarities and connections between the two, meaning that when I learnt a concept in ObjectPascal, it usually could be applied to the Windows API also. In this instance I am referring to the clipboard features, for which I considered two approaches based on the ObjectPascal and API systems. For while the API provides greater freedom to the programmer, ObjectPascal generally supplies a simplified method, using the same features with the unused details removed.

In addition to the work I have carried out on the core objectives, I have also been able to provide work on the advanced aims. The clipboard features have been implemented in a limited way, in that Imagica is able to copy and paste whole images to and from other applications via the clipboard. I had hoped to implement a utility to allow sub-sections of any image to be copied, cut or cropped but was unable to find sufficient information as to how this could be done.

The online help system is complete, and contains an extensive list of the error messages which may occur, with their possible causes. A shortage of time has prevented the image rotation and flipping features from being included, although image magnification has been successfully incorporated. From the interface any image can be increased or decreased in both the *x* and *y* axes independently, or reset to the image's original dimensions. In addition, beyond the specified advanced aims, printing capability has been added although tests indicate that this feature only functions when used with colour printers (only the PostScript header is sent if the printer is black and white).

Generally the software is at the stage I had intended. I am pleased I was able to do so much of it as I had concerns during the project as to whether or not it would be completed. Understanding the shareware software so I could implement it in my DLL proved to be difficult. The complexity involved meant I had to spend significantly longer trying to understand how it worked before I could apply it to my application. This is, perhaps, highlighted in the fact that PCX was not fully successfully incorporated.

The type of image display in use also affects the visual appearance of Imagica. When using a set-up which can cope with more than 256 colour on-screen at once, no problems arise. However, on lower colour depth displays (256 colours or less) true-colour images are not displayed correctly. This is because the images would require quantisation to reduce the 24-bits of colour information to 8-bits as suitable for display. Furthermore, such lower depth output also affects the magnification features so the colours fail when used. Only when the image is normalised in both axes does the colour of 8-bit images return to its correct state. These were not tackled as this problem is beyond the scope of this project. On higher depth displays neither of these problems will be evident.

7.2 Evaluation Of Project Management

The Gantt Chart in Appendix 1 details the original project plan set in October 1995. It lists the original topics this project had been classified under, and the expected date to start and approximate completion date. When choosing this time-scale, I had to take other considerations, such as course-work, into account. I felt it was important to have a clear plan so I would always know what could be worked on at any given time. On a study of this size it is crucial to know at the start exactly what needs to be done so the work can be paced and pre-researched accordingly.

The topics have been separated as much as possible to allow independent work units to be carried out simultaneously with little need for information from other units. The exception to this rule was the suggestions I have made as to how I believe image quality can be measured. This required that I had information from my industry sources beforehand, as this was the main basis of the work. Furthermore, it was also essential that my research into the file format details was carried out alongside it, so the technical knowledge was also available.

Due to the fact that I have placed myself under this working guideline, my project management has been quite successful. One area I should have given more time to was the collection of information from industrial sources. The replies I received were not prompt. The first was returned within one month, but then I had to wait until the end of the Autumn term before the others arrived.

Even more significantly, after I had finished the initial composition of Chapter 2, I received further information from a representative of Erdas, publishers of image processing software. The details given to me at that time were contact names and addresses of people at the Universities of Greenwich, London and Sheffield who had a particular interest in this field. Fiona Cocks is a University lecturer at Greenwich who previously undertook a study in colour perception for her MSc. Dr. Chris Clarke from Sheffield is also a lecturer whose students have been involved in similar, graphics-based work in the past. Due to the late arrival of this information, I have not had time to pursue these lines of enquiry, and so it is mentioned here purely as an addendum.

My plan has allowed three weeks at the end of the project with no tasks remaining. This does not mean that I expected to finish three weeks early, but that I expected some of the work to fall behind schedule. This period is the buffer by which I have been able to finish off tasks in time for the deadline. Fortunately, I have been able to make good use of this buffer to make up the time I lost with the slow responses detailed above and the difficulties encountered with the software.

Further to the chart plan, I have also been attending regular meetings with my project supervisor. This has allowed me to sub-divide the work units again so each meeting had a target associated with it, ensuring the work-rate remained consistent. Whenever problems were met, I could then discuss them and come to a decision on the best method for solving it. In my opinion, this has worked well, and has helped me maintain the flow without getting entangled in any one problem for too long.

7.3 Further Work

Certain elements in this project leave scope for further development. With almost any project which includes a software component, a list of future enhancements could be endless. In this case, I will only highlight the general areas where extra work would benefit the project.

Apart from the obvious inclusion of more formats into the graphics library, which is discussed below, the next step could be to include the kind of image processing options found in the likes of Adobe Photoshop, Erdas Imagine and Visilog. It depends on the direction of the software, although in my opinion, the multitude of current applications which already do this, including the ones already mentioned, are sufficient for most user's needs.

As mentioned in Chapter 5, to fully explore the implications of an image quality measurement system would require more time. While I have outlined the principles behind how this could be done, the actual work involved would be suited to a further project. To develop the system specified in Chapter 5.2.2 would naturally lead to its inclusion into the base of Imagica, the software already written. With the base in place, development could be concentrated on the measurements aspect, currently non-existent. The supported format base could then be extended by adding extra file formats to the graphics library in the same manner. As each new image is loaded, instead of merely being displayed on the screen, the application could set about analysing the image and converting it to the supported formats so the tests can be carried out, all automatically.

This appears to hold a better future for Imagica. Currently, to the best of my knowledge, there are no such products on the market which can analyse formats and rate them against each other in this way. Despite the general opinions of those replying to my questions that such a system is not essential, I feel sure that untrained users could find it useful and would prefer it to a reference book.

8. Bibliography

8.1 General References

Borland International (1991). *Borland C++ 3.0 Library Reference*, Scotts Valley, CA: Borland International.

Borland International (1991). *Borland C++ 3.0 Programmer's Guide*, Scotts Valley, CA: Borland International.

Cantù, Marco (1995). *Mastering Delphi [incl. CD-ROM]*, Alameda, CA: Sybex.

Sprigg, Graham (ed.) (1995). *Image Processing*, Volume 7: Issues 1-6.

Jackson, Richard and MacDonald, Lindsay and Freeman, Ken (1994). *Computer Generated Color: A Practical Guide to Presentation and Display*, Glasgow, Scotland: John Wiley & Sons.

Langdon, Glen G., and Rissanen, Jorma (1981). Compression of Black-White Images with Arithmetic Encoding. *IEEE Transactions on Communications*, **COM-29**(6), pp858-867.

Murray, James D. and vanRyper, William (1994). *Encyclopedia of Graphics File Formats [incl. CD-ROM]*, Sebastopol, CA: O'Reilly & Associates.

Nelson, Mark (1992). *The Data Compression Book.* New York, NY: M&T Books.

8.2 Specific References

Iterated Systems, Inc. (1994). *Images Incorporated.* Norcross, GA: Iterated Systems, Inc, Appendix A.

Sprigg, Graham (ed.) (1995). The Unsolved Problem of Image Compression. *Image Processing*, **7**(6), pp38-40.

Jackson, Richard and MacDonald, Lindsay and Freeman, Ken (1994). *Computer Generated Color: A Practical Guide to Presentation and Display*, Glasgow, Scotland: John Wiley & Sons, pp236-239.

Murray, James D. and vanRyper, William (1994). *Encyclopedia of Graphics File Formats [incl. CD-ROM]*, Sebastopol, CA: O'Reilly & Associates, pp 460.

Nelson, Mark (1992). *The Data Compression Book.* New York, NY: M&T Books, pp 360.

8.3 Internet References

ftp://ftp.uu.net/graphics/jpeg/jpegsrc.v5b.tar.gz

- Lane, Tom and Gladstone, Philip and Ortiz, Luis and Boucher, Jim and Crocker, Lee and Phillips, George and Rossi, Davide and Weijers, Ge' (1995). *Independent JPEG Group JPEG Software (Release 5b)*.

http://www.cis.ohio-state.edu/hypertext/faq/usenet/compression-faq/part2/faq.html

- Guttmann, Peter (1995). Introduction to Data Compression. *comp.compression Frequently Asked Questions (FAQ) (Part 2)*, item 70.
- Kominek, John (1995). Introduction to Fractal Compression. *comp.compression Frequently Asked Questions (FAQ) (Part 2)*, item 77.
- Lane, Tom (1995). Introduction to JPEG. *comp.compression Frequently Asked Questions (FAQ) (Part 2)*, item 75.

http://www.cis.ohio-state.edu/hypertext/faq/usenet/graphics/colorspace-faq/faq.html

- Poynton, Charles A. (1995). *Color-Space FAQ: Frequently Asked Questions about Color and Gamma*.

http://jaring.nmhu.edu/delphi.htm

- Summers, Wayne (1996). *Delphi Programming*.

Usenet:

- comp.compression
- comp.graphics.algorithms
- comp.lang.c
- comp.msdos.programmer