University of Durham

Department of Computer Science

An Investigation into Quadtree Fractal Image and Video Compression

By

James Halliwell

M.Sc.

2006

Abstract

Digital imaging is the representation of drawings, photographs and pictures in a format that can be displayed and manipulated using a conventional computer. Digital imaging has enjoyed increasing popularity over recent years, with the explosion of digital photography, the Internet and graphics-intensive applications and games.

Digitised images, like other digital media, require a relatively large amount of storage space. These storage requirements can become problematic as demands for higher resolution images increases and the resolution capabilities of digital cameras improve. It is not uncommon for a personal computer user to have a collection of thousands of digital images, mainly photographs, whilst the Internet's Web pages present a practically infinite source.

These two factors – image size and abundance – inevitably lead to a storage problem. As with other large files, data compression can help reduce these storage requirements. Data compression aims to reduce the overall storage requirements for a file by minimising redundancy. The most popular image compression method, JPEG, can reduce the storage requirements for a photographic image by a factor of ten whilst maintaining the appearance of the original image – or can deliver much greater levels of compression with a slight loss of quality as a trade-off.

Whilst JPEG's efficiency has made it the definitive image compression algorithm, there is always a demand for even greater levels of compression and as a result new image compression techniques are constantly being explored. One such technique utilises the unique properties of Fractals.

Fractals are relatively small mathematical formulae that can be used to generate abstract and often colourful images with infinite levels of detail. This property is of interest in the area of image compression because a detailed, high-resolution image can be represented by a few thousand bytes of formulae and coefficients rather than the more typical multi-megabyte filesizes. The real challenge associated with Fractal image compression is to determine the correct set of formulae and coefficients to represent the image a user is trying to compress; it is trivial to produce an image from a given formula but it is much, much harder to produce a formula from a given image. In theory, Fractal compression can outperform JPEG for a given image and quality level, if the appropiate formulae can be determined.

Fractal image compression can also be applied to digital video sequences, which are typically represented by a long series of digital images – or 'frames'.

For my wife Sonia,

Whose love and support made it all possible.

Table of Contents

Table of Illustrations

1.0 Introduction

There are currently a number of techniques being developed within research communities in both computer science and mathematics that aim to reduce the storage requirements of both generic data objects, bitmap images and digital video sequences. Bitmap compression often borrows from generic data compression techniques and interleaves these with content-specific compression methods that are only suitable for bitmap image compression; video compression tends to borrow heavily from image compression techniques and again applies specific algorithms tuned to the characteristics of digital video.

The desire to compress computer data objects (including binary executable, text documents, images, sound and video) originally stemmed from the scarcity of data storage available on personal computers - less than 10 years ago, many personal computers relied solely on floppy disk media or small (<500 megabyte) hard disk drives for offline storage. As a result of this, it was highly desirable to reduce the storage required by data objects to conserve and minimize usage of available storage space. The performance and time penalties involved in compressing and subsequently decompressing the data outweighed the expense or inability to store it verbatim.

Although PCs now have much greater storage capacity, the advent of the connected world and specifically the Internet has once again driven the need for data compression to minimize the bandwidth requirements and costs for the transmission of such data objects, rather than the data storage requirements of the data itself.

Digital images and video are common data objects, and are also likely to be relatively large in size. A typical digital photograph may occupy 10 megabytes whilst high-quality digital video requires substantially more space, with a feature-length film can occupy nearly 1 terabyte. As a result, both images and videos are a major focus of data compression efforts. Unlike generic data files, such as documents and executables, there are specific compression algorithms that will

only compress images and also specific algorithms for video. These specific methods are able to deliver high levels of data compression for both data and video, though there is a continued demand for innovation in this area.

One specific method of image compression technique is Fractal Image Compression. A fractal is iterative function whose output can be used to plot a particular image. The specifications of the function being used are generally many times smaller, often just a few hundred bytes, than the image that can be depicted by them. As a result, if the function is stored instead of the bitmap image that it would produce, then a large level of compression is achieved.
However, the major issue with this method is that whilst it is easy to take a fractal function and produce its associated image, it is not easy to take an arbitrary image and deduce the appropriate fractal function that we wish to store. This problem is computationally extremely difficult and there have been a number of algorithms and techniques developed to tackle it.

This thesis aims to research and evaluate this are of compression, including an understanding of the issues and challenges that fractal image compression presents and how feasible this method of encoding arbitrary images actually is.

1.1 Objectives

The following objectives outline the focus of this thesis and the areas of research that are to be investigated.

1. Evaluate Fractal and conventional image compression techniques. This should involve a literature survey to cover the factors involved in digital image representation, current compression techniques, and previous fractal image compression methodologies. The literature survey is presented in chapter 2.

2. Develop a software toolset that can readily compress images using a quadtree algorithm and allow meaningful and measurable performance comparisons to be made. As part of this process, there is a requirement to select an appropriate development platform, environment and data structures. Specifications and implementation details for the software toolset are covered in chapters 3 and 4.

3. Evaluate the performance of quadtree image compression in terms of image quality and levels of compression achieved using the newly developed software tool to investigate quadtree compression through the selection of appropriate and varied trial images. A detailed analysis of the quadtree implementation is provided in chapter 5.

4. Compare quadtree compression performance to the industry standard JPEG compression system in terms of image quality and the amount of compression actually achieved. This should be done using a series of trial images and at various compression levels. Results from both compression types are given in chapter 5.

5. Assess the computational intensity of a fractal compression implementation and understand which aspects of the algorithm significantly impact computing performance. Performance is discussed in chapter 4, whilst recommend areas for improvement that currently reduce the computational performance of the implementation are discussed in chapter 6.

1.2 Structure of Thesis

The remainder of this thesis details the progress and developments against the above objectives. Chapter 2 presents a comprehensive overview of data compression techniques, digital image representation, and image compression techniques including JPEG and fractal compression

methods, including quadtree partitioning. Chapter 3 details the selection of a suitable fractal compression method and the construction of a suitable algorithm around this method that can be readily implemented. Following on from this is Chapter 4, which covers the implementation of the chosen compression method as a software toolset and includes discussion of development environments, data representation issues and core functionality for the toolset. Chapter 5 presents the results achieved using the developed toolset to explore the quadtree compression method and how well the various trail images can be represented using the system – both in terms of image quality and the level of compression achieved for each. Finally, chapter 6 looks back at the objectives presented here and discusses how each have been achieved, together with suggestions for improvements and further research and a summary of this work.

2.1 Data Compression

A number of techniques have been developed to achieve this compression, and these can be categorized as either generic or content-specific compression techniques.

2.1.1 Generic Compression

Generic compression systems are designed to be able to take any form of data from any application or program and apply their general-purpose compression algorithms. As the algorithms have no awareness of the nature of the data being compressed, it is extremely important that no data is lost or changed as a result of the compression/decompression process and the decompressed data is bit-for-bit identical to the original data. This compression technique is known as lossless compression and can be used to compress any type of data. Lossless compression is used where any loss of information is unacceptable, for example binary executables. [HELD, 1983]

An example of this class of compression system is PKZip. PKZip can take any binary or ASCII file as input and apply lossless generic compression algorithms to the data, producing a compressed file that is generally smaller than the original. This output file has to be decompressed in order to make the data readable by the original system or application, this often introduces a separate step in the file handling process as such compression schemes are rarely integrated with end-user applications.

Generic lossless compression aims to reduce data storage requirements by removing redundancy from files, such as repeated byte/bit patterns. Amongst a number of popular algorithms for this is Run-Length Encoding, or RLE. This algorithm encodes streams of repeated data into a more compressed 'abbreviated' form, e.g. the following sequence

11111280928133320000000000000000000000000000000027309222222222222222

can be represented as

$$1_5 28092813_3 20_{33} 273092_{14}$$

RLE uses a run-length value to indicate that a value is repeated a number of times in a sequence [TANENBAUM, 1996]. Storing the value and run-length instead of the repeated sequence gives rise to storage savings and data compression. The original data can be recreated by reading the initial values back and recreating the original runs within the sequence. However, RLE is a very basic system, and if there is too much variation in the bit pattern then the resulting 'compressed' image can actually end up larger than the original, known as expansion [FISHER, 1995].

Other techniques, such as Lempel-Ziv-Welch [MIANO, 1999] are more advanced. LZW encodes a byte sequence by maintaining a dictionary of value sequences so far encountered and replacing instances of these value sequences with corresponding codes. Shorter dictionary codes are used for sequences which appear more frequently in the original sequence, resulting in an overall reduction in data storage requirements.

Both RLE and LZW rely on data-redundancy to achieve data compression. Indeed, if a byte sequence is truly random and does not contain redundant sequences then neither technique will be able to achieve any data compression. The most efficient way of encoding truly random data is in its original format without applying any compression whatsoever. This restriction applies to all compression schemes [WILKINSON].

2.1.2 Content-Specific Compression

Content-specific techniques are designed for the compression or a particular type of data, such as bitmap images. As a result, they are unsuitable for the compression of generic data sequences. However, such techniques are highly advantageous as they are able to take advantage of the characteristics of the data structure itself to maximise the compression

achieved. In terms of image and multimedia compression, these techniques actually become part of a file format for a particular media type, allowing the compression and decompression to be applied where appropriate within the application software itself. [MIANO, 1999]

For example, CompuServe's GIF file format also includes LZW compression and this compression/decompression software is performed transparently by any image manipulation or Web-browsing software that handles this file type.

Aside from the integration and automation of the data compression routines, such content-specific methods, offer major compression advantages over generic techniques, and some also optimise the structures they compress. Because such methods are aware of the both the data-storage characteristics and, perhaps more importantly, the characteristics of the target 'audience' that will subsequently view or experience the data, they are able to permanently lose selected elements of the original source data without unduly impacting on the 'quality' of the data once it has been decompressed. That is, once the original data has been compressed and subsequently decompressed, it may not be bit-for-bit identical to the original. This compression approach is known as lossy compression as this technique inevitably involved the loss of some data.

However, this loss of data is not as serious as it initially appears. Successful lossy compression algorithms are designed to throw away data which has little or no bearing on the 'quality' of the compressed output for that content type whilst making significant storage savings within the algorithm itself. By doing this, lossy algorithms can not only remove run-length and sequence-based redundancy from source data, they can avoid storing portions of it at all. Subtle changes in the appearance and perception of the compressed output can often be tolerated in returns for the substantial compression gains achieved using lossy techniques. Ideally, the errors generated by throwing data away will arranged so that they are very difficult to detect, and so the algorithm must understand the perception characteristics of their target 'audience'. [TANENBAUM] [MIANO, 1999]

With the demands for data-compression being increasing driven from the data transmission rather than storage perspective, lossy compression techniques are increasingly coming to the fore. Whilst a lossless compression algorithm cannot guarantee a particular level of compression, expressed as a ratio of original data size and compressed data size, lossy compression systems can. This ability stems from their ability to throw away less useful data, and lossy algorithms can continue to do this for a given data source until a desired compression level is reached. This is important in the communications world because the traditional leased-line telecom links that form the Internet offer a fixed data throughput rate per second. If a compression algorithm can guarantee to remain within this throughput rate by virtue of lossy data compression then it becomes possible to send such data in real-time without the risk of over-subscribing the telecom link. [TANENBAUM]

2.2 Digital Imaging

The most prevalent method for representing images within a computer system is known as a bitmap. Images ranging from simple line-drawings to high-resolution colour photographs can be represented using a bitmap structure. Bitmaps are also commonly used to represent the screen output of a computer system. A bitmap image is composed of number of discrete addressable screen/image elements known as pixels, where each pixel has a single, uniform colour. These rectangular pixels are arranged in a 2-dimensional array structure.

Image details and tonal transitions are represented by varying the colour of the bitmap's pixels accordingly. A bitmap image can be thought of a digital sample of an analog real-world image. As each pixel can only be a single colour, the level of accuracy of the representation in comparison to the real-world image, including both detail and colour transition, is determined by both the number of pixels used and the range of possible pixel colours. The more densely packed pixels are within a given physical screen area, the more smoothly and sharper image details can be shown; whilst the greater the number of colours available for the bitmap, the more accurately original image tones can be represented. These two factors are illustrated in figures 2.2.1 and 2.2.2.

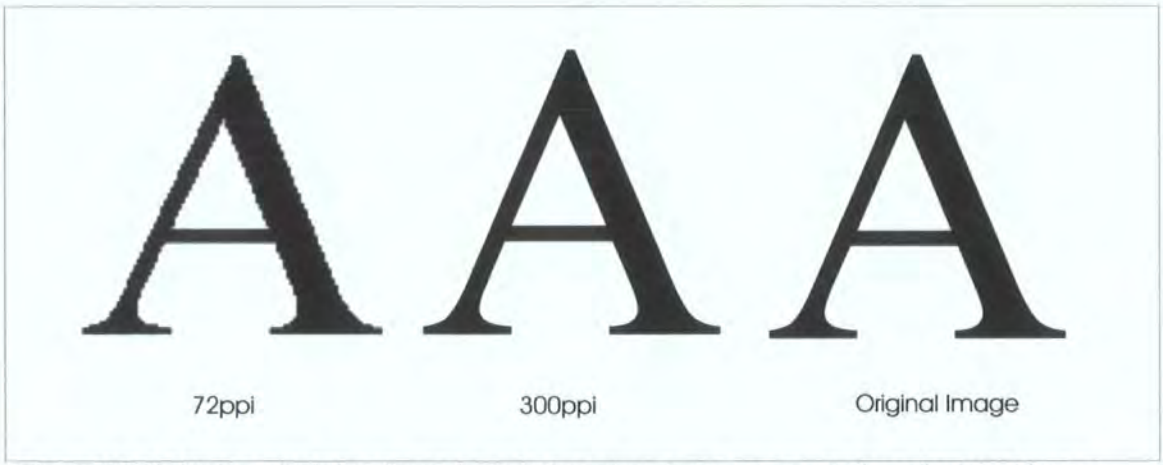


Figure 2.2.1: Higher pixel density and resolution gives truer representation of original image

Figure 2.2.2: Greater colour depth allows more gradual tonal transitions

In figure 2.2.1, it can be seen that increased resolution gives a more detailed and aesthetically pleasing representation of the original image. A major issue with low-resolution bitmaps is that they suffer from 'jaggies', or aliasing around portions of image detail that involve curves and other non-rectangular detailing. In figure 2.2.2, the number of colour levels that can be used within a given bitmap is shown to have severe effect when storing images with continuous tonal graduations or high-colour detail, such as photographic images. It must also be noted that due to each pixel only containing a single colour tone, a low bitmap resolution also causes problems with tonal representation, as there may not be enough pixels in an area of tonal change to allow for a sufficient number of tones to be deployed in the bitmap representation.

Fortunately, both the resolution (the number of pixels available, expressed as *H(orizontal) x V(ertical)* pixel array dimensions) and the colour-depth (i.e. the number of discrete colours that can be present within a single bitmap image) available on PCs have greatly increased

over the last few years to a point at which high resolution 1280x1024 resolution images in true-colour can readily be displayed and manipulated.

2.2.1 Colour Models and Colourspace [MIANO] [TANENBAUM]

The colour of an individual pixel is represented by a bit-sequence. The number of bits in this bit-sequence naturally determines the number of colours that can be present in an image, and is known as the colour sampling precision:

1 bit allows for 2^1 colours – i.e. black or white only (monochrome)

4 bits allows for 2^4 colours – i.e. 16 colours

8 bits allows for 2^8 colours – i.e. 256 colours

24 bits allows for 2^{24} colours – i.e. 16777216 colours (referred to as 'true-colour')

There are a number of methods to translate a numerical value to a colour, these methods are known as colour models. Colour models are often developed for use with particular physical display device: the RGB colour model is used for display with computer monitors based on Cathode Ray Tube or Liquid Crystal Display technologies, where each physical display screen pixel's colour is produced by varying the physical intensities of Red, Green or Blue light. The CMYK colour model is used for output to printing devices, where each colour is composed of differing levels of Cyan, Magenta, Yellow and blacK inks. [MIANO, 1999] [TANENBAUM, 1996]

2.2.2 RGB Colour Model

The RGB colour model is used in display devices such as computer monitors. 24-bit True Colour in the RGB colour model uses 8-bits for each of the three primary colours, giving a possible 256 levels for each, i.e. $2^{24} = 256^3 = 16777216$. The range of colours that can be represented using a particular colour model and sampling precision is known as the colourspace. The RGB colourspace is shown in figure 2.2.2.1.

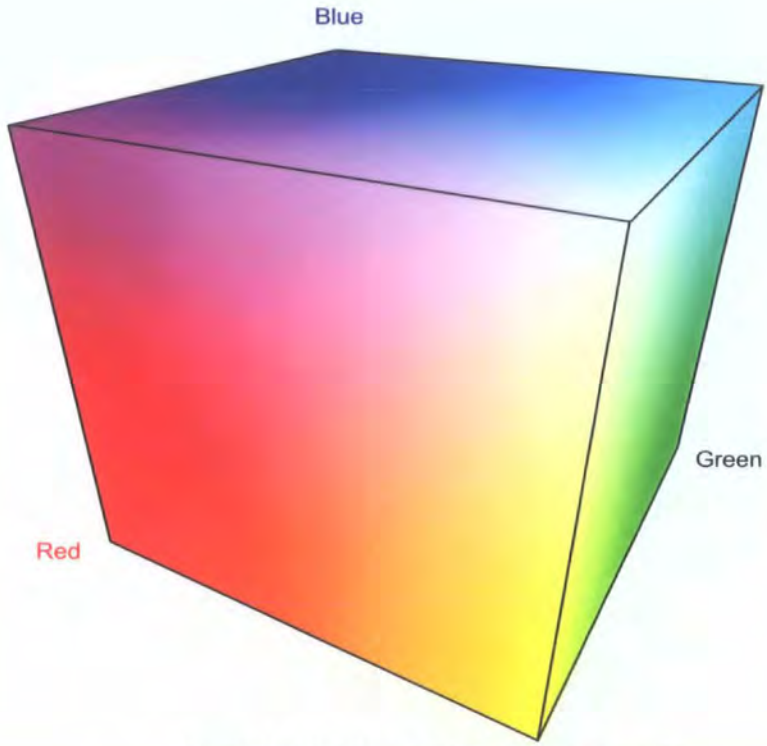


Figure 2.2.2.1 The RGB colourspace cube

The RGB colourspace is represented in three dimensions, one axis for each of the primary colours. An enormous range of colours can be duly represented with the RGB model – it can be seen that red+blue gives magenta, whilst blue+green gives cyan, etc. Within the cube, greyscale shades are also represented, and these run in a diagonal line between the black and white corners of the RGB cube, where each 'colour' consists of an equal amount of red, green and blue. This is illustrated in figure 2.2.2.2.

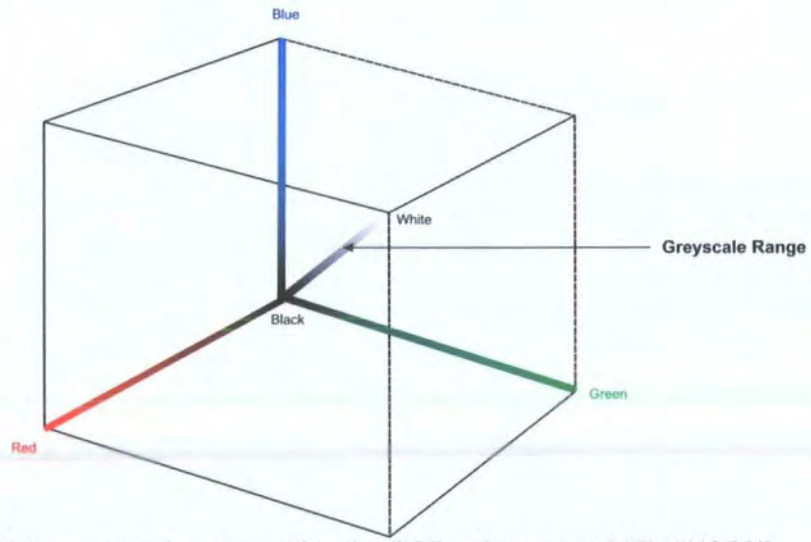


figure 2.2.2.2 The grayscale range within the RGB colourspace [WILKINSON]

Greyscales play an important role in digital imaging. Apart from the obvious need to use them to represent intermediate tones between the black and white monochromatic extremes, within the RGB colour model the individual primary colour components are each represented as 256 level (or 8-bit) monochrome bitmaps. Each pixel within the greyscale bitmap has a grey level and this represents the intensity of that particular primary colour in the final RGB image. These primary colour 'sub' bitmaps are called channels. [Miano, 1999]

2.2.3 Luminance and Chrominance – The YCbCr Colour Model

RGB and CMYK are not the only colour models in use. In the analog television world, picture colour information is also divided up into channels. The PAL system used three colour channels. The Luminance channel (Y) represents the intensity of the image, whilst two Chrominance channels (U and V) represent the colour information. The Y channel is a greyscale version of the colour transmission that would be formed by all three channels and this allows 'black and white' televisions to view the colour transmission signal.

In the digital world, the YCbCr colour model is similar to the PAL model in that it also uses three channels, one luminance (Y) and two chrominance channels (Cb and Cr). Again, Y represents the intensity of the composite image and is a greyscale representation of the composite image, whilst Cb specifies the blueness of the image and Cr specifies the redness. The most important property of the YCbCr (and also the PAL) colour model is that the Y channel contains much more useful information about the composite image that the Cb and Cr channels. Y is the most important channel because the human eye is much more sensitive to variations in luminance (intensity) than it is to variations in chrominance (colour) [KINGSBURY]. In fact, the maximum Cb chrominance sensitivity of the eye is only one-half of the Cr chrominance, which itself is only one-third of the maximum luminance sensitivity, as shown in figure 2.2.3.1.

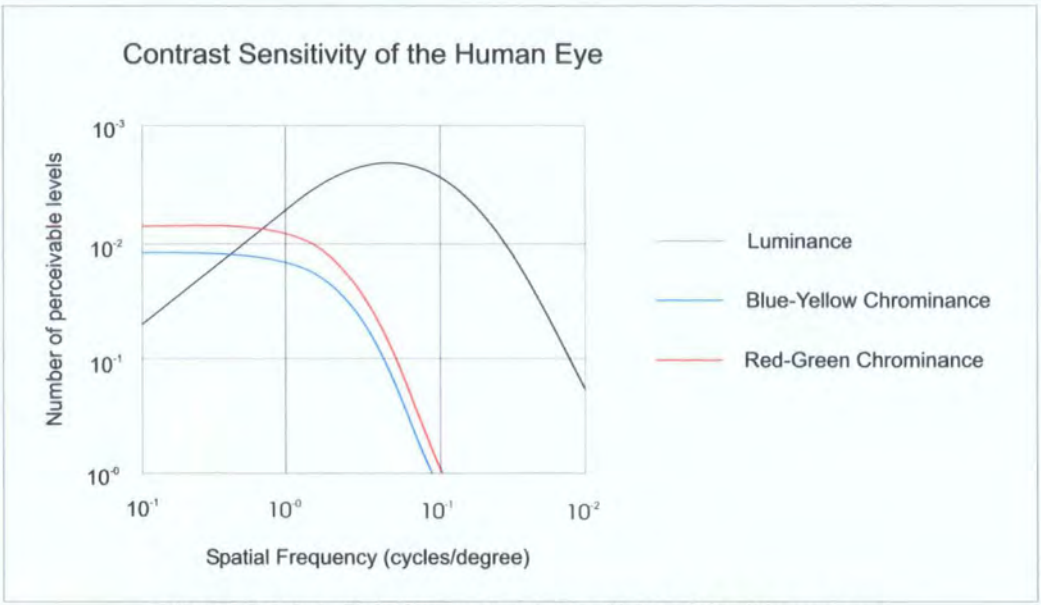


Figure 2.2.3.1: Relative contrast sensitivity of the human eye

As with the PAL model, the Y channel in the YCbCr model contains more useful information, as can be seen in the channel breakdown of a colour photograph in figure 2.2.3.2.

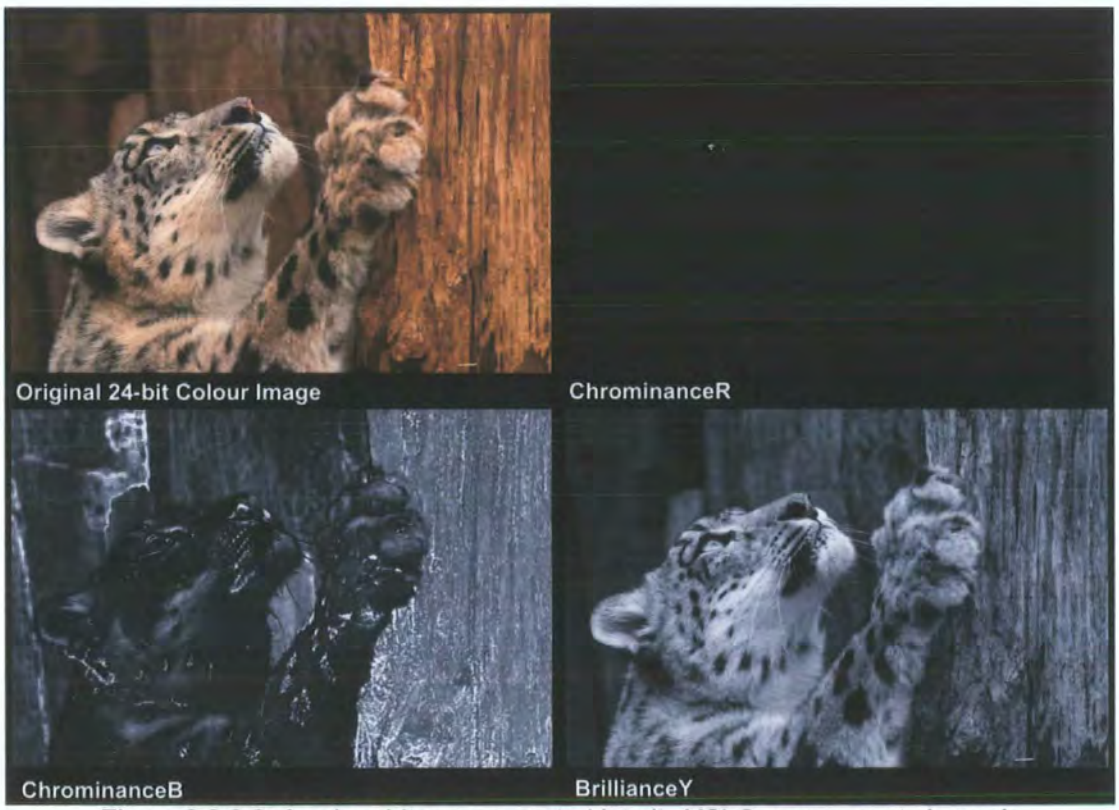


Figure 2.2.3.2: A colour bitmap separated into its YCbCr component channels

Given the low chrominance sensitivity of the eye and the heavy weighting of image detail to the Y channel within the YCbCr colour model, it would be feasible for a lossy compression algorithm to make savings by allowing less information to be recorded from the two

chrominance channels than from the more important luminance channels. Therefore, it is highly advantageous to convert images to the YCbCr colour model to aid in image compression. The formulas for conversion between the RGB colour model and the YCbCr colour model are as follows:

$$Y = 0.299R + 0.587G + 0.114B$$

$$Cb = 0.1687R - 0.3313G + 0.5B + 2^{24/2}$$

$$Cr = 0.5R - 0.4187G - 0.0813B + 2^{24/2}$$

$$R = Y + 1.402Cr$$

$$G = Y - 0.34414(Cb - 2^{24/2}) - 0.71414(Cr - 2^{24/2})$$

$$B = Y + 1.722(Cb - 2^{24/2})$$

Arguably the most successful lossy image compression system is the JPEG image compression standard, frequently used to compress detailed photographic-quality images. JPEG exploits the properties of the YCbCr model to achieve extremely high compression ratios. [Miano, 1999] [Tanenbaum, 1996]

2.3 Bitmap Image Compression

High resolution bitmap images with large colour-depths require a significant amount of storage space – a 1280x1024 bitmap with a 16 million colour palette requires 3.75 megabytes of storage. i.e.1280 *x* 1024 *x* 24 bits = 31457280bits or 3.75megabytes. Higher resolution images for publishing and photography can require tens or hundreds of megabytes. Such large file sizes have made image compression an important topic and there are a number of compression techniques available.

Bitmap images, by their very nature, conform to a rigid structure – a matrix of *h x v* pixels, with each holding a single colour value. In addition, the target device for a bitmap is always, eventually, the human eye. Finally, many bitmap images contain large amounts of

redundancy in terms of large numbers of contiguous pixels with identical (or very similar) colour values.

Bitmaps that represent drawings typically use 4 or 8-bit colour sampling. Drawings tend to have large blocks of solid colour, for example in figure 2.3.1.

figure 2.3.1: Drawing of Notre Dame de Paris

Such simple bitmaps lend themselves well to traditional non-lossy compression methods, such as RLE, where long runs of identical pixel values result in high compression ratios. However, in the case of 24-bit true colour bitmaps such methods do not necessarily work as well. These true colour bitmaps are usually photographs or images requiring large colour counts and continuous tonal graduation – hence the usage of such a large colour space. Whilst photographs and similar images may still contain large blocks of similar colours, such as areas of sky, conventional lossless compression algorithms are less useful because the colours in a block are often very similar but not *the same*. This can be seen in figure 2.3.2.

figure 2.3.2: Photograph containing large areas of similar colour

Whereas RLE may obtain a 5:1 compression ratio for figure 2.3.1, it can only manage a 1.05:1 ratio for the photograph in figure 2.3.2. The compression of true-colour images, and photographs in particular, requires a fundamentally different approach to traditional lossless compression – namely lossy content-aware compression techniques.

2.3.1 JPEG Compression

The JPEG compression standard [JPEG] [MIANO, 1999] was developed by the Joint Photographic Experts Group to improve on conventional compression techniques in the compression of digital photographs. It has become one of the most commonly used compression methods and, although it is a complex algorithm and compute intensive, it can achieve compression rations approaching 20:1 for photographic bitmap images.

JPEG is a lossy compression algorithm that minimizes the impact of data loss by capitalizing on the weakness in human visual abilities. It does this through a series of discrete steps which are given below. A 640x480x24bit source image will be assumed.

Step one of JPEG utilizes the YCbCr colour model to exploit the reduced value of the two chrominance channels in comparison to the luminance channel. Once the source image has been converted to this colourspace, giving three 640x480 matrices with 8bit values, discrete blocks of 2x2 pixels in the chrominance matrices are averaged to reduce the size of each

matrix to 320x240 pixels. This is clearly a lossy step but the eye's reduce chrominance sensitivity means that this is very difficult to detect. The result is that this step compresses the data by a factor of two – we have lost ¾ of the chrominance information in two of the three matrices. The value 128 is then subtracted from each element in each matrix to place zero as the middle value in the range – i.e. matrix elements now range from -128 to +127.
Finally, each matrix is divided up into 8x8 pixel blocks, the full-size Y matrix having 4800 blocks and the reduced Cb and Cr matrices having 1200 blocks each.

Step two involves the application of a discrete cosine transformation (DCT) individually to each of the 8x8 pixel blocks. The output of each DCT application is another 8x8 matrix of DCT coefficients.

An example 8x8 block and its associated DCT coefficients presented in [MIANO, 1999] are shown in figures 2.3.1.1 and 2.3.1.2 respectively.

58	45	29	27	24	19	17	20
62	52	42	41	38	30	22	18
48	47	49	44	40	36	31	25
59	78	49	32	28	31	24	31
98	138	116	78	39	24	25	27
115	160	143	97	48	27	24	21
99	137	127	84	42	25	24	20
74	95	82	67	40	25	25	19

Figure 2.3.1.1: Original 8x8 Block Grayscale Values pre-DCT

-603	203	11	45	-30	-14	-14	-7
-108	-93	10	49	27	6	8	2
-42	-20	-6	16	17	9	3	3
56	69	7	-25	-10	-5	-2	-2
-33	-21	17	8	3	-4	-5	-3
-16	-14	8	2	-4	-2	1	1
0	-5	-6	-1	2	3	1	1
8	5	-6	-9	0	3	3	2

Figure 2.3.1.2: Result of DCT application to 8x8 matrix in figure 2.3.1.1

Though in theory, the DCT itself is lossless, the matrix presented here consists of integers and not floating points and therefore is subject to rounding errors.

When used with photographic images, where sharp changes in pixel values within a block are uncommon, the DCT concentrates the most important coefficients for reproduction of the original block in the top-left corner of the output matrix. The coefficient at (0,0) in the DCT matrix in nearly three times the next largest at (0,1). By the time matrix rows/columns 5 through to 8 are encountered, the coefficients are significantly smaller and therefore have little impact on the reconstruction of the original 8x8 block pixel values. An illustration of the concentration of value distribution across the original matrix and also after the DCT application is given in figures 2.3.1.3 and 2.3.1.4 respectively.

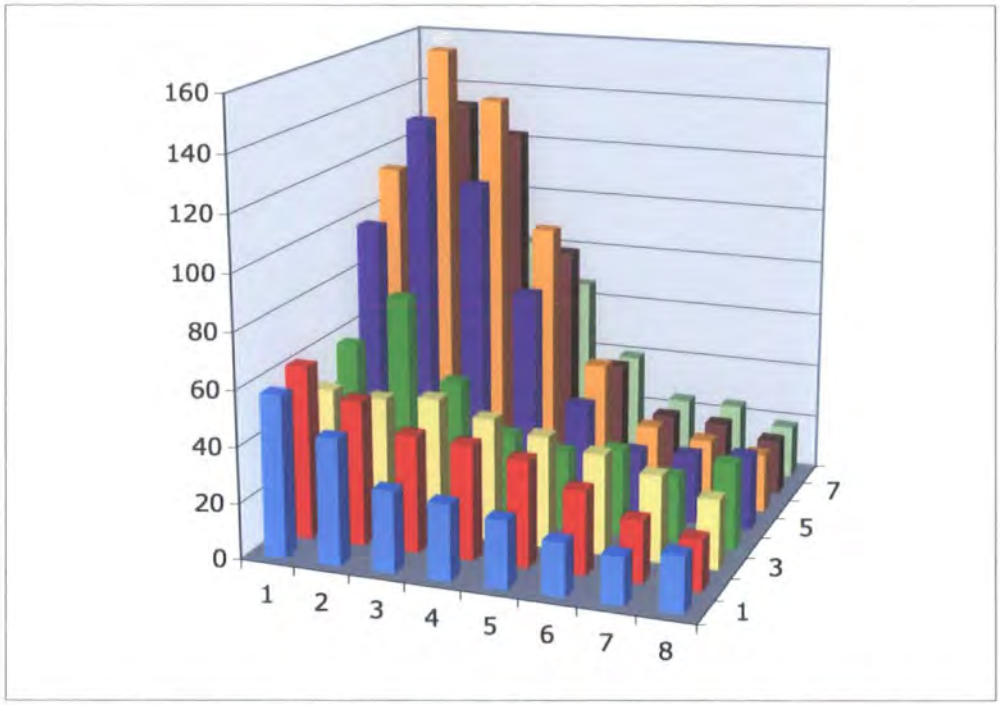


figure 2.3.1.3: Value distribution across the original 8x8 matrix

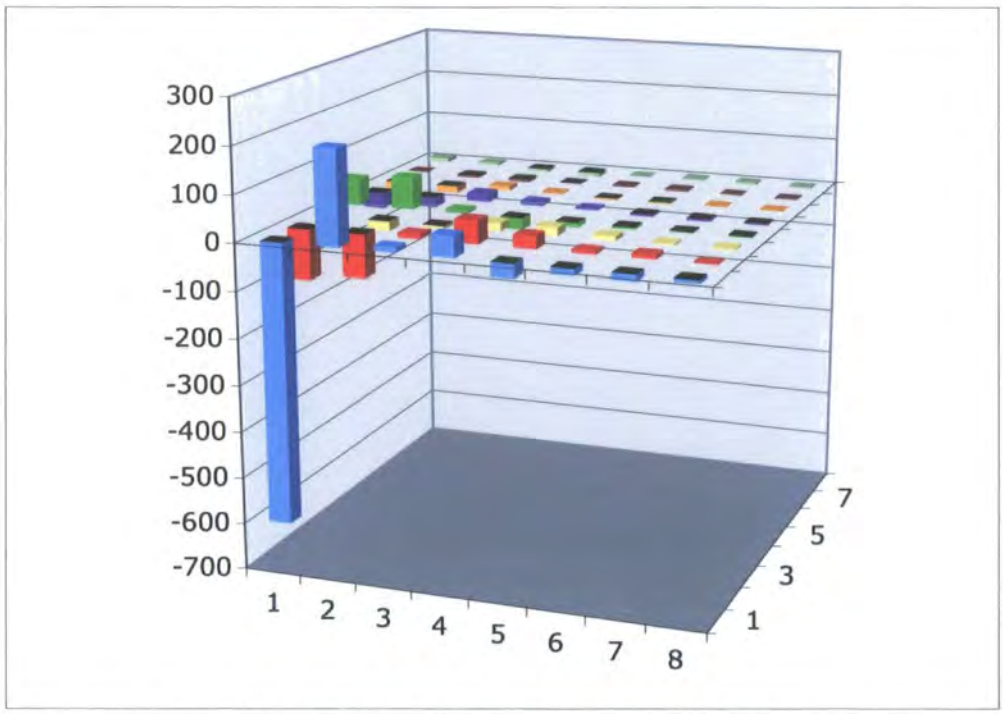


figure 2.3.1.4: Value distribution across DCT result, with values concentrated at origin

The third JPEG step, quantization, aims to nullify less important DCT coefficients. This step utilizes an 8x8 quantization matrix containing variable weights to do this. Although quantization is part of the JPEG standard, the values within the table itself are implementation specific. An example of a JPEG quantization table is given in figure 2.3.1.5.

17	18	24	47	99	99	99	99
18	21	26	66	99	99	99	99
24	26	56	99	99	99	99	99
47	66	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99

Figure 2.3.1.5: An example JPEG Quantization Table

Each element in the DCT matrix is simply divided by its corresponding element in the quantization table, before it is rounded to an integer. The quantization table doesn't vary according to the nature of a particular source bitmap image and remains constant within a specific JPEG implementation. Quantization's main function is to retain data in the top-left of the DCT matrix whilst nullifying matrix elements that are more distant from this point. It is possible to use different quantization tables for Y blocks than Cb or Cr blocks, though this will not be shown here. The result of applying the quantization table in figure 2.3.1.5 to the DCT matrix in figure 2.3.1.2, including rounding of the output, is given in figure 2.3.1.6.

-35	11	0	1	0	0	0	0
-6	-4	0	1	0	0	0	0
-2	-1	0	0	0	0	0	0
1	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Figure 2.3.1.6: Final Quantized DCT Matrix

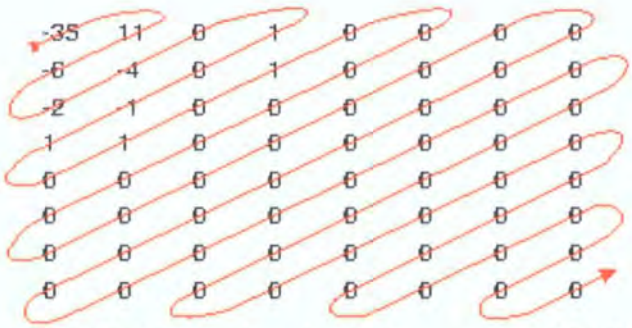


Figure 2.3.1.7: 'Zigzag' read order for quantized DCT Matrix

It can be seen that after quantization, only 10 out of 64 elements of the matrix are non-zero. Figure 2.3.1.7 indicates a zigzag order with which to read the elements of the matrix. This zigzag, the beginning of step four, aims to return a list of values where the concentration of zeros are read consecutively and can therefore be compressed using RLE. This would result in '-25, 11, -6, -2, -4, 0, 1, 2, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0…'

To decode a JPEG image, the algorithm is simply reversed. Unlike a number of image compression algorithms JPEG is roughly symmetric and requires a similar amount of computation to decode as it does to encode.

JPEG is a complicated compression algorithm but gives excellent compression ratios. The level of compression applied can be varied by adjusting the weights within the quantization table to be more or less aggressive in terms of the loss of DCT coefficients.

2.4 Fractals

A fractal is a shape or image that is defined by an iterated mathematical equation, where the output from the equation is fed back as the input. [GLEICK, 1988] The fractal is a set within a (subset of) space. There are a wide variety of fractal types, each with its own properties and applications. Many exhibit the following characteristics, which together provide a better definition of the term 'fractal'.

2.4.1 Preservation of Complexity

When magnifying part of a fractal image, the resulting image will not have lost any resolution or any of its original complexity. Some fractals, such as the Mandelbrot set, allow continuous magnification ad infinitum, with new detail and complexity appearing at every level. The Mandelbrot set is infinitely complex, with the detail seen in successive magnifications appearing through the increased rendering detail. [BARNSLEY, 1988] [FISHER, 1995] An example of this property is shown below:

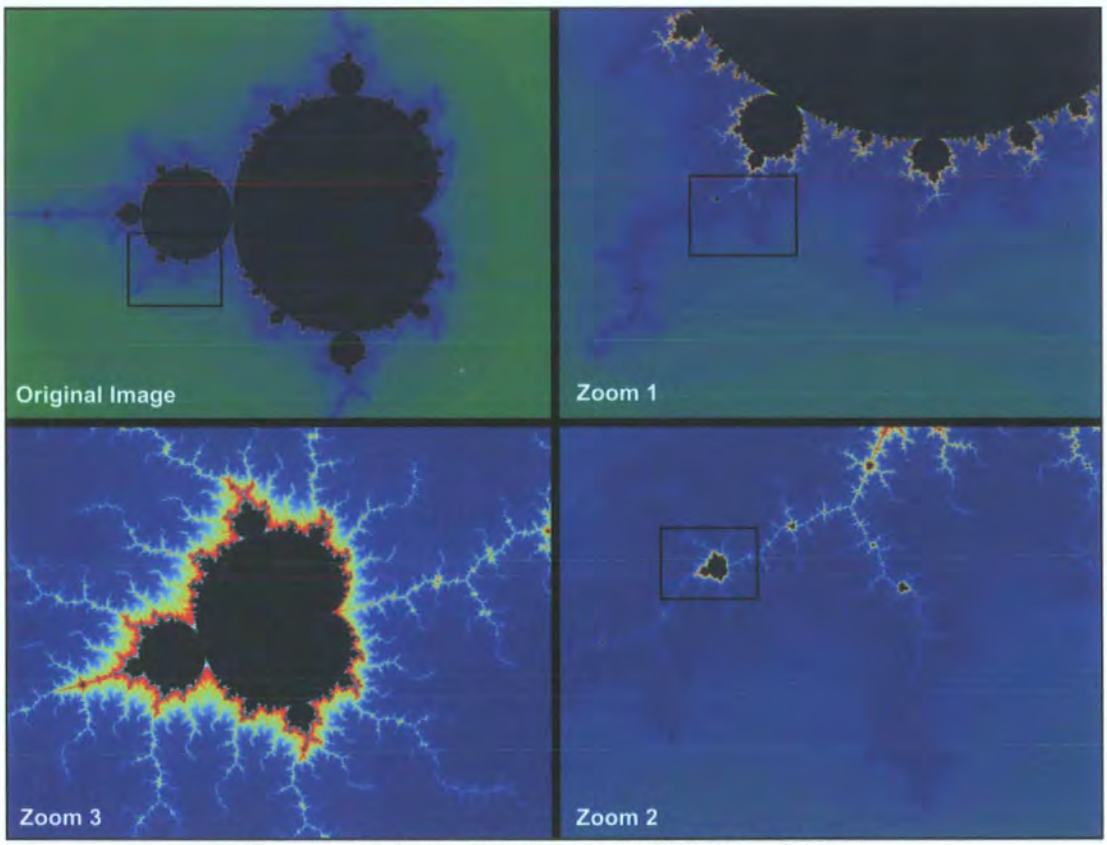


figure 2.4.1.1 - successive Mandelbrot magnifications

The images in figure 2.4.1.1 illustrate the initial set, and successive enlargements to produce the next three images. The image area to be enlarged for the next image is indicated by a black rectangle. Further magnification is possible ad infinitum.

2.4.2 Self-Similarity

Fractals often appear to be constructed of replicas, or near replicas of themselves, with each replica being distorted in certain ways. This is known as the self-similarity property. The images of the Mandelbrot set in figures 2.4.1.1 show that at each magnification level the image has a similar 'bug'-like black mass. The Mandelbrot set is self-similar in that these masses can be found at all levels, but each mass is slightly different in shape and size. A variation on this, but not as common, is the self-tiling property. A fractal exhibiting this property is constructed entirely of smaller, distorted copies of itself. A classic example of this is the Barnsley Fern [BARNSLEY, 1988]

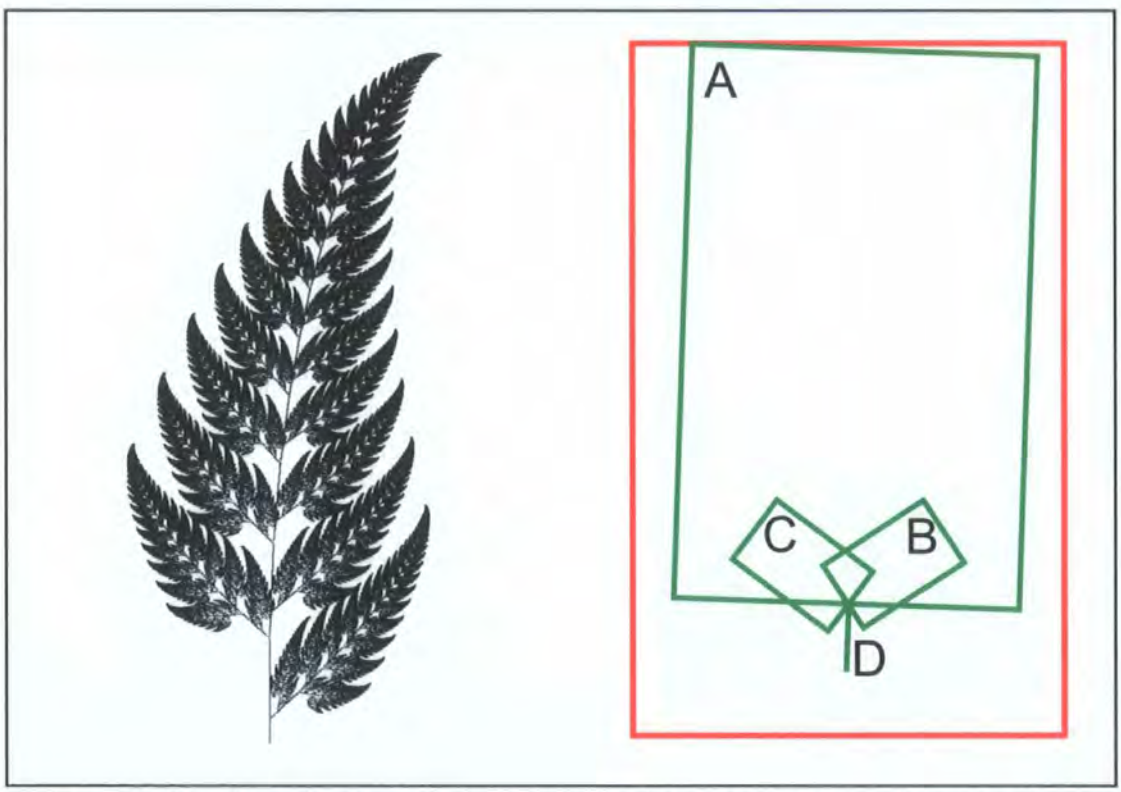


figure 2.4.2.1 – Barnsley Fern, and right, showing the branches as transformed replicas

Figure 2.4.2.1 illustrates the construction of the Barnsley Fern. Each of the green rectangles represents a transformation of the image. The leaves are the most obvious example of this, and are mapped by transformations B and C. Each leaf is essentially a copy of the upright fern that has been reduced in size and rotated to 'attach' the stem. The stem of the fern is constructed in the same way, although this is not as immediately obvious as the leaf self-

tiling. In transformation D, stem construction involves mapping the image so that it rotated sideways and mapped to give a thin line. As these transformations will only produce a short stem and two leaves per iteration, transformation A maps the previous-iteration's generated image to form the top of the fern. The Barnsley fern belongs to a class of fractals known as Iteration Function Systems (IFS). [BARNSLEY, 1988]

2.4.3 Iterative formation

Fractal images are usually produced according to a certain mathematical formula. This formula is iterated many times. The formula to generate the Mandelbrot set, as in figure 2.4.1.1, is as follows:

$$z \rightarrow z^2 + c \qquad \textit{(where z and c are complex)}$$

The above formula is used to generate another type of fractal, known as Julia sets. Varying values of *z* and *c* give different sets. The Mandelbrot Set is often referred to as the set of all Julia sets. [GLEICK, 1988]

2.4.4 Iterated Function Systems

A specific class of fractals, known as Iterated Function Systems, have the property that they can represent any image. An IFS consists of a set of 2D transformations which specify the self-tiling properties of an image, such as the Barnsley Fern. These transformations are applied iteratively to form an image constructed of repeated automated tiles of itself. All IFSs are self-tiling. [BARNSLEY, 1988]

The IFS function is a group of *n* sets of 6 coefficients, where *n* is the number of transformations. The image generated by iterated application of the transformations is known as the attractor of the IFS. The attractor is reached after a certain number of iterations of the generation algorithm. The attractor of an IFS is the image that does not change even if more iterations of the formula are performed. Theoretically, the attractor is the fixed image of the

IFS at an infinite number of iterations. Using a bitmapped computer display and a finite data structure means the attractor, at the given resolution, of the IFS can be determined by the image remaining fixed at the resolution of the storage / display. The number of iterations required to render the image for a given resolution of an IFS is finite, as once the attractor is reached then there will be no further change in the rendered image. [FISHER, 1995]

Apart from the properties of self-tiling and iterative formation, IFSs also exhibit preservation of complexity. As the attractor of an IFS is magnified, the user is able to see the self-repetition continuing down through the image. The more the attractor is magnified, the more of the same transformations are revealed.

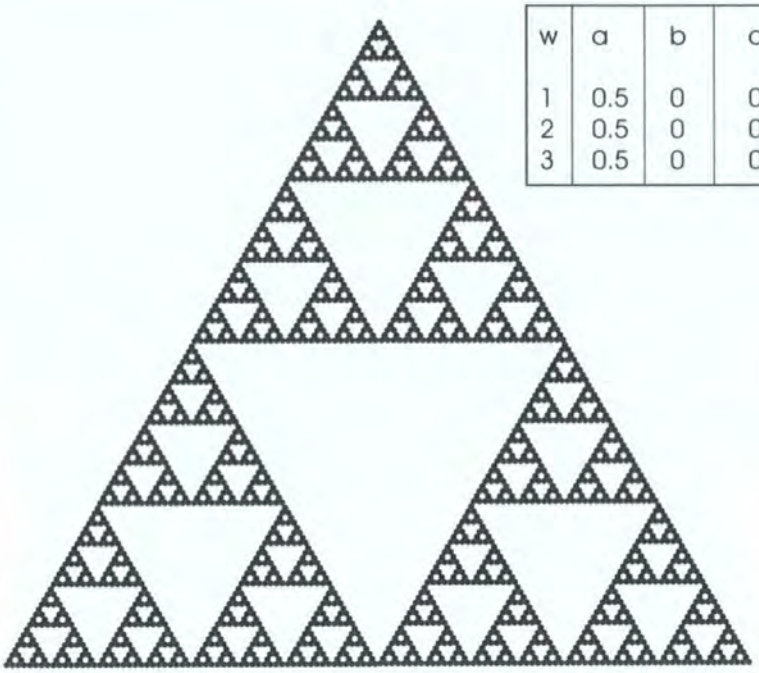
An IFS is specified as a series of contractive mappings. A contractive mapping is a mapping of the source image through a series of transformations (such as scaling, translation, rotation). These transformations are performed on a metric plane, in the case of a computer this usually means the main bitmap display. The mappings are contractive because when the transformation is applied, the points on the plane are brought closer together. Such transformations are known as 2D affine transformations. An IFS is usually specified using the following format:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} e \\ f \end{pmatrix}$$

This represents the tranformation of a point $w(x,y)$, where (x,y) represents the original point and (x',y') is the position of the newly transformed point. [BARNSLEY, 1988]
This notation is the same as the expression: $w(x',y') = (ax + by + e, cx + dy + f)$

2.4.4.1 An Example of an IFS and its Construction [FISHER, 1995]

A classic example of an IFS is the Sierpinski Gasket. It is widely used to illustrate the basic theory of Iterated Function Systems, and its attractor can be calculated manually by hand in a few simple steps. The attractor and IFS transformations are given in figure 2.4.4.1.



w	a	b	c	d	e	f
1	0.5	0	0	0.5	0	0
2	0.5	0	0	0.5	0.5	0
3	0.5	0	0	0.5	0.25	0.5

Figure 2.4.4.1: The Sierpinski Gasket

As discussed previously, an IFS consists of a series transformations on the plane. There are three transformations required to produce the Sierpinski Gasket. The formation of the attractor of this IFS can be illustrated by performing the transformations step-by-step, and plotting the image generated at each stage. Figure 2.4.4.2 shows the attractor being constructed in this fashion.

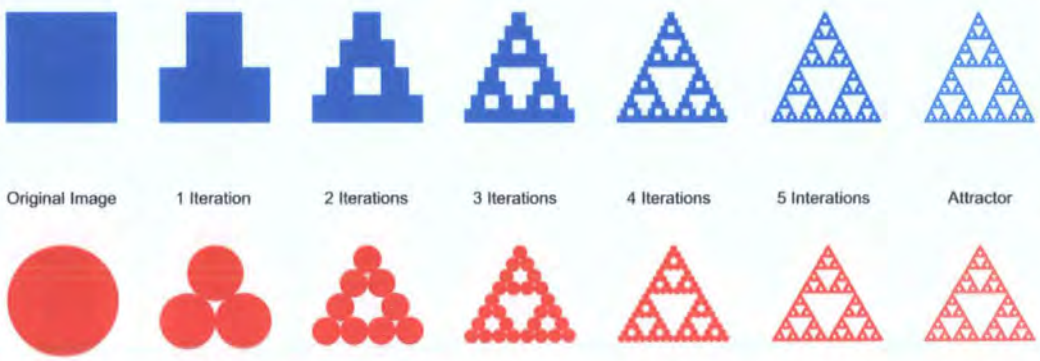


Figure 2.4.4.2 – Step-by-step construction of the Sierpinski Gasket

The construction method for the gasket could be described in simple English-like steps:

- Take the input image, scale it by a factor of 0.5
- Map the resulting image at position 0,0 on the plane
- Map the image again at position (0,0.5) on the plane
- Map the image again at position (0.5,0.25) on the plane

This simple series of iterated steps can produce an image of infinite detail. IFS theory dictates that whatever the starting image is, the resulting attractor will be same, as the attractor is only mathematically complete after infinite iterations when all the detail of the starting image would be infinitely reduced.

Figure 2.4.4.2 also illustrates another property of IFSs and their attractors. The appearance of the attractor of an IFS is not dependent on the appearance of the initial starting image. The image of the square and the image of the circle both result in the same attractor. This is due to the contractive mappings within the IFS, where points are successively brought closer together with each iteration.

Notice that the images of IFSs presented are monochrome. Colour IFSs [FISHER, 1995] rely on colouring the individual pixels according to how much iteration is needed to position them within the attractor – or to 'show' themselves on the attractor, but these are beyond the scope of this project.

2.4.4.2 Algorithms to Plot IFS Attractors

The mathematics of generating IFSs is relatively straightforward. In order to make use of IFS within many systems, it must be posible to generate the attractor of the IFS. There are numerous algorithms that enable rendering of IFS attractors, each with their own advantages

and disadvantages. Computers are well suited to the repetitive task of generating IFS attractors, which are usually rendered to a two-dimensional array or bitmapped display. Whilst IFSs are continuous functions with infinite detail, computer bitmapped displays are not, and the rendering algorithm must take account of this in order to render the attractor at the display device resolution. This avoids incomplete attractors or wasted iterations trying to add detail beyond device resolution, where the IFS output becomes invariant and has reached the attractor. The three most common generation algorithms are considered in the following sections.

2.4.4.2.1 Classic Deterministic Algorithm – CDA

CDA uses the same method as illustrated above in the 'hand-rendering' *(figure 3.3.3)* of the Sierpinski Triangle. Once the IFS reaches its attractor, any further mappings have no effect. The algorithm performs the transformations on the original image to reach iteration 1. It then takes this 'output' image and feeds it back as the initial image for the algorithm. This continues until the attractor of the IFS is reached – i.e. the output is the same as the input. CDA is a deterministic algorithm. It is easy to predict the number of iterations required to generate the attractor. The fact that the attractor is generated in an ordered fashion is one of the main advantages of this algorithm. Unfortunately, it is quite slow and inefficient compared to some other IFS rendering algorithms. [BARNSLEY, 1988]

2.4.4.2.2 Random Iteration Algorithm – RIA

The Random Iteration Algorithm is stochastic, unlike CDA. The algorithm uses a probability assigned to each mapping in the IFS. The probabilities of all the mappings must sum to one, and are used to determine how often the mapping should be selected. The algorithm picks a point on the attractor and a selected mapping is applied to this point to give a new point on the attractor. The mapping used is chosen using the set of probabilities. The algorithm then continues in this fashion for a predefined number of iterations. If enough iterations are specified by the user then a good approximation of the IFS attractor can be produced. [BARNSLEY, 1988] [BARNSLEY, 1993]

The pseudo-code for the algorithm is given below:

```
Select an initial point p on the attractor;
      For i=1 to number of iterations;
      Plot p;
      Select a mapping m, using probabilities;
      p=m(p);
Next i.
```

The Random Iteration Algorithm is popular within IFS applications because it provides an approximate image of the attractor very quickly. In order to provide a detailed image, though, the algorithm must perform a much greater number of iterations, as it will often randomly select the same point many times – therefore recalculating and plotting a previously determined point, which is wasteful. Dense attractors may require a massive number of iterations to render them completely. Because points are plotted by being randomly selected and then transformed, it is possible to use the RIA to produce coloured attractors of the IFS. [FISHER, 1995]

Weighted probabilities are used within the algorithm to enable more even point coverage. Mappings of high contractivity – mappings with small scaling factors – map points onto a relatively small proportion of the overall attractor. Mappings of low contractivity do not usually affect the position of the transformed point greatly. If the high contractivity mappings were picked in an equal ratio to the small contractivity mappings, then the final attractor would have inadequate coverage on the main body of the attractor whilst the area of contractivity would be rendered in much more detail. The algorithm would spend a disproportionate amount of time rendering this small area. The algorithm selects the mappings according to their individual probabilities to avoid this.

[BARNSLEY and HURD, 1993] propose the following to calculate the probabilities for an IFS:

$$p_i \approx \frac{\text{deter } M_i}{\sum_{j=1}^{n} \text{deter } M}$$

This formula calculates p_i - the approximate probability for mapping i; n is the number of contraction mappings comprising the IFS; and deter M_i is the determinant of the matrix M of the mapping i. The value of deter M_i corresponds to the factor by which the original area is reduced onto the contracted area. It is effectively the level of scaling of the mapping.

2.4.4.2.3 Minimum Plotting Algorithm – MPA

This algorithm aims to render a complete image of the attractor by rendering as few pixels as possible. The algorithm takes advantage of the property that applying the same contraction mapping to the same point will always give the same resulting point. Thus, this does not need to be repeated.

In order to avoid plotting points more than once and not repeating calculations, MPA keeps track of the points it has already plotted using a FIFO queue. This ensures that a point is never revisited. The algorithm can plot directly to a display or to a 2D array.

The algorithm renders the attractor by firstly plotting some initial points known to lie on the attractor. As points are generated they enter the queue of points to be transformed. This takes advantage of the property that transorming a point on the attractor will result in a point also on the attractor. The algorithm takes a point from the head of the FIFO queue and checks if it has been plotted. If it has it is discarded, otherwise the point is transformed to give a new point for the queue and also a new point to be plotted. The algorithm terminates when the FIFO queue is empty. [MONRO, 1990]

```
Initialise initial points on attractor;
Plot initial points;
Store points in FIFO queue Q;
```

```
Repeat;
        Take p from head of Q
        For every mapping m;
                p' = m(p);
                If p' has not been plotted;
                        Plot p';
                        Add p' to Q;
                EndIF;
        Next m;
Until Q empty.
```

The MPA is very efficient as it stops redundant calculation being performed. The disadvantage of the algorithm is that it needs a FIFO queue to be set up in memory, which can be huge and impractical for the generation of large attractors.

2.4.4.3 Using IFSs for Image Compression

Fractals have a number of real-world uses. These include modelling chemical reaction kinetics, growth patterns in bacterium and other simple organisms, and weather forecasting [GLEICK, 1988]. Image compression, however, is perhaps the most important and powerful application of fractal techniques.

It can be seen that highly detailed images can be formed from the iteration of simple mathematical formulae. The Barnsley Fern attractor in figure 2.4.2.1 occupies approx 400,000 bytes in 4-bit greyscale, whereas the coefficients used to produce it require less than 100 bytes. Storing the coefficients instead of the bitmap results in a high level of 'compression', giving rise to the concept of using fractals for image compression: find an IFS that generates the original image as its attractor, and thus encode that image as a series of IFS transformations.

2.4.5 The Inverse Problem

An attractor can be generated from an IFS using a relatively simple process. Barnsley [BARNSLEY, 1988] conceived the idea of trying to find the IFS for a given image, as opposed to trying to produce the image from a given IFS. The problem of determining the number of transformations and their associated coefficients is known as the inverse problem. There have been numerous attempts to solve this problem with an automatic solution – in that no human interaction/intervention is required to reach the solution. Interactive solutions, which are not classed as automatic, use human input to help restrict the search space. The reason for the difficulty is the size of the search space. Even if 4 transformations are required per image, that still leaves 24 coefficients to test with values between approx. –0.7 and +0.7 [BARNSLEY, 1998], typically calculated to 3 decimal places. This gives a search space of the magnitude of ~2×10^{1932}

In reality, the number of coefficients for a complex real-world image would be much larger as is unlikely that such images would display such large and simple amounts of self-similarity. It is impractical to explore this searchspace this using trial and error techniques, as even the most power computers would take tens of years to explore it. When attempting to find the IFS for a given image, there must be a compromise between image quality and compression time / amount. This trade-off is similar to that experienced with JPEG compression [TANENBAUM, 1996].

The inverse problem has been solved in theory. The solution is to produce an IFS that contains a mapping for every pixel in the image. Each mapping then is a transformation of the original image to a pixel. This is clearly an unsatisfactory solution, as the mappings occupy a far larger space than the original image, giving expansion rather than compression. The true aim of a solution to the inverse problem is to determine the minimum number of mappings required to represent the said image (and of course, their associated coefficients), within a certain accuracy range. The solution should consider:

- How can the minimum number of individual mappings required be determined?
- Are all images suitable for compression using this method, and if not how are they distinguished?
- How can the quality of the approximation of target image be determined?

The first point is crucial to the problem, and stems from the need to gain compression over the map per pixel theory solution. Only by reducing the number of required mappings can compression be gained. The second point highlights the fact that different images compress with varying degrees of success. Since IFSs exploit the ideas of self-similarity, images exhibiting this property, such as the Barnsley Fern, will lend themselves to the techniques more than less self-similar images. The final point is important because the algorithm has to be able to assess how good the solution it has produced is, and whether it represents an acceptable representation of the target image.

A feature of IFSs, known as the Robustness Property [NETTLETON, 1994], dictates that a small change in the coefficients specifying an IFS gives a small change in the final attractor. This is a very useful property because it allows for gradual fine-tuning of the match using progressively finer adjustments until the desired quality threshold is achieved. There are various methods for testing how good a match the IFS attractors in comparison to the original image. It is, therefore, essential that a reasonable fitness algorithm be employed in order to allow fine-tuning of solutions. The Hausdorff Metric [BARNSLEY, 1988] returns a value that gives the 'distance' between two images – that is, how similar they are. This method of comparison is very accurate but unfortunately it adds a large level of overhead to the exploration of the search space. As a result, simpler but less accurate methods are often used.

The simplest method would be to check each image point-for-point, and return a percentage according to the number of correctly placed points. A problem with this method is that if the attractor is very close to the solution it may still be at a slight offset to the original image, causing a low point-for-point match and the matching algorithm to return a very bad match

result as the images are not quite aligned. This would not only waste further machine time, but could destroy a near-optimal solution. This is illustrated in figure 2.4.5.1.

figure 2.4.5.1 – Weakness of simple image-comparison metrics for IFS searches

In figure 2.4.5.1, the trial solution is very close to the target image, i.e. it is near-optimal, with a simply x,y transform required to provide a perfect match. However, the yellow pixels in the third image are the only correctly-mapped pixels from the point-to-point comparison metric's perspective and this trial solution is accordingly assigned a very low fitness value.

A variation on the point-for-point method is collage point-coverage, where the image is split into subsections, which then each have their points considered in relation to the corresponding section on the target image by the point-to-point function and scored accordingly. The scores are assessed to give a final score for the image match. Other methods range in accuracy and computation up to the Hausdorff metric. Again, a compromise must be reached.

Figure 2.4.5.2 is a cross-section of the search space for the Sierpinski triangle [NETTLETON, 1994]. The fitness function used is collage point coverage. In order to restrict the cross-section to three dimensions, only 2 of the 18 coefficients are varied (18 = 6 x 3 mappings), these being the *e* and *f* values for the third mapping, giving the *x* and *y* axis. The remaining 16 coefficients are fixed at their optimal values. The *z* axis gives the fitness of the IFS attractor to the target image.

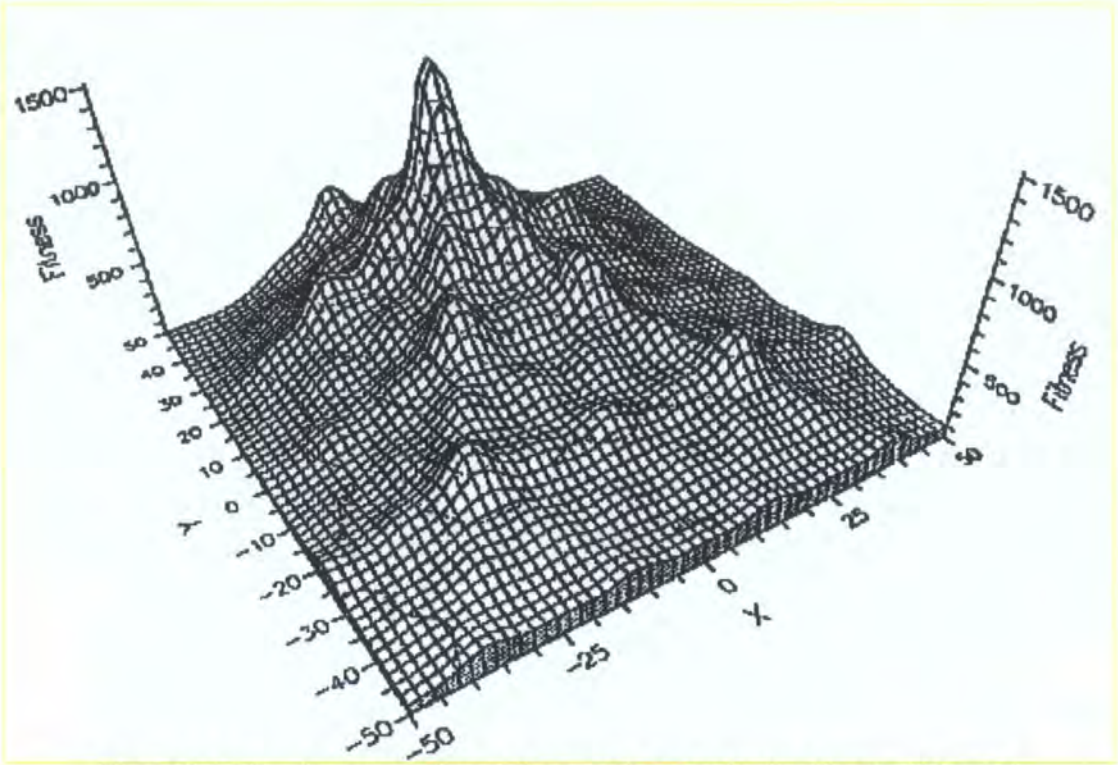


Figure 2.4.5.2 – A cross-section of the search space for the Sierpinski Gasket

Figure 2.4.5.2 illustrates just how complicated the search space of an IFS is. The largest peak with a fitness of ~1500 represents the solution. There are, however, many other smaller peaks that lead to nothing. An algorithm that searches such a space successfully will need to be able to disambiguate between these 'misleading' local maxima in order to find the true or best solution. Traditional search techniques, such as hill-climbing algorithms, tend to get stuck within these local maxima, and cannot generally be employed to solve the inverse problem – although they may form *part* of a theoretical solution. [NETTLETON, 1994]

2.4.6 Automatic Algorithms to 'Solve' the Inverse Problem

Automatic solutions to the inverse problem aim to achieve a certain accuracy range whilst making as few comparisons and explorations as possible. Because these algorithms can explore only a tiny fraction of the search space, none can provide an optimal solution for every single instance – that is, they do not solve the general case. Most of the techniques take advantage of particular properties, which whilst providing optimisation is terms of both speed and accuracy, restricts their range of application.

2.4.6.1 Evolutionary Algorithms

Evolutionary Algorithms are algorithms that try to mimic the natural process of evolution – selecting the best solution from a population using natural selection [BEASLEY, 1998]. Such algorithms use computational models of some of the well-known processes of natural evolution in both their design and implementation. EAs have a population of potential solutions within their environment, which evolve through the generations with the hope of finding an acceptable solution. Evolution is carried out according to 'survival of the fittest' – the better solutions are evolved, hopefully into better ones, whilst less optimal solutions will be dropped. Eventually, there should be a convergence towards the optimal solution. The following features are essential to the construction of an EA:

- **Fitness function** – Evaluates how good the current solution is in a quantitative manner
- **Retention of quality** – There must be a mechanism which allows good solutions to be retained into subsequent generations, whilst poor solutions are removed from the pool.
- **Regeneration** – In order for the solutions to 'evolve', there must be a mechanism that alters the solutions in order to provide the next generation. These are created by applying specific operators to the current solutions. In the case of mutation it is typically proportional to the fitness of the individual solution. If the solution is reasonable, then it would be mutated less than a poor solution. This approach aims to fine tune good solutions whilst still varying bad ones in order to find new possibilities.

EAs can be used as an automatic solution to the inverse problem if the solution to is the IFS for the image to be encoded; the environment is the display/memory in which the attractors are tested for fitness, and the elements a group of randomly initialised IFSs.
Evolutionary Algorithm is an umbrella term used to classify all algorithms of this nature, the two main types being Genetic Algorithms and Evolutionary Programming.

2.4.6.2 Genetic Algorithms

A Genetic Algorithm derives its behaviour from some of the well-known natural evolution mechanisms. The population of solutions emulates the idea of chromosomes, encoded as binary strings. GAs rely on a principle known as the Building Block Hypothesis, which states that by combining parts from numerous different solutions, the high fitness parts will be preserved throughout the evolutionary process and hopefully merge with others towards a complete, optimal solution. [HOLLAND 1975] [BEASLEY, 1998]

The process of merging features of the different solutions into newer child solutions is known as crossover. In addition to this, GAs also employ a mutation operator. When applied to an individual solution, the mutation operator alters it slightly to produce a new child solution. While within GAs the mutation function is invoked very rarely, it is important as it aims to prevent the loss of important genetic information, which may be essential to a near-optimal solution.

Mutation essentially ensures that the algorithm doesn't become too focused on performing crossover on the current population by allowing it to occasionally introduce new solution structures into the set. Because of the method of encoding of the solutions within the population, the operators such as crossover and mutation can be implemented using simple bit-pattern manipulation operations. Because of the way GAs build their solutions by adding blocks from previous parent solutions, they are classified as bottom-up solutions.

2.4.6.3 Evolutionary Programming

Whereas GAs use a bottom-up approach, Evolutionary Programming adopts a top-down mechanism. EP differs from GAs in that it does not employ a crossover operator, but instead makes greater use of the mutation operator. A typical EP will evaluate the performance of each of the current set of solutions and attempt to produce improved child solutions by performing mutation upon the parent solutions relative to the fitness of the individual solution determined by the fitness function. The population experiences a probabilistic removal of

solutions between the generations, with the lower fitness solutions being most likely to be removed. This process ensures that poor performing solutions are unlikely to survive though the generations whilst mutation allows for fitness proportional 'tuning' and introduction of different solutions into the population. [BEASLEY, 1998] [FOGEL, 1966]

2.4.6.4 Comparison of Evolutionary Algorithms

Nettleton showed that Evolutionary Programming performed consistently better than Genetic Algorithms when used with the same starting population size and number of generations of the algorithms, in fact EP produced near optimal results. The image used to test these systems on was a solid black equilateral triangle – extremely simple. From this result, Nettleton concluded that a top-down approach to solving the Inverse problem might be the best way to tackle it. The results achieved using the attractor and point coverage function as fitness are shown below in *figure 3.6.1,* with a range of generations of 0, 5, 10, 20, 30, 40, 60, 80 and 100 from top left to bottom right for each EA type:

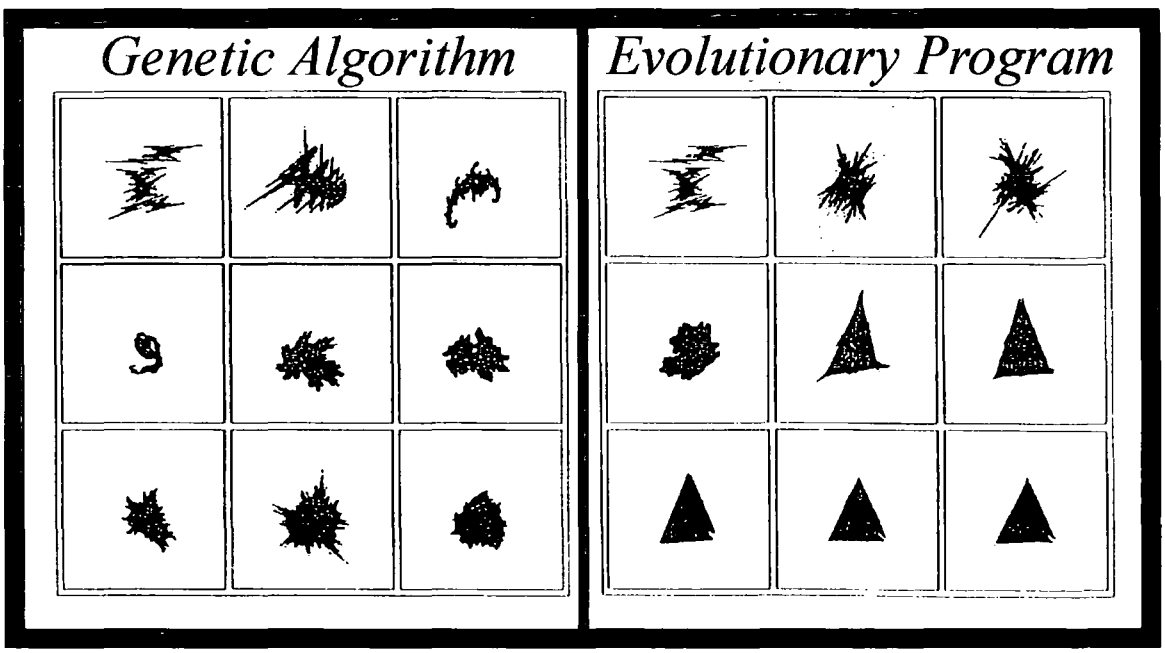


Figure 2.4.6.4.1 – Comparison of GA vs. EP in the automatic solution of the inverse problem

Using EAs, and in particular EP has had a degree of success in the automated solution of the inverse problem, the main areas of success have been with simple geometric shapes, such

as the solid equilateral triangle. Less acceptable results have been experienced with more complex shapes, such as the Sierpinski Triangle. [Nettleton, 1994]

2.4.7 Using Quadtree Partitioning for Image Compression

The IFS method, although offering the potential of fantastic compression ratios and relatively-rapid decode, is crippled by the difficulty of addressing the Inverse Problem and discovering the mappings required to represent a target image. These failings illustrate the requirement for an image compression algorithm to be deterministic – although it acceptable that it is often difficult to predict the compression ratio that will be achieved by an algorithm for a particular image, it is very important that we maintain an adequate image quality level and that the compression can always deliver this in a reasonable amount of computing time.

Quadtree-based Fractal image compression aims to address the non-deterministic nature of IFS compression techniques, though as a result the compression ratios it achieves are much more conservative than the theoretical IFS levels. However, quadtree compression can be applied to any image and sophisticated implementations deliver excellent compression ratios.

As with IFS, these compression techniques still rely on finding a series of contractive mappings to represent a target bitmap image. Basic quadtree techniques divide the target image, f, up into 16 squares, known as Ranges, and also divides f into 4 squares, known as Domains. The algorithm then attempts to cover every range with a domain, using a contractive mapping, where the similarity (or fitness) of the range/domain map is maximised. If the fitness of this mapping doesn't reach a preset threshold, the algorithm then treats the uncovered range as a new image and continues the factor-of-four subdivision. This gives rise to a quadtree representation of the image, as shown in *figure 2.4.7.1.*

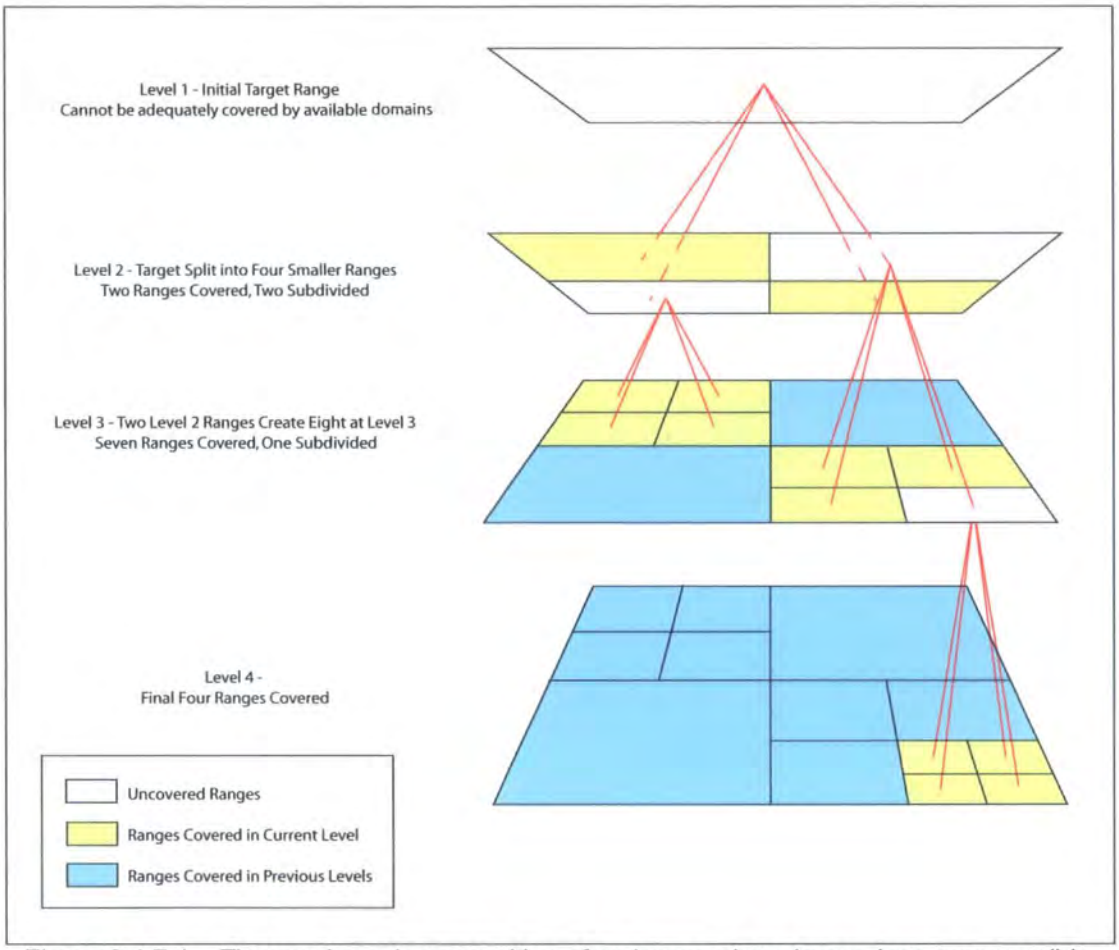


Figure 2.4.7.1 – The quadtree decomposition of an image where larger, less-compressible, ranges are sub-divided into smaller ranges at each level of the tree.

If the required fitness is still not achieved after a present number of subdivisions then the optimal fitness map(s) are used to cover that particular range. As each range is covered, the mapping used to transform the domain onto it is recorded. [FISHER, 1995]

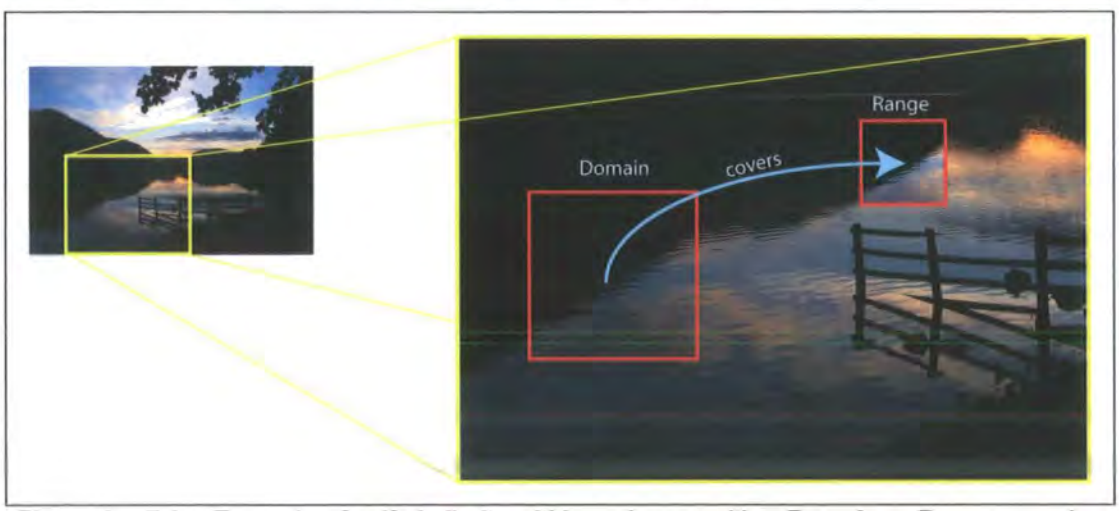


Figure 2.4.7.2 – Example of self-similarity within an image with a Domain to Range mapping

Figure 2.4.7.2 illustrates how a relatively small area of sample image (a range) can be recreated by mapping a larger area onto it (a domain). As part of this transformation, the domain is resized such that its dimensions are halved. Images are compressed in the quadtree system by finding suitable domain to range mappings that together cover the entire image.

The pseudo-code for the quadtree compression method is shown below:

Set a fitness tolerance level, t

Determine the set of ranges, $\mathbf{R}$, to be covered, according to the image size and range dimensions

Populate the domain pool, $\mathbf{D}$, according to image size and range dimensions

While there are uncovered ranges, R_i, do {

Search the domain pool, $\mathbf{D}$, to find the domain, D_i, where the fitness, F_{Di}, is maximized and greater than t (i.e. the best domain match is found for the range that meets our quality threshold)

Record the mapping M_i, specifying the D_i and R_i

If no domain adequately covers R_i, divide R_i into 4 new smaller ranges and add to $\mathbf{R}$

}

Quadtree compressed images also exhibit other fractal properties, specifically the ability to render a stored image at a higher resolution than the original and have the rendering algorithm 'scale' the image by the production of artificial details. Whilst this scaling technique doesn't provide extra data that is true to the original image's scene, e.g. a digital photograph will not contain any more real detail, the technique is useful in digital imaging because the artificial detail that's produced during this process is more aesthetically pleasing than other scaling techniques, such as bilinear interpolation [FISHER, 1995]. A commercial product that makes use of quadtree image scaling is www.lizardtech.com's Genuine Fractals.

2.5 Video Compression

Digital motion video adds an extra dimension to the image compression problem. In its simplest form, ignoring any audio encoding, digital video comprises a sequence of still images that when displayed in rapid succession allows the viewer/user to experience fluid motion video. The typical frame rate for encoding PAL video signals is 25fps [TANENBAUM, 1996]. Resolution of encoded video tends to vary enormously, depending upon the application but digital broadcast television and DVD video range between 320x240 and 768x584 pixels in true-colour. [TANENBAUM, 1996]

2.5.1 The Challenges Of Digital Video

With high resolutions, colour depths and frame rates, it is clear that storing digital video in an uncompressed state requires vast amounts of storage, – a DVD-quality video sequence would require 768 pixels x 584 pixels x 24 bit colour x 25 frames per second, i.e. just over 32 megabytes per second. Without the use of image compression, manipulation and storage of digital video sequences would require vast amounts of storage and would be impractical for distribution and widespread usage. Uncompressed, a 9GB dual-layer DVD disc would only be able to store 4.7 minutes of video at this resolution.

Just as with digital imaging, large file sizes present storage and transmission problems. However, due to the added dimension of time, the speed at which digital video can be transmitted or accessed from storage is also very important, as the videostream needs to be replayed in real-time in order to be acceptable to the viewer. The bandwidth available through a transmission medium (either conventional I/O or through a transmission network) must be taken into account and is often as critical as storage volumes in the field of digital video.

It is possible to reduce the storage requirements of digital video by simply changing the sampling of the original analogue source – reducing the frame-rate, colour-depth or resolution will result in a reduction in storage requirements for a given video sequence. However, reducing each of these parameters will result in a degradation of image quality – high frame-rates are essential for smooth video; 24-bit colour depth is the standard for any broadcast video encoding; resolution needs to be sufficiently high to accurately represent the quality of the original source. It is clear that just as we can use specific lossy compression algorithms to address the problems of digital imaging, we need to do the same when it comes to digital video.

2.5.2 Spatial and Temporal Compression

Although there are a wide variety of video compression algorithms, all video compression techniques rely on a mixture of temporal and spatial compression. Spatial compression is used by JPEG to reduce the amount of spatial redundancy in a picture, for this JPEG employs its DCT algorithm. [WATKINSON, 2001]

Spatial compression refers to the compression of an individual bitmap and is equally valid in both image and video compression – this explains why there is often a great deal of overlap between image compression and video compression algorithms and techniques. An example of a purely spatial video compression technique is Motion-JPEG (M-JPEG/MJPEG) and this system is discussed below.

More sophisticated image compression techniques also employ temporal compression to further improve on the compression levels achieved by purely spatial systems. Temporal compression involves analyzing and eliminating inter-frame redundancy as well as the intra-frame redundancy work of the spatial algorithm. Temporal compression often compares a frame with its precursor and attempts to store only the changes or differences between then rather than just storing the whole of the frame using spatial compression. [WATKINSON, 2001]

The Motion Pictures Expert Group "1" video compression algorithm (MPEG1) uses both spatial and temporal compression and is discussed below.

2.5.3 Motion-JPEG Video Compression (M-JPEG)

Modern video compression techniques often borrow extensively from image compression algorithms – the M-JPEG video compression algorithm uses the JPEG image compression standard and applies it to the long sequences of bitmap elements that make up digital video. As such, it is an excellent example of this image compression reuse technique. [TANENBAUM, 1996]

Although M-JPEG only uses spatial compression on the digital video sequence, and therefore only achieves similar compression rations to standard JPEG data compression levels, the standard is still widely used for video compression as it provides moderate compression levels whilst providing the user with relatively fast encode and decode times compared to temporal algorithms. [TANENBAUM, 1996]

M-JPEG remains popular due to the different performance requirements of the two typical usage models for digital video – typically, digital video is either being edited, re-sequenced, etc., before being finalised for storage; or the video is being viewed/transmitted to the end user. These two different usage profiles have very different requirements in terms of performance of the compression algorithm used. If a video is constantly being accessed,

especially with frequent access to differing points in a sequence or moving blocks of video within a sequence, then M-JPEG is better suited to than temporal algorithms as each frame is encoded independently of the other, allowing random access within the frame-sequence without having to refer to previous frames in order to construct the one the editor is interested in. If, however, the video is in its final state and ready for viewing/distribution then a higher compression temporal algorithm can be used as the main consideration will be the final size of the video sequence – random access, apart from relatively small levels of rewind or fast-forwarding, is not a major consideration – and such an algorithm would be preferred to M-JPEG. [WATKINSON, 2001]

2.5.4 MPEG Video Compression

The majority of video compression effort and research takes place to satisfy the usage characteristics of the viewing/distribution model rather than the editing model. There are a number of reasons for this:

a) Commercial video tends be 'edited once, viewed many' in that once the video, typically a television programme or movie, has had its editing completed, it is then rarely re-edited by the end user – rather it is simply viewed repeatedly.

b) Video editors tend to have access to large and high performance storage resources whilst the editing takes place. As the emphasis is on maintaining quality and access performance rather than saving storage space during this process, there is little demand for high compression algorithms in this area.

c) The distribution of finished commercial video typically uses relatively low capacity media (such as CD and DVD) or transmission links. This is due to the economics and practicalities of the distribution process: the distribution medium has to be low-cost as it is not viable to ship films on high-capacity optical discs or deliver 10-megabit links to consumers.

2.5.4.1 MPEG-1

The first major video compression standard to satisfy these demands was MPEG1 [MPEG, 1993]. MPEG1 typically encodes video at 352 x 240 resolution and 25/30 frames per second using 24bpp and was designed to store 74minutes of VHS-quality video on a 650MB CDROM. Later, MPEG2 was devised to improve on the resolution of MPEG1 and is currently used to deliver approx 120mins of video on a 4.7GB Digital Video Disk (DVD). Because movies and other video sequences often contain audio as well as video, MPEG1 is also capable of compression audio in addition to video – the audio storage technique is the basis of the popular MP3 (MPEG layer-3 audio) used to store music on personal computers. However, since we are primary focused on video compression, MPEG's audio compression techniques will not be discussed further here.

Due to its original success, MPEG has become the defacto image compression standard in terms of support and development. Although MPEG1 and MPEG2 have been largely superseded by MPEG4, the original MPEG specification and revised version 2 standard radically improved the quality and compression ratios achievable for compressed digital video sequences.

Like M-JPEG, MPEG exploits the JPEG compression engine but also employs spatial compression to increase compression rates whilst maintaining visual quality. This spatial compression takes advantage of the fact that within movie video-sequences, there is often little or no difference between consecutive frames. Even though it is common for movie-makers to 'cut' to a new scene every 3-4 seconds, with a typical frame rate of 25fps, this is still a high level of redundancy and allows for significant savings over simply applying JPEG compression to each frame in its entirety. [TANENBAUM, 1996] [WATKINSON, 2001]

For a scene where there is little or no difference between frames, such as when the camera is stationary, there will be large amounts of redundancy and this could be found be simply comparing two such frames on a pixel by pixel basis. However, if the camera was panning

slowly rather than stationary, then this technique is less useful as corresponding pixels between the two frames may not have the same colour values due to the offset of the panning, despite the two images appearing to be very similar. MPEG can handle both of these scenarios within a video sequence – this inter-frame compression is the main difference between MPEG and JPEG/M-JPEG. [WATKINSON, 2001]

2.5.4.2 MPEG Frame Types

MPEG compression output consists of four distinct frame types:

a) Intra-coded Frames (I-Frames) – these are simply JPEG encoded frames.

b) Predictive Frames (P-Frames) – these are the difference between the current frame and its precursor.

c) Bidirectional Frames (B-Frames) – these are the differences between the current frame and its successor and precursor.

d) Delta/DC Frames (D-Frames) – These frames are low quality frames that can be shown during 'fast-forwarding' of a video sequence. These can be used to alleviate the need for the decoder to decode I/P frames at these higher rates.

In practice, most implementations do not use B or D frames, as I and P frames are sufficient to encode an arbitrary video sequence and retain forwarding and rewinding capabilities. I-Frames are needed for a number of reasons – although it is feasible that an MPEG sequence could have a single I-Frame at its start, followed by all P-Frames, this would likely have problems in that maintaining image quality would be difficult for the MPEG algorithm due to the lack of a pure frame to periodically act as the reference for the calculating inter-frame differences, i.e. if lossy P-Frame comparisons were allowed to go on for too long, it is likely that the displayed image would lose quality over time. Additionally, and perhaps of more practical benefit, periodic I-Frames are required to allow random access to the video sequence – without I-Frames, if viewing started in the middle of the sequence, it would be impossible for the decoder to display the initial request frame correctly without having

calculated all the previous P-Frames from the initial I-Frame. This also applies to fast-forwarding. Lastly, if the MPEG stream is being broadcast live, should a view not receive the initial I-Frame correctly, they may never be able to view the remainder of the sequence correctly. As a result of these issues, I-Frames (also known as key-frames) are inserted into the MPEG output at regular intervals. [WATKINSON, 2001]

P-Frames store the difference of the current frame with its precursor, whether it was an I or P-Frame. These frames consist of macroblocks, which represent a 16 x 16 block of luminance information and a 8 x 8 block of chrominance information for a given section of the current frame. The encoder searches the previous frame to find a macroblock that is a match with the current frame block, within a predefined tolerance. An example frame-sequence is given in figure 2.5.4.1, with a static background and moving object between the 3 frames.

Figure 2.5.4.1: An illustration of differences between successive video sequence frames

The three images above show how the background remains constant whilst the car's position changes between the frames. The macroblocks that contain portions of the background scene will match exactly, whilst those that represent the car will be offset between the frames and will need to be found. However, the MPEG standard itself does not specify how such searches should be done, rather it just specifies the concepts for the various frame types and how then should be subsequently decoded into a representative video sequence. It is up to the encoder to decide how far to search, the macroblock similarity thresholds required to constitute a match between respective frames. a simple implementation could be to search like-positioned macroblocks for the current and preceding frame and if they match a given tolerance, *t*, then mark them matched – otherwise encode the whole macroblock of the current frame anew. [WATKINSON, 2001]

Once a macroblock match is found, it is then encoded by taking the difference between the block's value in the current frame and its value in the previous frame. This is then subject to JPEG encoding. Likewise, an unmatched macroblock gets treated to JPEG encoding directly, just as it would be in an I-Frame. [TANENBAUM, 1996]

2.5.5 MPEG Performance

MPEG is incredibly flexible and powerful in the freedom it gives the encoder in the macroblock search process whilst still allowing any standard MPEG decoder to decode and display the results. This allows implementations to use enhanced search techniques to optimise the MPEG output, but it also gives the flexibility of producing an MPEG videostream that fits a particular bitrate for transmission purposes (such that the MPEG stream can be played in real-time down a particular link bandwidth) or that a particular media-size could hold a predetermined duration of digital video. This deterministic ability makes MPEG well-suited to these applications.

Largely as a result of the macroblock searching process, MPEG is a highly asymmetric compression algorithm in that compression of a video sequence takes much more computation than its subsequent decompression. As the macroblock search engine increases in sophistication and complexity, the more asymmetric the implementation becomes. Although this makes MPEG1 unsuitable for real-time video editing and videoconferencing, it still makes it well suited for video distribution as the encoding step can be centralised where dedicated hardware can be employed to reduce encoding times. [TANENBAUM, 1996]

Although encoding MPEG is much more intensive than simply decoding it, the playback of MPEG1 streams wasn't practical for software-only implementation until the advent of the Pentium Pro / Pentium-II line of 80x86 CPUs – the decoder requirements were simply too great for lesser-specified machines.

Finally, MPEG1 is capable of taking a 472MB/sec video stream and producing VHS-quality video at the same resolution in just 1.2MB/sec. Although it cannot produce broadcast-quality video at this bitrate and resolution, its successor, MPEG2, is capable of broadcast-quality video at HDTV resolutions of 1920x1080. MPEG2 has other subtle improvements on MPEG1 and remains backwards compatible with the original standard.

2.5 Summary

There are numerous data compression approaches but these can generally be categorised as generic or lossless content-specific or lossy content-specific. There are applications for each of these types, though lossy compression only works in practice where the algorithm has knowledge of the type of content being encoded and thus understands the types of data loss that can be tolerated by an application.

Digital imaging and video are extremely demanding in terms of both storage space and also the bandwidth needed for their transmission, making them ideal candidates for data compression in order to alleviate these issues. Both images and video are presented to a viewer for perception using analogue methods, such as the human eye, and therefore also lend themselves well to content-specific lossy compression. Indeed, the challenges associated with both these forms of digital media necessitate more sophisticated content-specific techniques to minimise their impact.

Both JPEG and MPEG are well-established lossy compression schemes in the fields of imaging and video, respectively, with MPEG drawing on the image-compression abilities of JPEG as part of its approach to video compression. Fractal methods, such as quadtree and IFS compression, have also been used to compress images and hold the possibilities of significant gains over more conventional lossy algorithms. Just as MPEG borrows from JPEG, it is certainly conceivable that video compression could be readily achieved using an algorithm that borrows from a fractal-based image compression method.

3.0 Fractal Video Compression

3.1 Introduction

There are a number of algorithms and techniques available to compress both still bitmap images and the sequences of such images that represent motion video. Many of the techniques used in the compression of single bitmap images – frames in terms of a digital video sequence – equally apply to the compression of motion video.

Many modern video compression techniques take a 'two-step' approach to tackling the video compression problem: firstly, compress each frame using a high performance (in terms of both quality and storage reduction) image compression algorithm; secondly, exploit inter-frame similarities often inherent in motion video sequences to further increase the compression results.

MPEG is the de-facto image compression technique being widely supported, innovative and highly efficient. MPEG builds heavily on the JPEG image compression standard and is perhaps the pioneering example of the two-step approach. Using a sequence of P-Frames (inter-frame differences with JPEG compression applied) with regularly interspersed I-Frames (straightforward JPEG compressed frames), MPEG can achieve compression ratios of 100:1 whilst still delivering VHS-quality video performance.

MJPEG (Motion-JPEG) however, is a much-simplified scheme that simply JPEG compresses each frame of the video sequence. The MPEG and JPEG algorithms together demonstrate both the usefulness of building upon existing image compression techniques and exploiting video's unique properties. Because MJPEG does not examine the inter-frame relationship, it clearly cannot deliver anywhere near the performance of MPEG, as each subsequent frame is subject to full JPEG compression on its entire contents.

3.2 Evaluation Criteria

The focus of the research concerns the performance of industry-standard MPEG compression implementations compared to Fractal video compression techniques, specifically video compression techniques based on the quadtree Fractal image compression approach. Assessment criteria for this comparison include:

i) Compression ratios: Theoretically the most crucial factor – the relative reduction in storage requirements for a standard frame-sequence whilst still delivering to a set visual quality threshold.

ii) Encode and decode performance: The amount of computing time required to deliver a compressed encoded sequence (again, to a specific quality threshold) and similarly the computing time required to subsequently decode the compressed data and obtain the 'original' video sequence. This should also include a comparison of how asymmetric the encode/decode functions are for each implementation of the algorithms.

iii) Accuracy of the retrieved video sequence: the most powerful video compression algorithms, including MPEG, are lossy but aim to throw away the least useful data such that the quality of the resulting video sequence remains relatively low for the level of compression achieved. The fractal compression method will need to be assessed to determine how well it maintains image quality and integrity through the compression process

iv) All comparisons with MPEG will naturally need to take place at the same resolution; this should be the native 320x240 resolution of MPEG1.

All of the above factors will be affected by the computing time available for each method, i.e. increasing the complexity of the P-frame search method employed by the MPEG algorithm or the

depth of quadtree searching using the Fractal compression technique will increase the amount of processor time required to encode a given sequence. The visual accuracy thresholds for each algorithm will also affect compression ratios and encode performance – as the threshold for a search match increases, it becomes more likely that more fruitless searches/comparisons will take place per frame. This will potentially lead to increased storage requirements as more detail needs to be encoded to attain the threshold's requirements.

Whilst MPEG also includes a method to compress the audio (MP3 – MPEG layer 3 audio [MPEG.COM]) that often accompanies a video sequence, i.e. in a film, etc., audio compression is a separate media compression research area and will not be handled as part of the fractal algorithm or this research.

3.3 Algorithm Construction

To aid in the comparison of Fractal video compression versus MPEG compression, it is necessary to devise an algorithm that uses a suitable Fractal image compression for its base whilst borrowing established video compression techniques from MPEG and others. The elements of the algorithm are as follows:

3.3.1 Fractal Image Compression Technique

The chosen Fractal image compression algorithm will form the core of the frame compression function. There are two principle Fractal compression techniques that could be employed here – quadtree-based compression or Iterated Function System (IFS) compression. However, the issues surrounding the Inverse Problem and the problems in the practical implementation of a suitably accurate metric makes IFS-based image compression difficult. Quadtree compression, however, is easy to implement and has a number of attractive properties, such as:

i) Finite Search Space - quadtree divides each target image into a series of squares (ranges) and attempts to map other square image sections (domains – these are twice the dimension of the ranges) onto these. The domain-pool is the collection of all domains that can map to a range – this pool is finite as each domain must be larger than a range (where a range contains 1 or more pixels); the domains can only map to a specific range through rotation and offset transformation together with fixed levels of brightness/contrast adjustments.

The size of the transform search space (T) for a given range can therefore be calculated according to the following, where R is maximum number of ranges, D is the maximum number of domains, Q is a positive integer denoting the maximum depth of quadtree exploration and B & C are the maximum number of Contrast and Brightness changes that can be applied to a transformation.

i. $R = 4^Q$

ii. $D = (R / 4)$

iii. $T = DRBC$

As there is a maximum depth to which the quadtree can be explored and the number of ranges is finite for a finite image resolution, the search space is also finite and the algorithm is deterministic.

ii) Storage efficiency - large image area can be covered in a single transformation whilst allowing other areas to be covered with a larger number of transforms in order to preserve visual detail. Transform mappings can be stored as follows for a given range:

a) Source Domain - either indexed from left to right throughout the image if a fixed size domain pool is employed or described by an x and y offset in the image

b) Orientation - describes one of 4 possible rotations of a source domain (turns 0^{o}, 90^{o}, 180^{o}, 270^{o})

c) Brightness/Contrast - describes the amount that each source domain has it's brightness/contrast increased or decreased by to fit a range

d) Offset - The co-ordinates for the top left-hand pixel of a target range

e) Depth - The depth of a range within the tree (and therefore the its dimensions)

iii) Image rendering at a larger scale - the contractive nature of each transform (i.e. that each domain is twice the size of each range) give quadtree's their fractal properties and allows images to be arbitrarily scaled through the introduction of artificial fractal detail. The effect produced by such scaling is sometimes considered visually preferable to bilinear interpolation.

iv) Metric – quadtree can use a variety of metrics, with the simplest implementation being a metric that, for every pixel in a target matrix, compares each pixel for the proposed mapping to that of the target pixel. The mapping with the lowest difference across all pixels has the best solution 'fitness'. Such a simple metric would not provide sufficient accuracy or relative distance to the true mapping solution to be useful in an IFS-based implementation.

Although it has drawbacks, basic quadtree fractal compression is a sensible choice to provide the image compression element of the algorithm as it is relatively easy to understand and implement.

3.3.2 Inter-frame redundancy

To compete with the successful and established MPEG standard, any new video compression algorithm needs to be able to calculate inter-frame redundancy and effectively eliminate the need to store this. Inter-frame redundancy and image compression are the two most important elements in digital video compression.

The algorithm will borrow heavily from MPEG in that inter-frame redundancy will be calculated using separate process to that of the image compression, i.e. the fractal compression technique will not in itself be used to remove this redundancy and instead the system will follow MPEG's two-step approach to video compression. There is a possibility that a fractal compression algorithm could be used to carry out both of these vital two steps, this could be achieved by allowing contractive fractal mapping between two consecutive frames, that is inter-frame as well as intra-frame mappings. However this is outside of the scope of this study.

The above inter-frame method should also retain the I-Frames and P-Frames of MPEG, with I-Frames being subject to complete fractal compression and occurring at a predefined frame interval to both preserve the quality of the sequence by providing a complete verbatim 'baseline' frame periodically; to allow for rapid 'fast-forwarding' within the compressed sequence without incurring a large performance penalty in the decode element; to allow the sequence to be resumed at any point whilst being able to rebuild the select frame at that point accurately via the preceding I-Frame. The frequency of I-Frames is a compromise between the benefits that they offer – i.e. a quality reference frame that is only subject to image compression for both sequence quality integrity and also random access to the sequence itself – and to the fact that I-frames require more storage than P-Frames due to the fact that they do not exploit inter-frame

redundancy and therefore reduce the overall compression of the video sequence. I-Frames are typically used at intervals of 0.5 or 1 seconds within an MPEG sequence and a 0.5 second frequency will be used for the fractal compression algorithm.

It is intended that the inter-frame redundancy will be calculated by comparing the same 'domain' from two consecutive frames and then applying the same metric as used by the quadtree algorithm in order to determine whether that portion of the initial frame needs to be recorded in its successor. This will certainly allow for code re-use within the implementation and is the basis of MPEG's approach to this particular challenge, where the domains within quadtree compression are akin to the macroblocks in MPEG compression.

3.3.3 Chrominance Compression and Colour Representation

Both JPEG and MPEG exploit the human eye's higher response to luminance than chrominance, allowing them to achieve a 50% reduction in image data during this stage of the JPEG algorithm. The compression achieved during this discrete step within the JPEG algorithm would not only allow the fractal compression algorithm the same initial compression gains as seen in the MPEG implementation, it can also be used to allow the algorithm to compress colour images.

To exploit the chrominance compression possible with YCbCr, JPEG averages blocks of 4 pixels in the 'Cb' and 'Cr' chrominance matrices to reduce them to 25% of their original size. When JPEG breaks up all three matrices (including the original size 'Y') into 8x8 matrices, there are 50% less blocks to be compressed by the rest of the algorithm. The fractal compression algorithm must also take advantage of the matrix averaging step of JPEG but needs to ensure that this compression gain is not lost or reduced within the remainder of the algorithm.

These gains can be retained by allowing the algorithm to compress matrices that are 25% of the target image size – i.e. the algorithm accepts the reduced size chrominance matrices natively and

the larger luminance matrix 'Y' itself is split into 4 matrices, each of which are the same size as their chrominance partners. This means that for a 640x480 resolution source image, the system would break this down into 6 320x240 matrices and compress each of these individually. This is shown in figure 3.3.3.1.

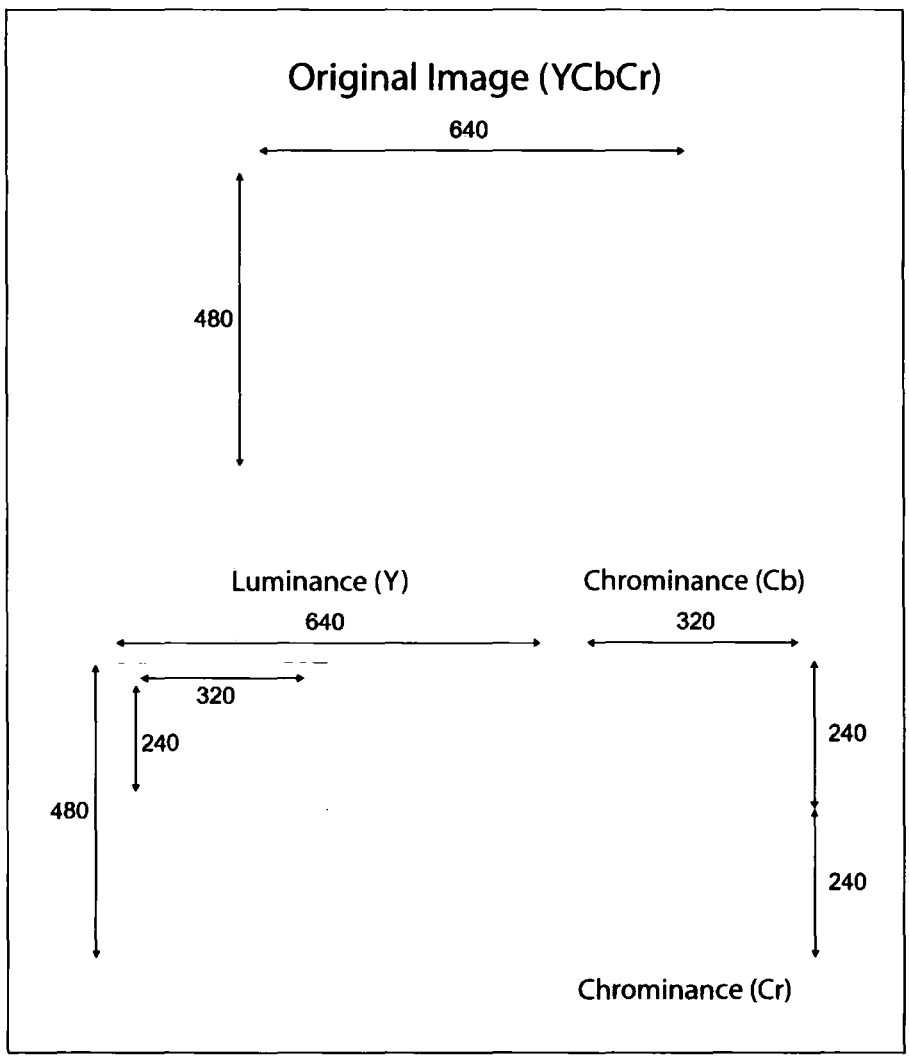


Figure 3.3.3.1: Breakdown of a 640x480x24 image into six 320x240x8 images

The above approach has a number of benefits for quadtree compression:

i) It allows a full 24-bit 'true colour' image to be compressed using a series of 8-bit grayscale images, with the 50% data reduction being preserved due to the use of 6 320x240 matrices rather than set of 12

ii) The grayscale images themselves are easy to assess for fitness, as an 8-bit value assigned to a pixel directly corresponds to a grey value between 0 (black) and 255 (white). For example, a pixel within a domain that is only 2 values (and accordingly, shades) away from that of the range will have a good fitness whilst if it was 46 shades away it would have a much poorer fitness.

iii) The matrices that result from this process are sufficiently large to give a good sized domain pool, as an image's resolution decreases there are fewer domains available given a constant domain size. Statistically, more domains will help in finding better matches during the quadtree search process and quadtree performs much better when this is the case.

3.4 Toolkit Requirements

The implementation of the above algorithm will involve a number of tools, each performing a single element of the video compression algorithm. It is not the intention to produce a fully functioning 'black box' codec that simply from a raw video sequence to a fractal compressed sequence and vice-versa, rather each stage of the compression will be performed separately by the manual feed of one tool's results to another as input thus performing all the steps of the algorithm outlined above. As well as simplifying the implementation of the various steps involved, this approach also allows each step to be analysed independently. Parameters for fitness, range/domain sizes, etc., can be adjusted as appropriate.

Implementation of certain elements of the algorithm, such as the representation of colour through 6 grayscale chrominance/luminance matrices, is not required; rather the outlined theory can be demonstrated as a manual process using a 3rd-party photo processing package such as Adobe Photoshop. The most critical stages will be the two-step video compression process, and in particular the quadtree compression of both I-Frames and P-Frames.

Screenshots of the tools are shown in figures 3.4.1 - 3.4.3:

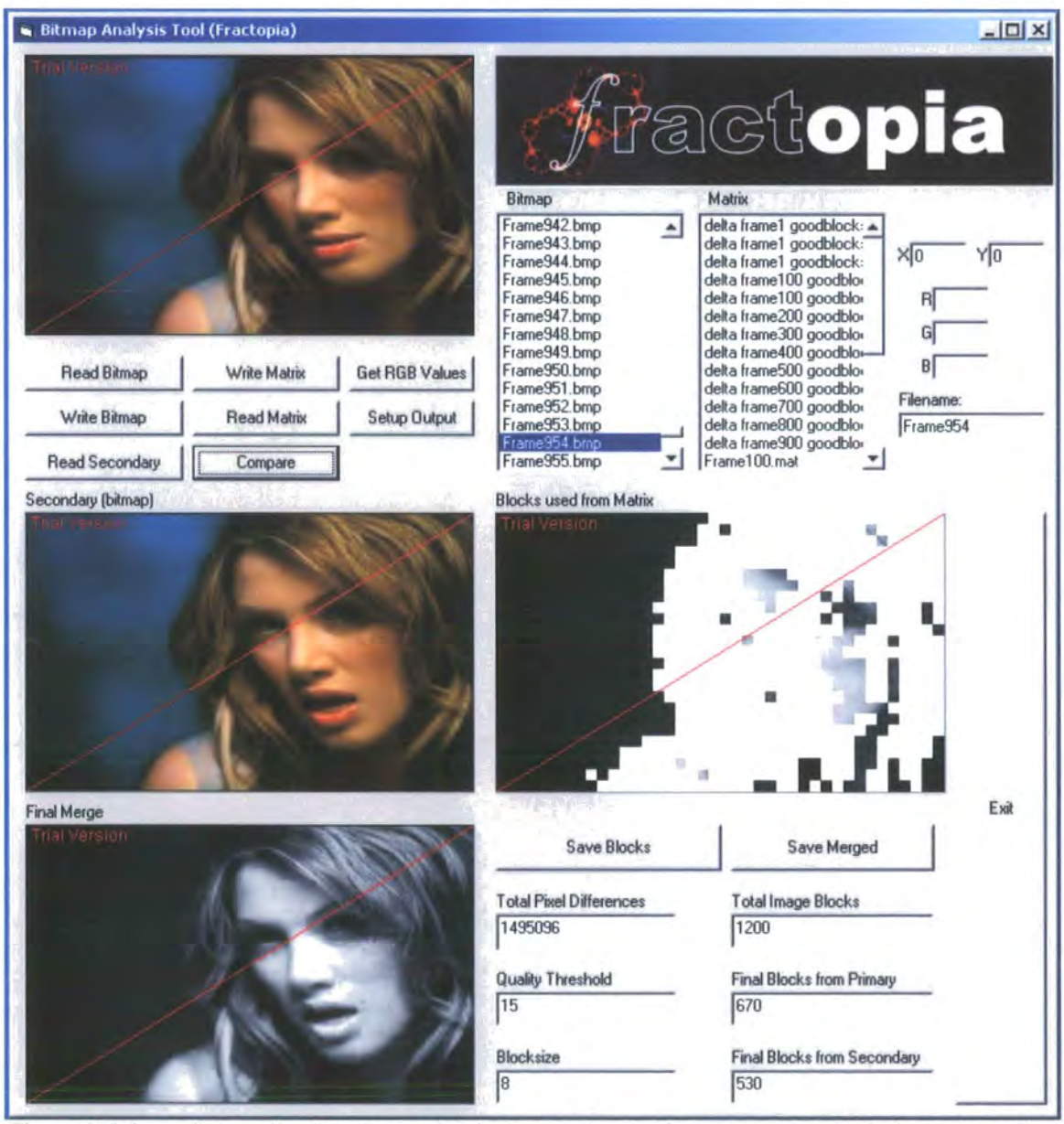


Figure 3.4.1: dialog allowing conversion between standard uncompressed Windows .bmp files and the matrix files readable by the quadtree compression tool, and vice versa.

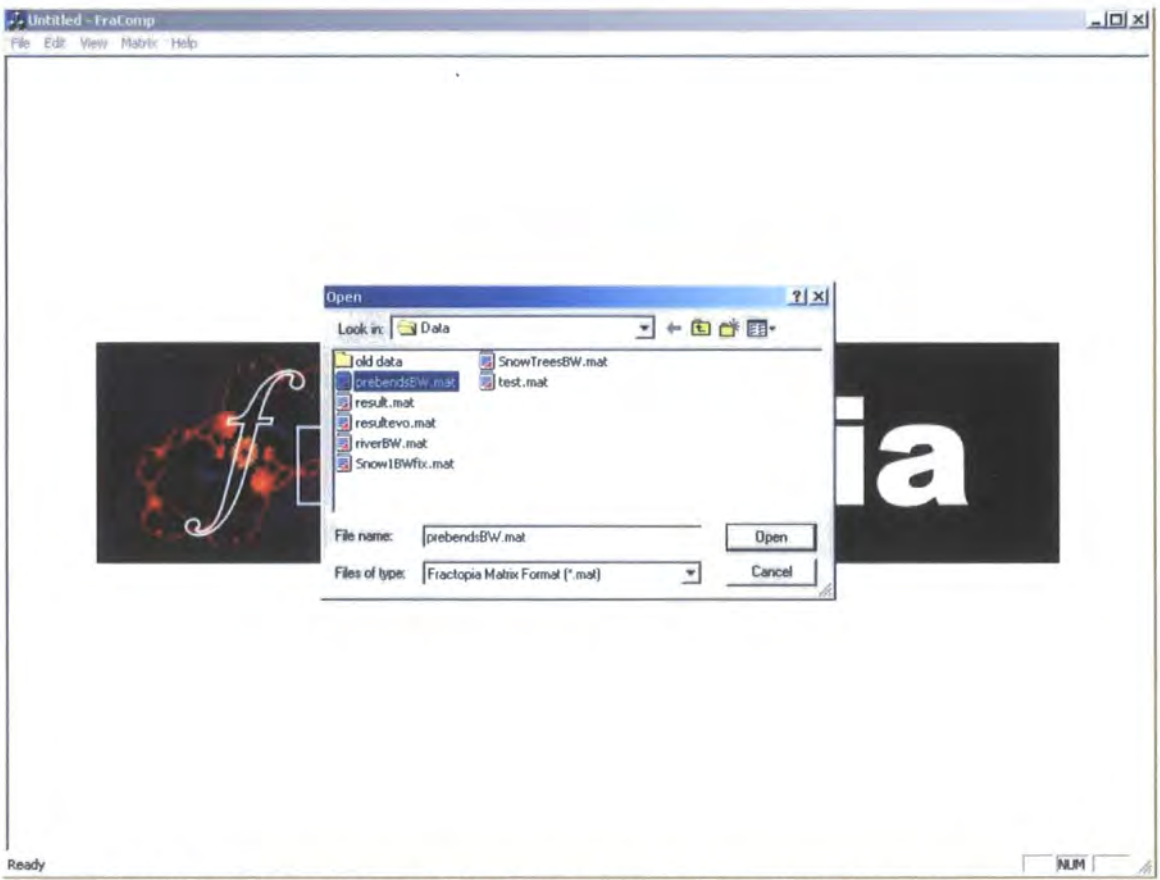


Figure 3.4.2: dialog showing matrix files being opened and read into fractal compression tool

Figure 3.4.3: selected matrix undergoing compression within the tool itself.

3.5 Summary

The aim of this research is to understand how an algorithm to compress video using a fractal image compression technique can be implemented. This should involve the consideration of how to adapt other best-practice elements of established video compression algorithms to fit with this image compression method and how the resultant compression method compares in performance, both in terms of quality and speed, to benchmark compression standards such as MPEG. In addition, some of the unique features enabled by fractal image compression, such as image scaling and the artificial detail creation involved in such a process can be examined to understand how well these behave in a video implementation.

4.0 Fractal Image Compression Tool

4.1 Introduction

This chapter covers the construction and acquisition of the two main software tools that have been used to investigate both the Fractal and conventional compression techniques researched during this work. It is intended to cover the development environments used for each software tool and the functionality of each of the tools and how these meet the testing requirements.

4.2 Summary of Testing Requirements

In order to evaluate the performance of fractal image compression techniques, and specifically quadtree-based covering techniques, when applied to video compression, the base requirement necessitates that a number of tools be developed, which together allow the compression of a 2D-Matrix with an appropriate range of values for each element – e.g. 256 levels of colour/luminance. These tools should also be able to cope with differing parameter sets to adjust the level of compression applied to an image (and generally, therefore, adjust the quality of the result).

A summary of requirements is given below:

i) Conversion of a standard picture format file (e.g. Windows Bitmap) to a matrix representation that can be handled by the rest of the toolset. The file format used should be itself be compression-free to avoid introducing other variables in the testing process.

ii) Conversion of a matrix representation of an image to a standard file format.

iii) A tool that allows a user to open a matrix representing an image and apply fractal-based compression techniques to it. This tool should be able to save the results of these fractal transformations on the original image into the same matrix file format. In addition, the tool should also record the fractal transform set (as per fractal compression technique applied) that would be required to recreate the compression results – this set is effectively the compressed version of the original image.

iv) A tool that allows the user to view a matrix representation for comparison between uncompressed and compressed images.

v) A tool that allows the user to take the fractal compression set that has been generated as a result of the application of a particular fractal algorithm to an image and use this set to generate (i.e. render) an approximation of the original base image. This allows validation of the image compression tool's output and results.

vi) The compression tool will need to be able to cope with a number of parameters being of a variable nature, such that different compression scenarios using the same base algorithm can be examined. These should include parameters such as the depth/size of a quadtree, or the dimensions of the tiles used to cover and image.

vii) In a similar way to vi) above, the tools must also be able to feedback success factors for a compression run on an image. This should simply be the return of the best match/similarity value seen during the exploration of the fractal search space. This value will be calculated by the metric used to evaluate the level of similarity between source and target matrices/images.

viii) A number of the compression and other operations are likely to be compute-intensive. Development should be optimized and use a platform that can deliver the best performance and experience, especially given the large number of operations that will be required for meaningful analysis and comparison.

4.3 System Environment Selection

The first decision point in the project is to decide which operating system and which development system should be used to implement the system. The two main environment choices are UNIX/LINUX and 32-bit Windows (Windows 2000 / XP). However, since the vast majority of the development and computational work will be done on an Intel PC, it is likely that the choice will be between LINUX and WIN32.

4.3.1 UNIX / LINUX

UNIX has been around for a long time now, and as a result is very well supported. Despite the number of different incarnations available, essential features remain the same, making for a relatively consistent programming environment. Most of the university's high-end multiprocessor machines use this OS.

The main problem with UNIX is the lack of fully integrated development environments (IDEs), which are essential for developing applications (especially image manipulation) for a windowing system. If such an IDE is not employed, there is a steep learning curve in terms of the development libraries for X-Windows programming. Apart from this, the lack of such programming tools makes it difficult to interactively debug the system, which would certainly be required with a system with a number of complex and recursive algorithms. A major advantage that UNIX does have is that it does provide a wide range of software tools, together with much more robust control of memory and processing resources that the Microsoft Windows line of OSs. This may be prove significant when a large number of large matrices are being stored and accessed during the use of these tools.

4.3.1.1 Advantages of UNIX / LINUX:

i) Wide range of software tools

ii) Robust Memory management

iii) Access to multiprocessing machines

4.3.1.2 Disadvantages of UNIX / LINUX:

i) Limited development tools – fewer IDEs

ii) Steep learning curve with X-Windows programming

iii) Personal access to the university UNIX systems requires network connection

4.3.2 Microsoft Windows (32-bit)

The Microsoft Windows series of operating systems are the most popular operating systems for IBM compatible PCs. The family has been around since the mid-1980s, and has gained a large level of hardware and software support.

There are a number of excellent programming environments available that are suited to the task of producing Windows based programs in an interactive 'visual' environment. Systems such as The Microsoft Visual Studio family of products offer visual programming in both C++ and structured basic languages. Visual C++ offers a full IDE with many software tools and supporting libraries. This environment has proved itself to be stable and has been widely accepted as an industry standard. There is also a level of consistency with Windows

programs, they all look and feel very similar – this is a big advantage when trying to make a system user friendly.

4.3.2.1 Advantages of Windows

i) Excellent development environments and tools, with good graphics handling routines

ii) Widely used and supported

iii) Consistency of user-interface approach

4.3.2.2 Disadvantages of Windows

i) Windows systems can be more unreliable than their UNIX/LINUX equivalents

ii) Windows systems are generally less powerful than large UNIX machines

4.3.3 Operating System Selection

Despite the more mature and bullet-proof memory management of the UNIX environment, Windows has been used to develop the toolset for this investigation due to the consistency and maturity of the user interface and the quality of the Visual development environments.

4.4 Tool Development Environment

Whilst Windows supports a wide range of languages, it is really only sensible to consider the integrated visual systems as the tools used to within these investigations need to be able to display images and give on-screen feedback in order to meet the requirements set out above. A GUI for each tool will greatly aid its usability both in terms of carrying out various testing

operations and also for evaluating the relative success of these tests. The most widely available and adopted systems for Visual programming under WIN32 are Microsoft Visual Basic and Microsoft Visual C++.

4.4.1 Visual Basic (VB)

Visual Basic is a system for developing 32-bit Windows applications that are written in a BASIC style language, with various adaptations to use the various Windows features and functionality. Visual Basic is a popular development language and environment, mostly with beginners to Windows programming. It offers ease of use and rapid application development with reasonable tool support, including simple debug facilities. A big advantage of VB is the level of support for imaging operations and the availability of libraries that allow the developer to examine a bitmap at the pixel level. VB's major disadvantages are that it has relatively slow instruction execution speed, inflexible memory handling and severely limits the developer using more advanced features of the Windows system.

4.4.2 Visual C++ (VC++)

Visual C++ is a system for developing powerful 32-bit Windows applications. It includes a full featured yet easy to use IDE. It is based upon the standard C++ language, with Windows support automated by the Microsoft Foundation Class libraries (MFC). Visual C++'s high-end features include fast execution speed, dedicated processor support for the various 80x86 instruction sets, an excellent toolset and extremely powerful interactive debug, full WIN32 API support and the ease of MFC based Windows development.

With Visual C++ being supporting ANSI C++, any massively processor intensive code can be developed under UNIX on C++ where greater computer power is available in order to fully develop and optimize algorithms. Once the code is running at a reasonable speed then it can be ported back to VC and a standalone PC and integrated into the system. The major disadvantages of Visual C++ are that it is generally more complex than other visual

environments and it can be difficult to integrate external ActiveX and other modules into the environment.

4.4.3 System and Language Selection

Visual C++ is the obvious choice for building a software tool that runs compute-intensive algorithms and needs to work on large data structures, as necessitated by image processing and the large matrices required to store and contrast the various images used during the compression processes. Fast execution is crucial to a system that performs a great deal of repetitive processing whilst the excellent low-level functionality and environment will not restrict development or final functionality. The excellent debugging facilities will be crucial to developing a system that aims to explore the boundaries of such an abstract concept.

However, Visual C++ suffers in the its support for bitmap images – specifically in terms of reading these into a data structure where they can be processed rather than just displayed. To resolve this issue, two software tools have been constructed – the first dealing primarily with the image processing tasks and generation of a simplified matrix for subsequent processing – this tool has been developed in Visual Basic. The second tool will take the matrix output from the first tool and perform the compression and analysis on it. This second tool, which is the primary computational tool, has been developed using Visual C++.

4.5 Developed Software Tools and Functionality

Two software tools have been developed to support this project:

4.5.1 Fractopia Compression Tool (FCT)

FCT is the primary investigative tool and performs the actual compression functions and results to the user. The tool is GUI based and is built in Visual C++ using both MFC and custom-built classes, supporting the following features:

i) Matrix support for 320 x 240 images using 256 distinct levels, thus allowing an 8bit grayscale quarter-VGA image to be processed.

ii) User load/save of Fractopia Matrix format image matrices (*.mat files) into the system via Windows file browser.

iii) GUI driver queries on matrix content, including value interrogation at (x, y); value change/set at (x, y); randomization of matrix content; sum of all matrix elements.

iv) A simplified image-comparison metric implementation based on gray-level comparison for each matrix element.

v) Quadtree compression of current matrix state results in quadtree representation of matrix, matrix output of that representation and output of the metric of similarity between quadtree-generated and original images (*.fmf files).

vi) IFS compression using Evolutionary Programming techniques. The function simply returns the metric from the comparison between the original image and best IFS EP solution.

4.5.2 Fractopia Bitmap Analysis and Render Tool (FBART)

FBART allows QVGA 8-bit standard bitmap images (Windows Bitmap files) to be converted to an ASCII matrix representation for processing by the FCT. This tool is GUI based and has been built in Visual Basic to take advantage of the image manipulation libraries available. FBART supports the following features:

i) Viewing of Windows Bitmap (.bmp) files

ii) Viewing of Fractopia Matrix files (.mat) files

iii) Bi-directional conversion between .mat and .bmp file formats

iv) RGB value interrogation of both file formats

v) Quadtree render of a Fractopia compressed image (.fmf) file to the screen

4.6 Data Structures

The primary data object employed by the Fractopia Compression Tool is an 8-bits per pixel bitmap image. A bitmap is simply a matrix, where the value of each element holds a colour (from a predefined palette) or luminance value (in the case of a grayscale image, which is the image type used by FCT).

Arrays are the most commonly used data structure to store matrices and bitmap images [AHO, 1987], specifically in a bitmap where the [x,y] position of an array element exactly corresponds to the relative (x,y) screen position of a pixel – each element of the matrix represents a pixel (*pic*ture *el*ement). The luminance value held at the specified array location, therefore, is the value for the corresponding pixel. Allowing 8 bits of storage as an integer [LIPPMAN, 1991] per array element allows for $2^8 = 256$ distinct luminance values to be stored. This concept is demonstrated in figure 4.6.1, where the matrix representation of a bitmapped image has the corresponding grayscales overlaid in accordance with each element's value:

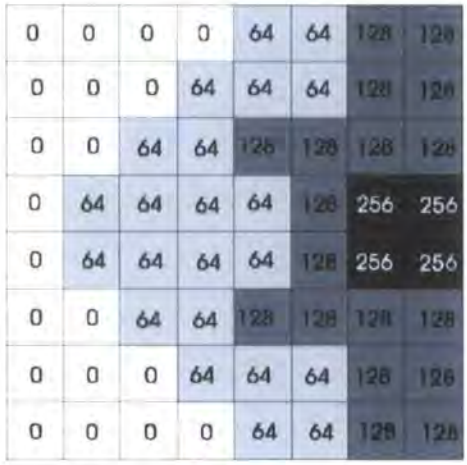


Figure 4.6.1 – 2D matrix representation of a bitmap image

In this instance, an 8x8x8 (8x8 pixels, 8 bits per pixel) bitmap image is being stored. Whilst it is possible to store the bitmap matrix using a linked-list, this would only be beneficial if it had a large proportion of zero-elements within the matrix. As each element of the list data structure would need to store four values – the luminance value; the x coordinate; the y coordinate; and finally the pointer to the next list element, the potential for memory savings using the linked-list model would soon evaporate. A 2-dimensional array's inherent matrix structure negates the need to store x and y coordinates for a pixel.

An array also lends itself to representing a bitmap because:

i) Read/update operations on the matrix – since the matrix elements directly relate to the image elements, the functions that read and write these elements do not have to traverse a list or perform any algorithmic work to map the data structure to the real-world object.

ii) Data structure definition – with the FCT tool, it is known from the outset that there is a need to handle a 320x240x8bit bitmap image. Therefore, it is possible to define the array at the outset in the knowledge that the size of this data-structure does not need to change.

4.7 Computational Performance

An important factor in compression algorithm performance is the amount of computation time required to encode an image into the compressed state. Compression time is significant for a number of reasons. If a compression method required significantly more computational time than other, established, methods then it is less likely to gain mainstream acceptance. If a compression method is computationally intensive then the number of devices that are able to use it will be reduced – e.g. low power embedded devices may not have sufficient computing performance to implement the algorithm.

The developed quadtree compression tools can routinely compress an image in approximately 5 minutes on a 2.8GHz Intel P4 computer and uses less that 5 megabytes of RAM including the GUI. Whilst a reasonable performance in terms of fractal compression, it's important to bear in mind that the equivalent JPEG routine will take less than 2 seconds on the same hardware. Improvements to the performance of the algorithm as part of a future development strategy will be explore in chapter 6.

4.8 Summary

Two software tools are to be developed under the Win32 environment to handle all aspects of image format conversion, fractal compression/decompression and subsequent analysis. The fractal compression aspect is likely to be the most computationally intensive and as such will be written in Visual C++, whilst internal image representation will utilize an 2-dimensional array.

Two tools will be developed – the Fractopia Compression Tool will handle the compression of bitmap images using the quadtree algorithm, whilst the Fractopia Bitmap Analysis and Render Tool will allow for image compression and analysis of fitness of compresses images compared with the originals.

5.0 Evaluation of Quadtree Image Compression

The performance of the quadtree image compression element of the system under evaluation is critical, as this stage is responsible for the removal of spatial redundancy. The following section presents the results of applying the quadtree compression to a selection of six non-sequential images, selected according to their differing characteristics. The performance of the quadtree algorithm is evaluated according to the following criteria:

i) Perceived Image Quality – How similar is the compressed image to the original target image?

ii) Calculated Pixel Shading Difference – a total of the difference in greyscale value for each pixel in the compressed image when compared to the original target pixel in the source image. The greater the difference between a source pixel and the corresponding pixel on the compressed image then the poorer the match and visual appearance. The image is judged on the total of the difference for each and every pixel.

iii) Storage Size and Compression Ratio – how much storage space is required to store the compressed image and the ratio of this space compared to that required for the original image. A single greyscale 320 x 200 image occupies 64kilobytes uncompressed.

iv) JPEG Comparison – how well do the quadtree images compare to a JPEG compression of the original source image, taking the storage requirements and compression ratio of each into account.

Subsequently, the compression characteristics of three of these six images are subject to further analysis in terms of the relationship between the quality of the compressed image and the level of compression achieved.

5.1 Trial Images

The following set of digital images was used as they each highlight certain aspects of fractal compression and the associated challenges.

5.1.1 Image 1 – "Jan"

Figure 5.1.1 is a greyscale image of a Spaniel dog against a background of strewn fallen leaves. The compression characteristics of this image are that it appears to have a large degree of self-similarity – the fur of the dog is of a very regular texture as is the collage of leaves that forms the background. The image does, however, have a relatively high level of detail in the foreground subject as well as exhibiting high levels of contrast on areas that are similarly textured in appearance.

5.1.2 Image 2 – "Prebend's Bridge"

Figure 5.1.2 is a greyscale image of Prebend's Bridge in Durham. The image has trees that partially obscure the view and also a large amount of white sky. This has resulted in an image with very high levels of contrast, with the trees rendered as very dark levels and the sky as practically white level greyscales. The bridge itself is present as mid-tone greyscale.

5.1.3 Image 3 – "Snow Trees"

Figure 5.1.3 is a greyscale image of a winter snow scene. Unlike the previous two images, the image doesn't present highly contrasting areas but does instead exhibit a larger amount of mid-tone greyscale together with lots of fine details in the branches of the trees. However, there is also a degree of self-similarity present in both the clear sky and foreground areas of the image.

5.1.4 Image 4 – "River Wear"

Figure 5.1.4 is a greyscale image of the River Wear in Durham. The image shows moderate levels of detail in the trees along the river's banks but these aren't as great as those shown in Image 3. However, this image is notable for being comprised almost entirely of mid-tones and that these are distributed in four horizontal 'stripes'.

5.1.5 Image 5 – "Match"

Figure 5.1.5 is a greyscale image of an igniting match. Contrast levels here are very high, with the black background and white foreground flame burst, making it almost monochromatic. The image is interesting in that such a large proportion of it is comprised of black, as such it could be expected that this image would compress well using RLE-type compression schemes.

5.1.6 Image 6 – "Sparrow"

Figure 5.1.6 is a greyscale image of a common sparrow perched upon a fence. The image itself is very sharp in that it has excellent edge definition and the background of the image has been blurred, causing the subject to stand out. Both the foreground and background of this image are interesting – the background is devoid of any focus and does instead have a number of gradients from dark to light spread across it; the foreground sparrow has a lot of detail on its head and breast, with much of this detail being regular. The fence railing also has good regularised texturing. Although this image is highly detailed, there is a lot of self-similarity in the detailed areas of the image.

5.2 Compression Technique

Each of the six images has undergone quadtree compression using the developed software, with initial 8x8 pixel blocks at the root of the tree, and being allowed to descend to 4x4 pixel blocks and finally 2x2 pixel blocks. At each level of the quadtree, the number of successful NxN covers was recorded. The fitness function allows a total margin of error to exist within a given block and this error is defined as the total of a block's individual pixel's (in the compressed image) variation from the equivalent pixel in the original image. In other words, if a 2x2 pixel block comprising 4 pixels total has an fitness error of 17 then, on average, each pixel in that block is just 4.25 linear greyscale shades away from their original image values. In order to make fitness error levels comparable across differing matrix sizes, the basic fitness value applies to a standard 2x2 block and is extrapolated when dealing with larger sizes – e.g. effectively 120 error level are allowed in a 4x4 block because it compromised 4 discreet 2x2 blocks (with an allowance of 30 each). The higher the fitness error value, the more tolerant of image quality loss the fitness evaluation function becomes.

Figure 5.1.1: Image 1 "Jan" (Original)

Figure 5.1.2: Image 2 "Prebend's Bridge" (Original)

Figure 5.1.3: Image 3 "Snowtrees" (Original)

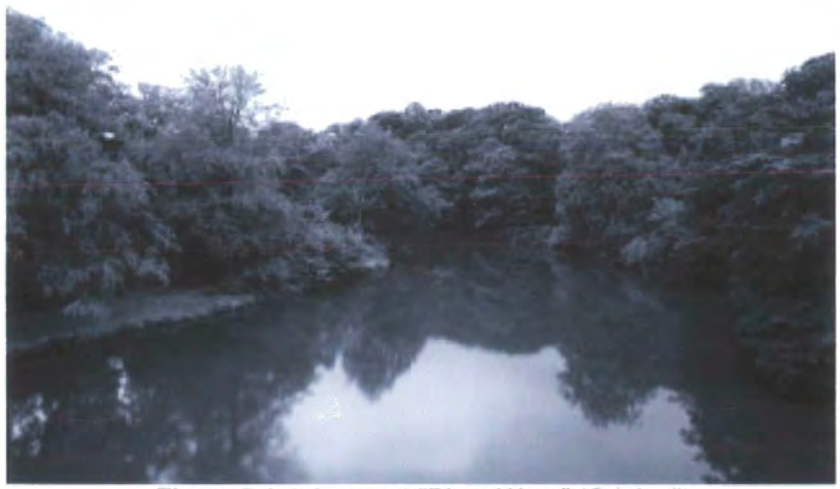


Figure 5.1.4: Image 4 "River Wear" (Original)

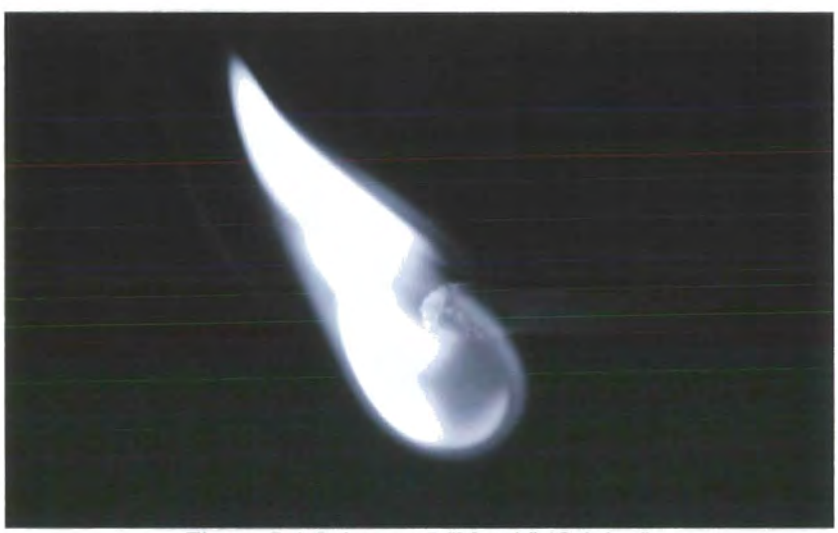


Figure 5.1.5: Image 5 "Match" (Original)

Figure 5.1.6: Image 6 "Sparrow" (Original)

5.3 Compression Results at Fitness 30

Figures 5.3.1-5.3.6 contain all six trial images when a fitness level of 30 has been applied to the quadtree compression, these should be compared with the original six images shown in figures 5.1.1-5.1.6. A fitness level of 30 means that in each 8x8 block, the sum of all the pixel value errors must not exceed 30 at a given level of the quadtree. This level was picked as initial results indicated it offered a good balance of image quality and compression level for a direct comparison with other compression techniques – results presented later in this chapter include images compressed with fitness levels between 7 and 120. Clearly a higher fitness level/tolerance is more forgiving of image errors whilst conversely with a lower tolerance. The filesize results obtained from the compression are shown in figures 5.3.7 and 5.3.8.

Figure 5.3.1: Image 1 "Jan" with QT compression applied (fitness 30)

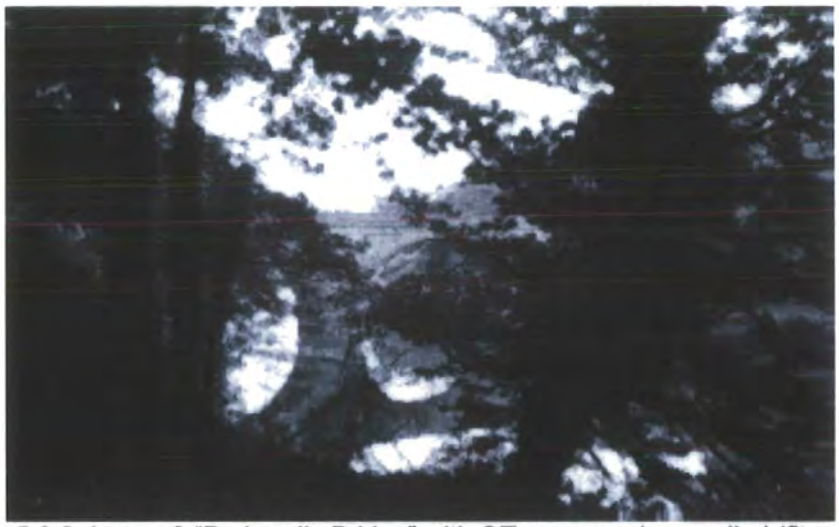


Figure 5.3.2: Image 2 "Prebend's Bridge" with QT compression applied (fitness 30)

Figure 5.3.3: Image 3 "Snowtrees" with QT compression applied (fitness 30)

Figure 5.3.4: Image 4 "River Weir" " with QT compression applied (fitness 30)

Figure 5.3.5: Image 5 "Match" with QT compression applied (fitness 30)

Figure 5.3.6: Image 6 "Sparrow" " with QT compression applied (fitness 30)

Image	Filesize (Kilobytes)
1	32.53
2	26.96
3	27.92
4	11.86
5	6.27
6	7.82

Figure 5.3.7: Table of filesizes for each image when QT compressed at fitness 30

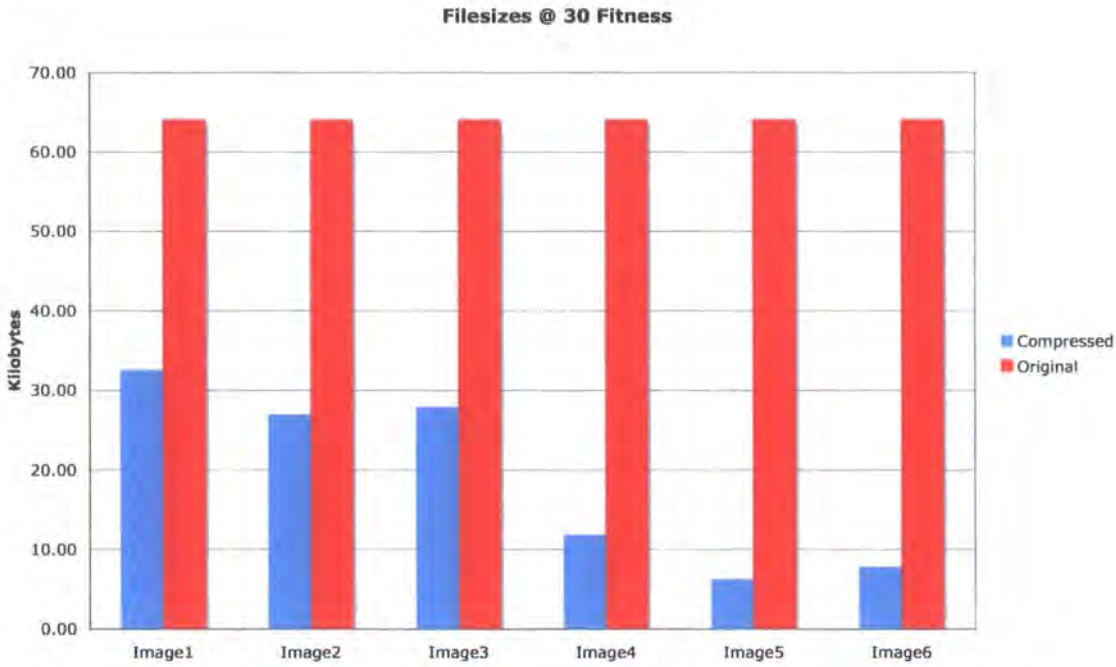


Figure 5.3.8: Filesizes for each image versus the original uncompressed size

5.4 Blocksize 'Levels' and Resultant Filesize

It can be seen that there is a large degree of variation in the quadtree compression characteristics and the resultant filesizes for the six selected images. Given an initial filesize for each (uncompressed) of 64 kilobytes, the least compression for the static fitness is shown in image 1, with just under 2:1 compression achieved, whilst the best is displayed in image 6, with just over 10:1 compression. There is also a large difference in terms of the construction of the six images from blocks at each of the 3 layers of the quadtree. Image 5, at this fitness level, can be almost completely represented at the top layer of the tree – 8x8 pixel blocks – whilst image 1 hardly contains any of these 8x8 pixel blocks.

In quadtree compression, the resultant compressed image is stored as a series of transforms. If an image can be adequately represented using larger blocks of pixels per transform, then fewer transforms are required to store the entire image and this results in a smaller filesize. With reference to figure 5.4.1, this can clearly be seen with image 5, where we see that most of the

image is using the largest blocksize allowed – 8x8. Conversely, image 1 requires smaller block sizes to match the fitness criteria and therefore requires many more transforms to store all the 2x2 block mappings, resulting in significantly less compression.

Comparing the figures 5.3.7 (filesize) and 5.4.1 (block construction) side-by-side shows the expected correlation between a compressed image's quadtree blocksize distribution and the resultant filesize. However, in images 2 and 3 it can be seen that although image 3 uses more 8x8 blocks in its compressed representation that image 2, it has a larger compressed filesize. This is clearly because image 2 comprises a larger number of 4x4 blocks that image 3. Image 3 relies on the largest number of 2x2 blocks of any of the images to represent some of its finer details and this has more than offset the levels of 8x8 blocks present.

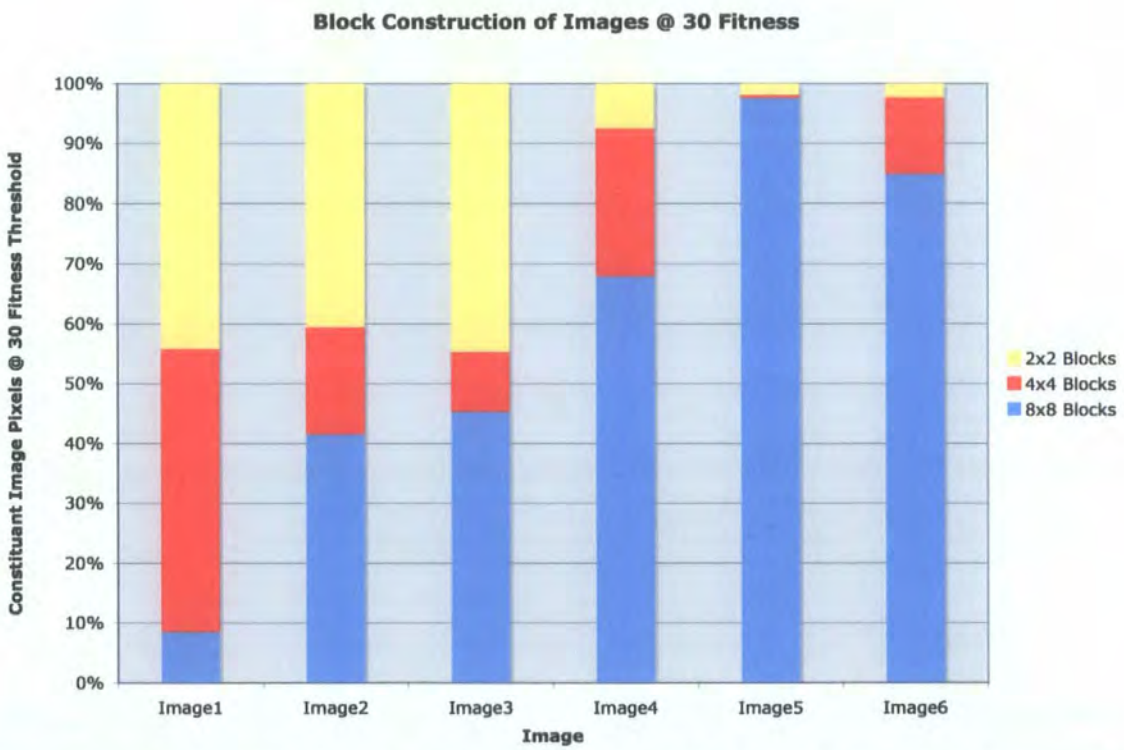


Figure 5.4.1: The percentage of each image's construction from each of the 3 block sizes used

5.5 Fitness

Although it is extremely useful to compare the original and compressed variants of an image by trying to observe differences – indeed passing this test sufficiently well is the final aim of an image compression method – it is also useful to perform a more structured and less subjective fitness analysis.

Using our quadtree method, the 2x2 block stage is the final stage to represent areas that could not be covered adequately by the previous 8x8 and 4x4 stages. Therefore, the fitness assessment function has not been used at this final stage to assess whether that stage's best coverage solution for an area is good enough. It is useful, therefore, to compare the fitness of an entire compressed image to that of the original and use this to help us determine how successful the compression has been on each of the six trial images. The table in figure 5.5.1 shows the fitness result for each of the six images compressed using fitness 30.

Image	Total Pixel Shade Errors	Mean Error Shade / Pixel
1	531175	8.30
2	1471045	22.99
3	1253921	19.59
4	439122	6.86
5	366536	5.73
6	403473	6.30

Figure 5.5.1 – Calculated fitness across the six quadtree-compressed images

The table shows that, at fitness 30, we see relatively consistent performance for four of our six images, and that the similarity compared to their originals is high – on average, each pixel differs by just 6 linear greyscales when compared to the original. However, images 2 and 3 show significantly higher levels of error when compared to the originals. This could be for a number of reasons:

i) Referring back to figure 5.4.1, neither image 2 nor image 3 are using 2x2 blocks (the last stage of the quadtree) for greater than 50% of their construction. It is possible,

therefore, that there is an issue with the fitness function that is stopping problem areas within the image being passed down to lower levels of the quadtree.

ii) That the images are not exhibiting sufficient levels of self-similarity to perform well in the quadtree compression algorithm. This would mean that although problem areas are being passed to lower levels of the quadtree, even the 2x2 block level cannot represent these areas sufficiently well to result in a high fitness result.

5.5.1 Fitness Issues with Images 2 and 3

If it were the case that problem (i) was the cause of the poor results with images 2 and 3 then it would be reasonable to expect that an image compressed using just the 2x2 level of the quadtree would also give a poor result. It would also be reasonable to expect that when the images were compressed with a more stringent intra-level fitness value would also exhibit these consistently poor results. It is worth noting that the mean error per pixel rate for both images is higher than that permitted by a fitness level of 30 in that the quadtree fitness function. Such a fitness level would only allow a total of 30 pixel errors across a 2x2 block, or approx 7.5 per pixel. This suggests that problem areas within both of these images are indeed reaching the lowest 2x2 block levels of the quadtree but even then are exhibiting low-fitness representation and therefore points to the cause being problem (ii). Once compressed using more stringent fitness levels of 15 and 7, together with using a 2x2 block pattern exclusively, image 3 gives the results in figure 5.5.1.1.

QT Fitness	Total Pixel Shade Errors	Avg. Error Shade / Pixel
30	1253921	19.59
15	1251683	19.56
7	1250310	19.54
N/A (2x2 only)	1216246	19.00

Figure 5.5.1.1 – compression results for image 3 at higher fitness levels and 2x2 only

It can be seen that the quality of the compressed image 3 does not improve at more stringent fitness levels and when compressed solely at the 2x2 block layer there is not significant reduction in the level of average pixel error. This suggests that, using this implementation at least, images 2 and 3 do not compress favourably using the quadtree process.

5.6 Analysis of Images at Different Fitness

Although we have examined all six trial images at a fitness level of 30, it is important to investigate how image quality and the level of compression achieved varies with more and less stringent fitness levels in the quadtree compression. For this investigation, rather than examine all six images, images 1, 3 and 6 have been selected. Image 1 is interesting because, at a fitness level of 30, hardly any 8x8 blocks are used in its construction (see figure 5.4.1), suggesting that image quality would fall off markedly should a larger proportion of 8x8 blocks be used in its construction. Image 3 is interesting due to the discussion above – it is one of two images from the six originals that do not appear to perform well at fitness 30 and below. Finally, image 6 can be seen to be a highly detailed image yet delivers a very high level of compression at fitness level 30 and manages to use a large proportion of 8x8 blocks in its construction.

5.6.1 Compression Results at Varying Fitness

Figures 5.6.1.1 – 5.6.1.3 give the statistical results for images 1,3 and 6 at five fitness levels – 7, 15, 30, 60 and 120. Figures 5.6.1.4-5.6.1.18 illustrate images 1,3 and 6 at these compression levels and also include the original image for reference.

Threshold	Filesize	Ratio	Error/Pixel
7	52.1	1.23	6.48
15	49.25	1.30	6.68
30	32.53	1.97	8.30
60	10.21	6.27	17.02
120	5.12	12.50	23.86

Figure 5.6.1.1 – Image 1's compression results at varying quadtree fitness levels

Threshold	Filesize	Ratio	Error/Pixel
7	35.82	1.79	19.54
15	34.35	1.86	19.56
30	27.92	2.29	19.59
60	12.55	5.10	19.74
120	5.34	11.99	20.16

Figure 5.6.1.2 – Image 3's compression results at varying quadtree fitness levels

Threshold	Filesize	Ratio	Error/Pixel
7	17.51	3.66	5.18
15	14.48	4.42	5.39
30	7.82	8.18	6.30
60	5.13	12.48	7.59
120	5.12	12.50	8.14

Figure 5.6.1.3 – Image 6's compression results at varying quadtree fitness levels

The results show that image 1 has the lowest compression levels at 30 fitness, although these do improve to match those of image 3 at fitness levels greater than 30, though this comes at the expense of fitness compared to the original image. Image 6 exhibits excellent compression levels at all fitness levels, together with consistently low fitness scores.

In addition to the above results, generated by application of a fitness function to the compressed image in comparison to the original, it is obviously important that a compressed image, when viewed by the human eye, appears as close to the original image as possible. Whilst this is clearly a subjective test, a lossy compression algorithm aims to maximise compression whilst minimising the impact to the perceived image quality – as shown previously, this is often at the expense of technical accuracy as minor variations in the image are difficult to detect. The compression results for images 1, 3 and 6 that complement those in the tables in figure 5.6.1.1 – 5.6.1.3 are given in figures 5.6.1.4 – 5.6.1.18.

Figure 5.6.1.4: Image 1 "Jan" with QT compression applied (fitness 7)

Figure 5.6.1.5: Image 1 "Jan" with QT compression applied (fitness 15)

Figure 5.6.1.6: Image 1 "Jan" with QT compression applied (fitness 30)

Figure 5.6.1.7: Image 1 "Jan" with QT compression applied (fitness 60)

Figure 5.6.1.8: Image 1 "Jan" with QT compression applied (fitness 120)

Figure 5.6.1.9: Image 3 "Snowtrees" with QT compression applied (fitness 7)

Figure 5.6.1.10: Image 3 "Snowtrees" with QT compression applied (fitness 15)

Figure 5.6.1.11: Image 3 "Snowtrees" with QT compression applied (fitness 30)

Figure 5.6.1.12: Image 3 "Snowtrees" with QT compression applied (fitness 60)

Figure 5.6.1.13: Image 3 "Snowtrees" with QT compression applied (fitness 120)

Figure 5.6.1.14: Image 6 "Sparrow" with QT compression applied (fitness 7)

Figure 5.6.1.15: Image 6 "Sparrow" with QT compression applied (fitness 15)

Figure 5.6.1.16: Image 6 "Sparrow" with QT compression applied (fitness 30)

Figure 5.6.1.17: Image 6 "Sparrow" with QT compression applied (fitness 60)

Figure 5.6.1.18: Image 6 "Sparrow" with QT compression applied (fitness 120)

Image 1's results for fitness levels 7 and 15 compare very favourable with the original bitmap – there seems to good preservation of detail and toning. At fitness level 30, it is clear that there is a slight loss of detail around both the dog's fur and on the leaves in the background. There is also a slight compression in tonal representation, especially in the darker areas of the image. Fitness level 60, therefore, leads to a further drop in visual quality and areas of the dog are starting to get lost amongst the background. At fitness level 120, the quadtree compression has resulted a mosaic-like approach due to the prevalence of 8x8 blocks within the image and it is just about possible to make out the image subject.

With fitness level 7 for image 3, the one thing that is noticeable when compared to the original image is the tonal compression in both the sky and the foreground snow areas – it is possible to see the tonal steps in these areas rather than the continuous tone appearance of the original. Apart from this, fitness level 7 offers a very good representation of the original. The same tonal compression appears in fitness level 15 but again, this image is otherwise visually very similar to the original. Fitness level 30, as with image 1 above, is when we start to see deterioration in the detail within the compressed image. Some of the branches that hang down from the tree in the top of the frame are omitted and the area immediately below the fence has lost detail. Although the rendering of the trees continues to be good, the left-hand trees are beginning to show blockiness in their branch details. Fitness level 60 has exacerbated the issues seen with fitness level 30 both in terms of detail loss and visible blocks appearing in the tree branch areas of the image. Finally, fitness level 120 has even less detail and more blockiness, though it is possible to make out the nature of the image itself.

In terms of compression results for image 6, fitness level 7 has again deliver a very good quality image, albeit with compressed tonal representation in areas with gradients. It is notable that the image doesn't appear as sharp and well defined as the original although it remains a very good likeness. Once again, there is little to separate fitness level 15 from 7. Fitness level 30 sees the introduction of the trademark blocks in the breast of the sparrow, though these aren't too severe.

Fitness level 60 is just sufficiently good to identify the nature of the subject through the increased blockiness whilst fitness level 120 offers quite a poor rendition of the original image.

Overall, the quality of the quadtree compressed images is reasonable, with fitness levels 7 and 15 offering good image quality at the expense of compression; fitness levels 60 and 120 offering higher compression but much reduced image quality and fitness level 30 being an compromise between the two extremes. It is also interesting to observe that, despite the higher pixel shading error counts seen in image 3, the visual appearance when compared to the original image is extremely favourable.

5.6.2 Changes in Blocksize Distribution at Different Fitness Levels

In the original assessment of all six of the trial images, figure 5.4.1 showed the construction of each of the compressed images in terms of the 3 block sizes / levels of the quadtree. Figures 5.6.2.1 – 5.6.2.3 show the quadtree construction (by number of pixels represented by particular block sizes) for image 1, 3 and 6 respectively.

These three charts illustrate a marked difference in characteristics between the three images. Image 1 makes heavy use of 2x2 blocks for all but fitness levels 60 and 120, and makes good use of 4x4 blocks at the expense of some 2x2 blocks at fitness levels 30 and 60. Image 3 uses less 2x2 blocks that image, apart from in the cases of fitness levels 60 and 120, where image 1 actually uses less. Image 6 makes very heavy use of 8x8 blocks, even at fitness level 7 nearly 75% of pixels in the compressed image are represented in this way. In some ways, Image 3 behaves as a happy medium between images 1 and 6, although it is important to remember that image 3's pixel error levels remain high throughout.

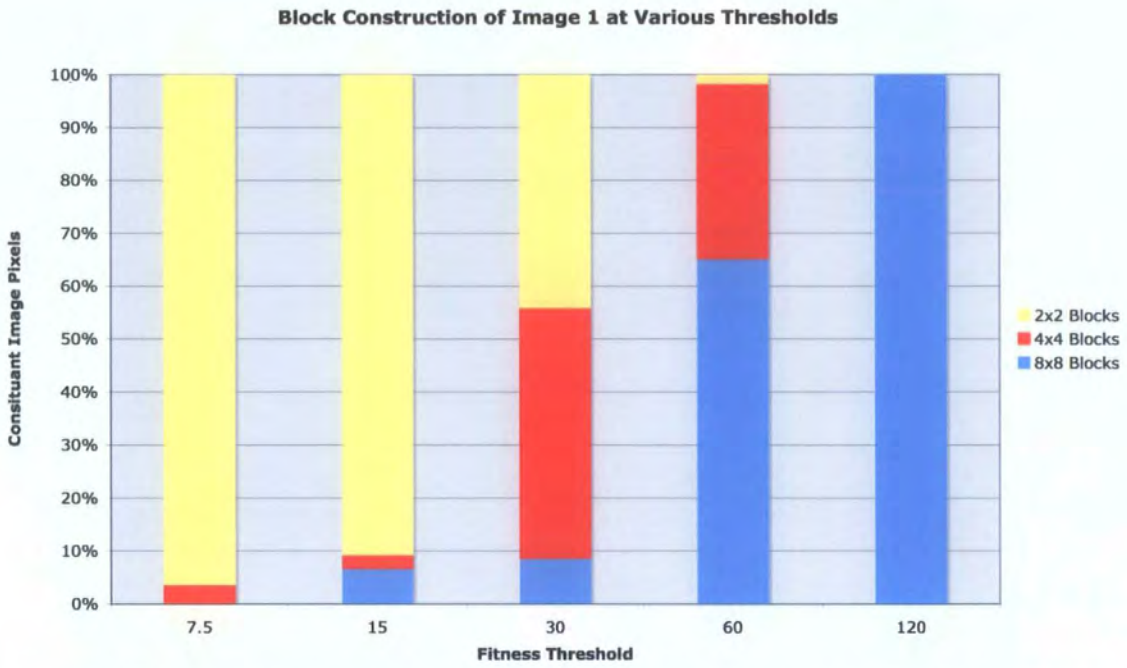


Figure 5.6.2.1: Construction of Image 1 from each of the 3 block sizes at different fitness levels

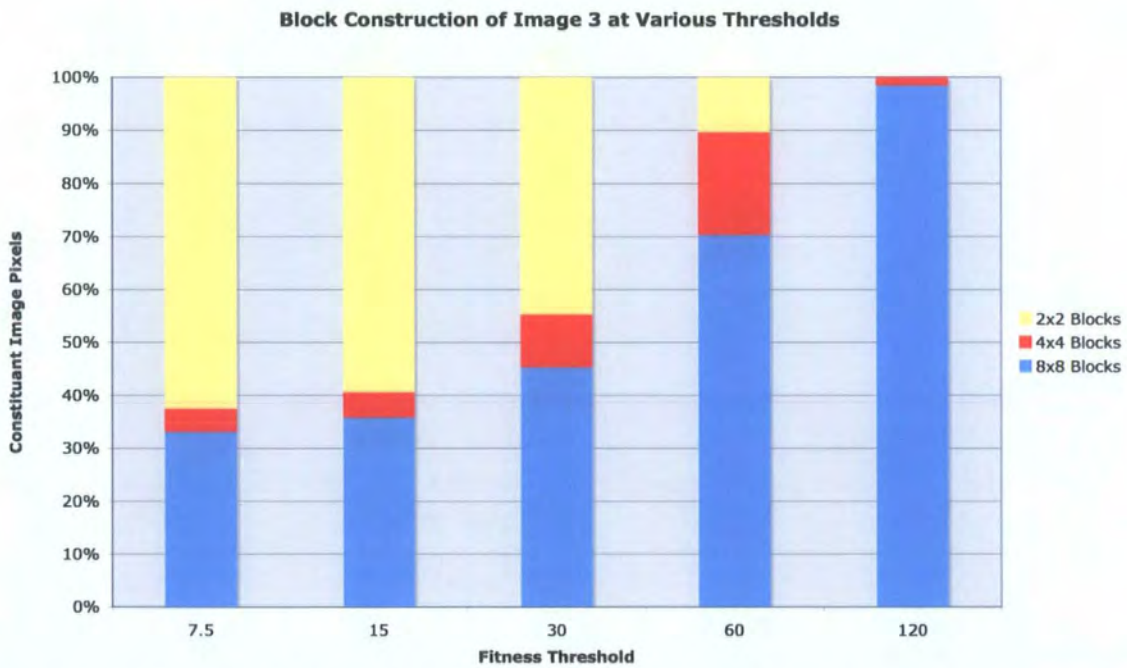


Figure 5.6.2.2: Construction of Image 3 from each of the 3 block sizes at different fitness levels

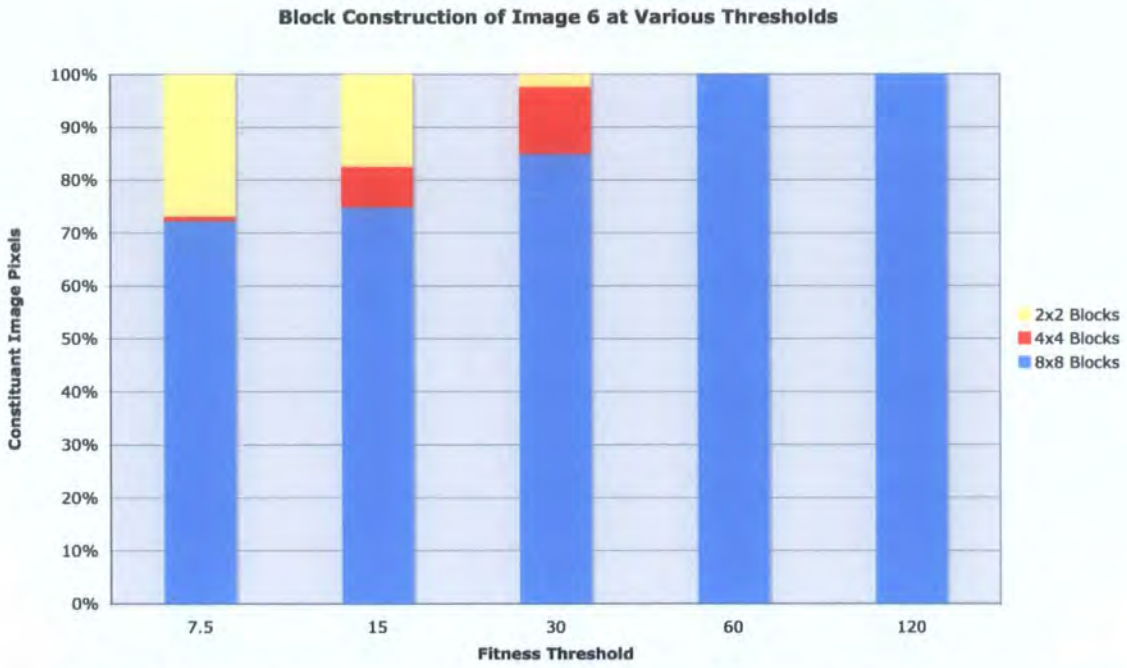


Figure 5.6.2.3: Construction of Image 6 from each of the 3 block sizes at different fitness levels

Image 6's domination by 8x8 blocks, and the fact that both fitness levels 60 and 120 produce effectively the same result in terms of quadtree block distribution, suggests that this image would benefit from a large blocksize being available as the root of the quadtree – perhaps 16x16 and possibly 32x32 pixel blocks.

It is also interesting to note that, as fitness tolerance to error increases, the usage of 4x4 blocks by the three images at first increases and then decreases, whilst the usage of 2x2 blocks decreases continually, as does the usage of 8x8 blocks. The likely reasoning for this is that at first, the fitness level demanded is very high and as such the majority of pixels are likely to use 2x2 blocks – those that do not may well represent block colour areas and as such 8x8 blocks will still produce good results. However, as the fitness criteria become more tolerant of pixel error, more blocks that used to be only satisfied at the 2x2 block level can be represented by 4x4 blocks more efficiently, hence the redistribution towards the middle level of the tree. Finally, when the fitness criteria become highly tolerant of error, nearly all blocks can be presented by the root 8x8

block level and as such relatively few areas require either 2x2 or 4x4 block representation. This can be seen in the block distribution charts for images 1,3 and 6 shown in figures 5.6.2.4 – 5.6.2.6.

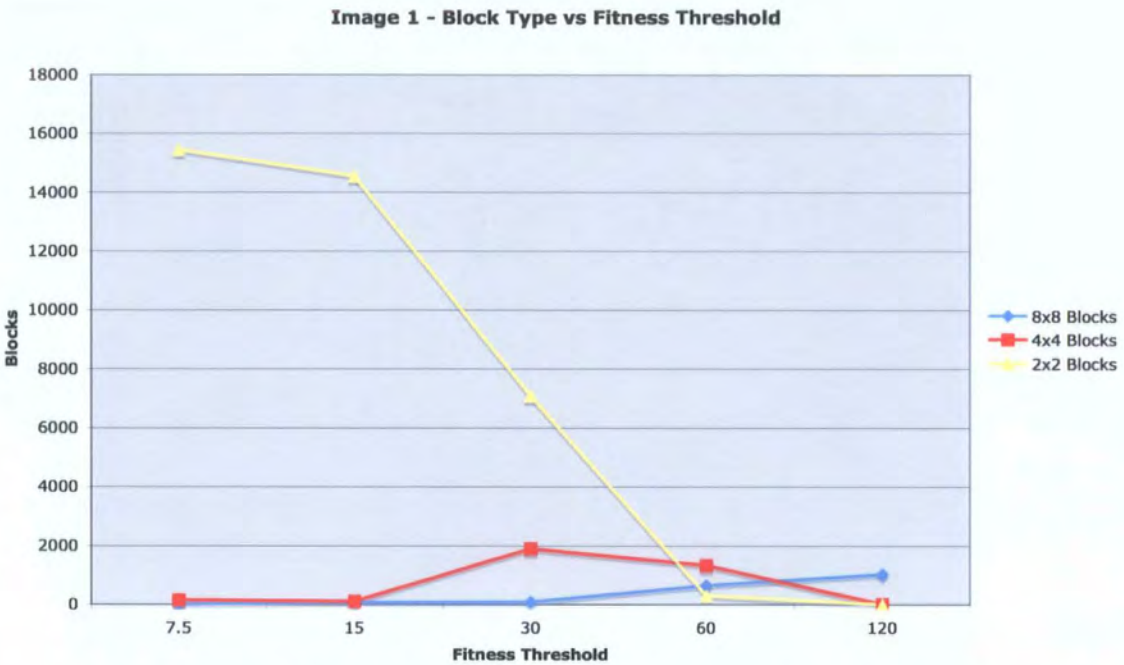


Figure 5.6.2.4: Block distribution at each fitness level for Image 1

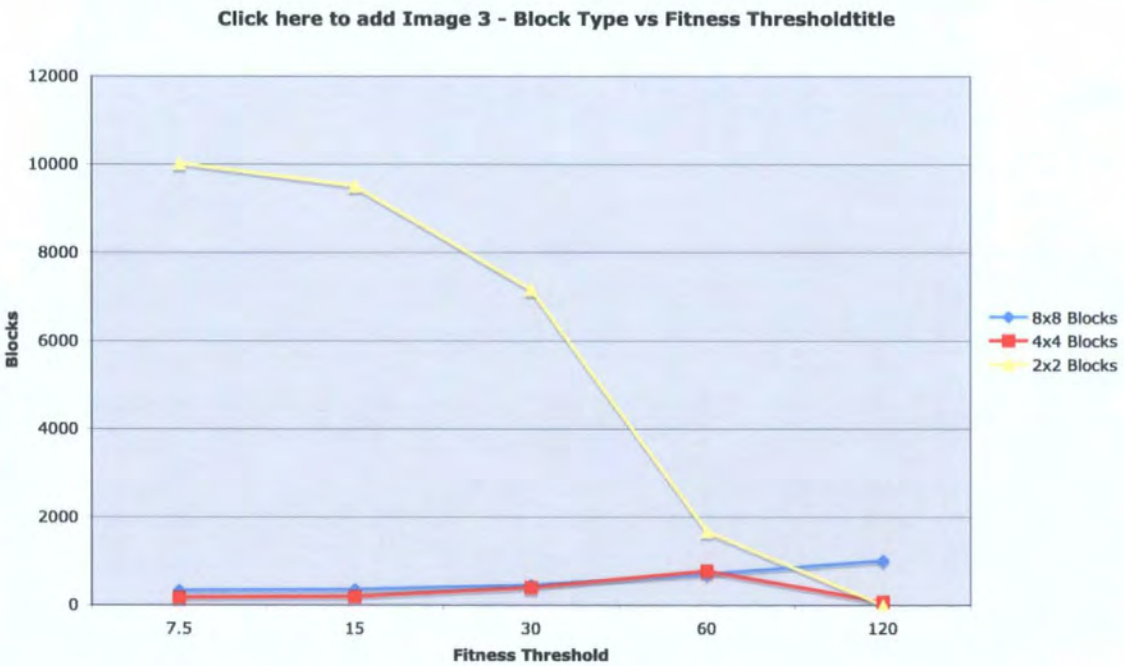


Figure 5.6.2.5: Block distribution at each fitness level for Image 3

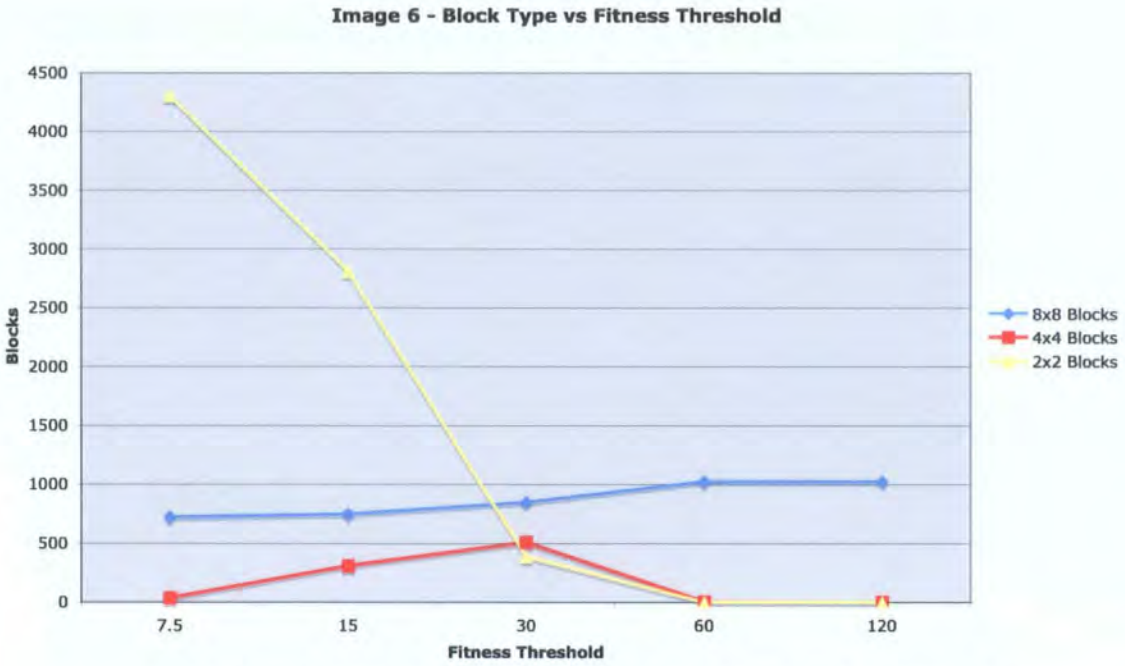


Figure 5.6.2.6: Block distribution at each fitness level for Image 6

5.7 Comparison with JPEG

In order to allow comparison between the popular and established JPEG compression standard and the quadtree algorithm, images 1, 3 and 6 were also compressed using JPEG at different quality/fitness levels. Although these fitness levels aren't directly comparable to those used during the quadtree process, they do serve to show the spectrum of quality versus compression levels achievable with a standard JPEG implementation. Just as with the fitness levels for quadtree, high JPEG fitness levels increase the tolerance of the lossy compression system and deliver lower image quality as a result. Figures 5.7.1 – 5.7.3 contain the results in terms of fitness and filesize for different levels of JPEG compression applied to Images 1, 3 and 6. Comparisons in the performance in terms of image quality (fitness) and compression levels (filesize) between JPEG and quadtree for all three images are given in the charts in figures 5.7.4 – 5.7.6. When analysing the charts, it should be noted that smaller values represent higher algorithm performance for both fitness and filesize.

JPEG Level	Filesize	Total Pixel Errors	Errors/Pixel
1	56	17222	0.26
25	20	392076	6.12
50	12	524107	8.18
75	8	666168	10.40
100	4	1558717	24.35

Figure 5.7.1 JPEG Compression Results for Image 1

JPEG Level	Filesize	Total Pixel Errors	Errors/Pixel
1	40	5892	0.09
25	16	277896	4.34
50	12	458284	7.16
75	8	599279	9.36
100	4	1303789	20.37

Figure 5.7.2 JPEG Compression Results for Image 3

JPEG Level	Filesize	Total Pixel Errors	Errors/Pixel
1	28	15098	0.23
25	8	114629	1.79
50	8	156376	2.44
75	4	221002	3.45
100	4	819249	12.80

Figure 5.7.3 JPEG Compression Results for Image 6

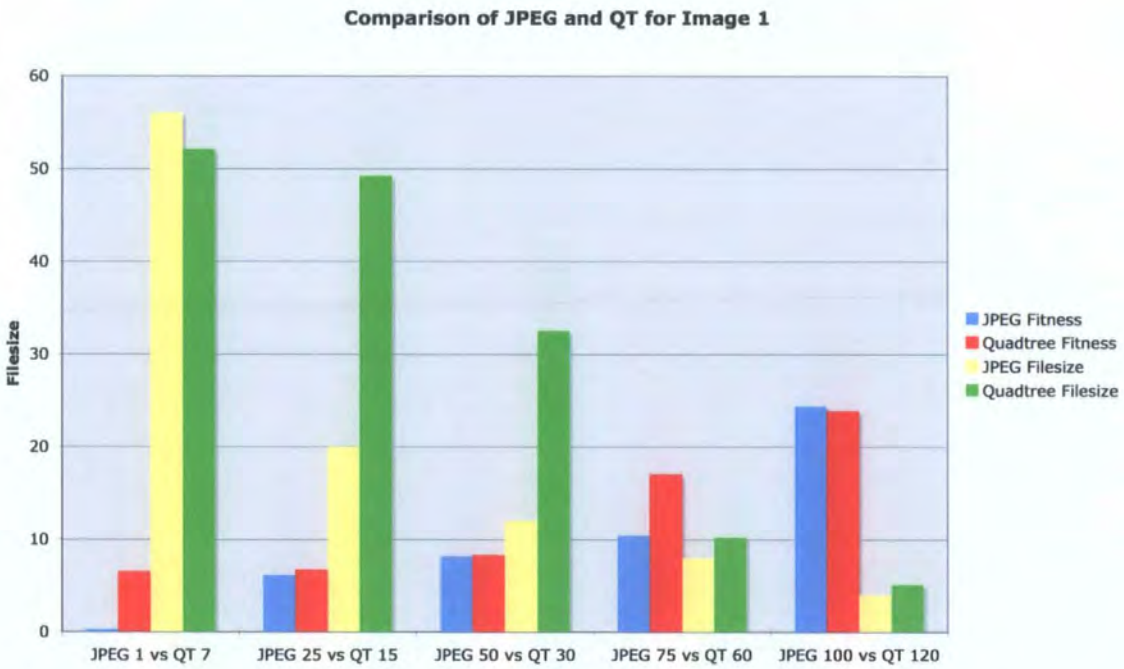


Figure 5.7.4: Comparison of JPEG and QT for Image 1

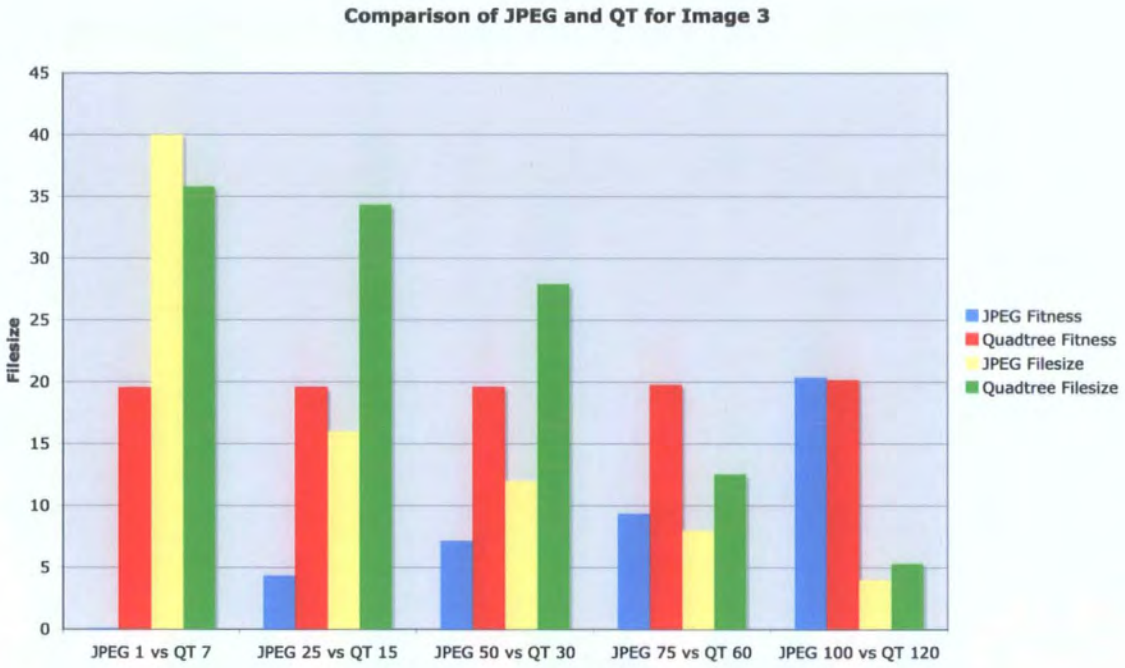


Figure 5.7.5: Comparison of JPEG and QT for Image 3

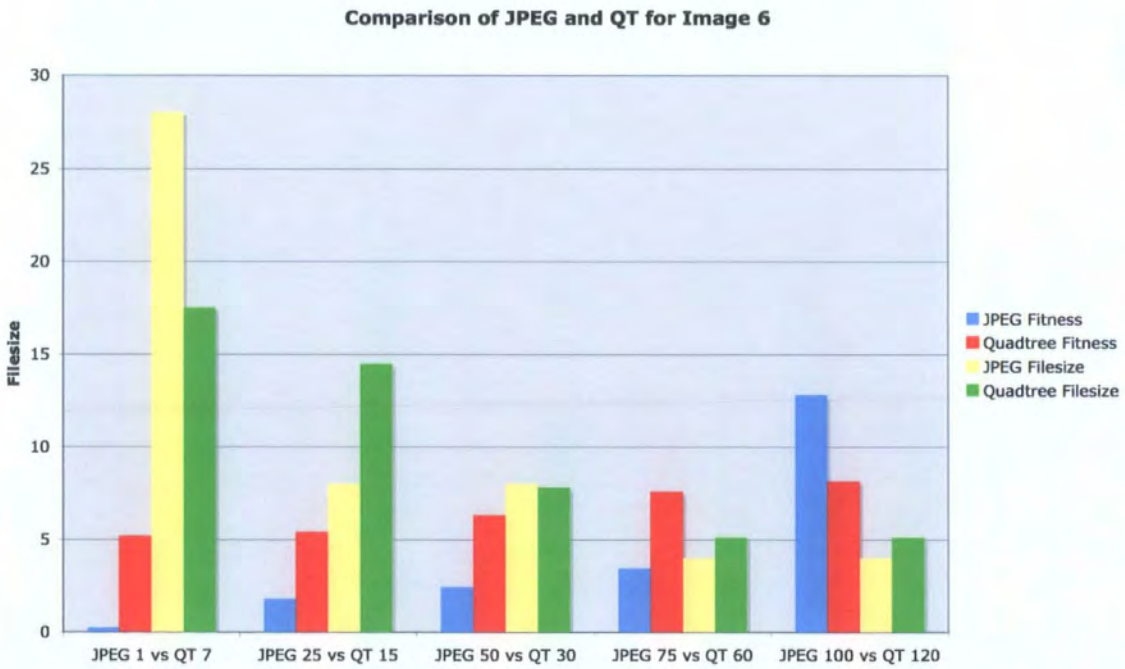


Figure 5.7.6: Comparison of JPEG and QT for Image 6

Both the tables of results for the JPEG compression of the three images, together with the charts, generally show that JPEG gives better compression for a given image quality and similarly, smaller filesizes for a given image quality. The JPEG compressed images used to generate the JPEG results from each of the 3 images are presented in figures 5.7.4 – 5.7.21.

Although JPEG compression is superior to quadtree here, at least according to the fitness function, it is important that we attempt to compare the images to see if this is borne out visually.

For image 1, JPEG fitness level 1 is superior to that of quadtree fitness level 7, as suggested by the associated chart. However, JPEG fitness level 25 and quadtree fitness level 15 prove a very equal match visually, as also suggested by the chart. This trend continues through to JPEG fitness level 50 and quadtree fitness level 30. Quadtree fitness level 60 does lose out to JPEG fitness level 75 but the final images from quadtree 120 and JPEG 100 show that the quadtree example at this level is superior to a much greater extent than is suggested by the chart.

Image 3's visual results show that whilst, as shown previously, the quadtree fitness level 7 compression results in tonal compression in the sky and foreground, the JPEG fitness level 1 image avoids this and as a result is visually superior. However, whilst the chart for image 3 suggests a similar result for the next fitness step, quadtree at fitness level 15 appears to deliver a more visually appealing result than JPEG fitness level 25 as a result of the JPEG's artefacts and loss of detail within the branches of the trees. This theme continues for the quadtree fitness level 30 and JPEG fitness level 50, though JPEG does have a slight edge at quadtree fitness level 60 and JPEG fitness level 75 before quadtree 120 delivers a significantly better image than JPEG 100.

With image 6, it is clear that JPEG delivers a superior image to quadtree in each comparison, with the exception of quadtree 120 / JPEG 100. The background of the sparrow once again suffers due to the quadtree tonal compression. The only consolation for the quadtree result is that

it continues to maintain a good degree of edge sharpness as this tails off slightly in the JPEG versions.

On each of the three images, JPEG level 1 gives a compressed image that is virtually indistinguishable from that of the originals. Unlike quadtree, JPEG doesn't exhibit tonal compression at large filesizes. It is also interesting that given their very similar filesizes, JPEG 100 always results in a lower image quality than quadtree with a 120 fitness level. This result is borne out when the results are compared visually.

From the above comparisons, it should be clear that JPEG is producing results superior to those of the quadtree implementation, although it is interesting to see the relative performance of the quadtree improves as lower filesizes are required.

Figure 5.7.7: Image 1 "Jan" with JPEG compression applied (level 1)

Figure 5.7.8: Image 1 "Jan" with JPEG compression applied (level 25)

Figure 5.7.9: Image 1 "Jan" with JPEG compression applied (level 50)

Figure 5.7.10: Image 1 "Jan" with JPEG compression applied (level 75)

Figure 5.7.11: Image 1 "Jan" with JPEG compression applied (level 100)

Figure 5.7.12: Image 3 "Snowtrees" with JPEG compression applied (level 1)

Figure 5.7.13: Image 3 "Snowtrees" with JPEG compression applied (level 25)

Figure 5.7.14: Image 3 "Snowtrees" with JPEG compression applied (level 50)

Figure 5.7.15: Image 3 "Snowtrees" with JPEG compression applied (level 75)

Figure 5.7.16: Image 3 "Snowtrees" with JPEG compression applied (level 100)

Figure 5.7.17: Image 6 "Sparrow" with JPEG compression applied (level 1)

Figure 5.7.18: Image 6 "Sparrow" with JPEG compression applied (level 25)

Figure 5.7.19: Image 6 "Sparrow" with JPEG compression applied (level 50)

Figure 5.7.20: Image 6 "Sparrow" with JPEG compression applied (level 75)

Figure 5.7.21: Image 6 "Sparrow" with JPEG compression applied (level 100)

5.8 Temporal Video Compression

Although the quadtree compression is able to handle still image compression, modern video compression requires the exploitation of temporal (inter-frame similarities) compression in addition. For the quadtree algorithm to be able to compress the i-frames associated with this, they must first be generated by minimising temporal redundancy. A 900-frame video sequence of movie footage, containing numerous scene-changes and panning motions in addition to periods of high temporal redundancy, was processed to generate i-frames for compression. The levels of

redundancy removed are shown in the chart in figure 5.8.1, where, for each frame, the % of change from the preceding frame is graphed.

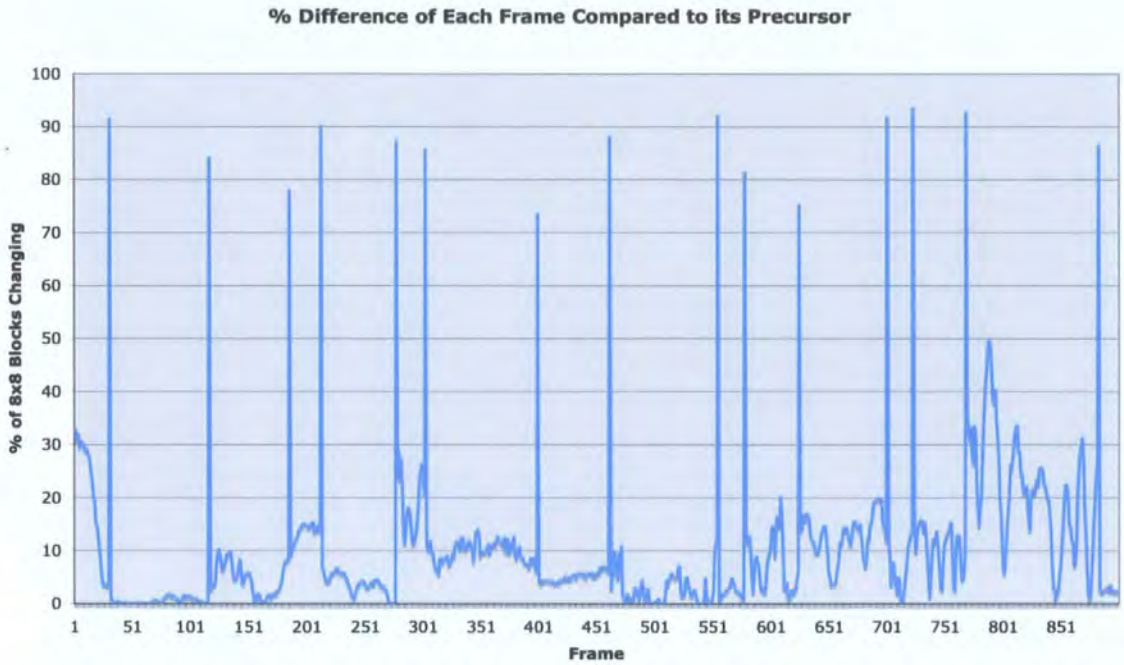


Figure 5.8.1: Percentage Difference of Each Frame Compared to its Precursor

As the frames progress, the amount of temporal redundancy varies dependent on the nature of the video and the relationship between the current and previous frames. The complete angle and scene changes within the sequence are represented by the high peaks in the 80-100% region, whilst the 300-450 frame range values represent panning action across the scene and the last 125 frames also have lower redundancy due to large amounts of on screen movement.

The mean average temporal redundancy for this particular sequence is 89.78%. On average, only 10.22% of the 8x8 blocks within frame changes compared to its precursor. Exploitation of this would be maximised if the relatively regular scene-changes could be detected and stored as i-frames rather than p-frames within compressed output. A sample of 10 non-sequential frames (frames 100 – 1000 at 100-frame intervals) shows that the average QT compression level for a fitness threshold of 30 is a consistent 6.5:1.

It is, of course, possible to remove less or more inter-frame differences from this same sequence – removing less simply reduces the level of temporal compression achieved, whilst removing more can lead to visually important but subtle inter-frame changes being discarded. When applied to this sequence this was seen to lead to erratic movement in scenes with large levels of motion within a particular area of the frame and was quite distracting. The erratic movement and artefacts are caused by i-frames that contain too little information about inter-frame changes being overlaid on the previous frame – this results in misalignment and visual duplication occurring in the areas of rapid movement.

Assuming that the algorithm can deliver an approximate 6.5:1 compression rate in terms of temporal redundancy, and a further 10:1 through quadtree spatial redundancy for i-frame compression, this suggests a video compression performance of approximately 65:1 whilst maintaining reasonable image quality (quadtree fitness level 30 is assumed, since this gives an intermediate quality level).

5.9 Summary

In order to test the performance of the quadtree implementation across a wide range of images, the six images selected each exhibited different characteristics and challenges. Of these initial six images, three were subjected to further levels of quadtree compression. For each of these three images, at each level of prescribed fitness, an evaluation of the compression performance in terms of both visual quality and filesize was carried out, together with subsequent comparison to JPEG at similar filesizes. Finally, the individual construction of the compressed images was examined with a view to understanding how the various blocksizes used was impacted as compression levels increased and reduced.

6.0 Conclusion

This chapter contains an assessment of the quadtree compression system, the results obtained and what has been learnt during the development of the system. With reflection on these, further enhancements and future areas of research are also highlighted.

A major problem with digital imaging is the amount of storage required for images, especially high-resolution photographs. Whilst generic data compression can go some way to reducing the sizes of images whilst they are stored or transmitted, they amount of compression achieved is quite low. Content-specific algorithms, such as JPEG, are able to achieve a much higher level of compression through the use of lossy techniques – these algorithms 'throw away' data from the original image in such a way that it isn't apparent to a human viewer.

A fractal is commonly understood to be a complex, colourful image, which is generated by the repeated iteration of a small, simplistic formula. The challenge of fractal image compression is that whilst it is easy to generate an image from a fractal formula, it is very difficult to generate the appropriate fractal formula for a given arbitrary image. The quadtree compression algorithm presented and evaluated in this thesis is an attempt to automate the search for a 'solution' to this problem.

6.1 Performance Against Thesis Objectives

The original objectives of this thesis were set out in chapter 1, with the intention that the remainder of this thesis would subsequently satisfy each objective. A review of these objectives, including the relative success that can be attributed to each, is given in 6.1.1 – 6.1.5.

6.1.1 Evaluate Fractal Image Compression Techniques

A thorough discussion of existing image compression techniques has been presented in chapter 2 of this thesis. This survey encompasses a wide range of image compression methods, from traditional lossless RLE encoding through lossy JPEG compression to developments in fractal image compression techniques.

6.1.2 Development of a Fractal Compression Software Toolset

Chapters 3 and 4 give details of the toolset developed to explore the capabilities of Fractal image compression, and quadtree-based Fractal image compression in particular. The toolset itself can compress arbitrary images using a basic quadtree implementation whilst allowing for the various quadtree compression characteristics and fitness target levels to be adjusted. The tools also allow decompression of these images and fitness evaluation between two arbitrary images using the same fitness function as the quadtree encoder.

6.1.3 Evaluate Quadtree Image Compression Using Developed Toolset

The developed toolset allows the compression characteristics of the quadtree algorithm to be evaluated in a number of ways. Using a standard image across compression trials, it has been possible to investigate how increasing or decreasing the fitness tolerance thresholds changes the level of compression achieved. Similarly, it is possible to see how the quality of the image increases as the compressed image is allowed to occupy more space, and whether the returns are diminishing or are simply linear, etc.

The toolset also offers insight into the quadtree construction used to represent an image at various compression levels – i.e. it is possible to observe how the blocksize distribution changes dependent on both the image being compressed and the level of fitness specified. It can also be

seen that some images compress much more readily than others, even when the fitness levels are changed to favour the less-compressible image.

These investigations have illustrated that whilst the quadtree scheme presented in this thesis can compress an arbitrary image quite successfully, there is scope for significant improvement – potentially including a move to a more flexible and dynamic partitioning scheme than quadtree as a basis for a fractal compression system – this is discussed further in 6.2.

6.1.4 Comparison of Quadtree Algorithm to JPEG and MPEG

One of the principal aims of this work was to develop a quadtree compression scheme and compare its performance, for both image and video-compression, to established compression systems. JPEG and MPEG, as discussed previously, are ideally suited as performance benchmarks, especially since the latter (MPEG) makes heavy use of the former (JPEG) in its approach.

The results presented in Chapter 5 show that, in the majority of instances, JPEG's performance is significantly superior to the quadtree algorithm for a given filesize/quality threshold. Whilst the quadtree implementation presented here cannot be regarded as superior to JPEG, its performance, given the relatively short development period, is still reasonable and certainly offers a solid platform to build upon as well as giving a taste of what might be possible with further development.

Although it is not yet capable of delivering results that can better JPEG for a given filesize in the general case, the results given in Chapter 5 indicate that this algorithm can in fact surpass JPEG as lower filesizes are required by a user / application. This suggests that quadtree compression could have an impact in low datarate image and video applications, such as video streaming across mobile telephone networks and the Internet. Indeed, whilst MPEG is widely accepted as

the de facto baseline video compression standard, other compression technologies, such as the proprietary RealNetworks video formats and the upcoming H.264 standard [HTTP://MPEG.ORG], continue to dominate the video streaming market.

6.1.5 Image Encoding Computational Performance

Using the current algorithm to encode an image, whether a p-frame or an i-frame, can take in excess of 20 minutes on an Intel Pentium-4 2.8GHz machine. The vast majority of computing time is spent in the domain / range comparison search function, with the amount of processing time required being dependent on a number of factors:

1) The initial blocksize of the quadtree affects the computation time in a number of ways. If a small initial blocksize is selected, then the domain/range search function's initial domain set is larger than that where a larger initial blocksize is selected. This means that areas of an image that had the potential to be covered by much larger blocks instead need to be covered by a number of smaller blocks and therefore require additional domain searches. However, if the fitness tolerance for a compression attempt is extremely demanding then it's possible that selecting a large initial blocksize would increase the overall processing time through its attempts to achieve the required fitness in the earlier stages of the quadtree.

2) The selection of permutations available for domain to range mappings can also greatly affect the image encoding time. For example, if the domain to range mapping search only attempts rotational mapping then the search will take less time than if the mapping search attempted to additionally vary the contrast and brightness to achieve a more optimal result. Whilst there are a number of transforms that can be performed on a domain in order to allow it to better cover a range, these come at the expense of

encoding time. Including additional transforms on the domain increase the likelihood of finding a more optimal domain to range mapping.

3) Given its extensive utilisation during the domain / range comparison search, the complexity of the fitness function greatly influences the overall performance of the quadtree implementation. The current fitness function is simplistic, and though more sophisticated metrics could be employed to aid in more accurate domain to range mapping, this would significantly increase encoding times.

In comparison to JPEG, the current encode performance of the quadtree algorithm is slow. Considering that the Fractal video compression requires that sequences using 25 frames/second can be compressed then it is essential that the computational efficiency of the algorithm be improved. Suggestions for improvements are covered in section 6.4.

6.2 Further Work

The development and testing of the quadtree toolset and its compression performance has highlighted that whilst the current implementation gives reasonable image compression performance, there are a number of areas which could be improved.

6.2.1 Improvements to the Toolset and Algorithms

Although the developed toolset allows for basic quadtree image/video encoding and decoding, there are a number of areas that could be improved to increase the efficiency of the compression achieved and the appearance of the compressed results:

a) The current toolset only allows for a limited quadtree size in that the initial blocksize is relatively small. This should be addressed so that the initial blocksize is the resolution of the target image itself.

b) Whilst the current fitness metric performs well, if greater compression results need to be achieved with this current implementation, even at the expense of a vast uplift in computation time, then other, more accurate metrics could be implemented. If efficiencies are made in other areas of the range / domain comparison routine then this could be a worthwhile and feasible improvement.

c) There is currently no post-processing of the encoded/decoded image, as such this can lead to visible blockiness where the neighbouring edges of neighbouring image blocks do not share a similar pixel shading. In figure 6.3.1, the section of the sparrow enlarged from image 6 shows how larger blocks have clearly visible edges as they attempt to recreate the detail in the bird's feathers. Post-processing can improve the appearance of images compressed using the quadtree method, as shown in figure 6.3.2:

Figure 6.3.1: Portion of Image 6 – "Sparrow" compressed output

Figure 6.3.2: Portion of Image 6 – "Sparrow" as in figure 6.3.1 but with basic post-processing manually applied.

6.2.2 Areas for Further Research

Although there are improvements that can be made to the tool itself, this investigation has highlighted other areas outside of the original scope that merit further analysis and investigation.

6.2.2.1 Use of Different Fractal Image Partitioning Methods

Quadtree is the simplest form of image partitioning in the field of fractal compression and relatively straightforward to implement. It is possible that using other schemes, such as those employing rectangles or triangles to partition images, may produce more optimal results as neither of these shapes have the geometric constraints that are imposed by the use of a square.

6.2.2.2 Classification of Domains

Since the domain / range comparison step of the image encoding process is so computationally expensive (see 6.1), one method to reduce the encoding time is to classify domains within the initial domain pool to be used for the search. Just as with partitioning methods, there are numerous schemes for classifying domains; for example it is possible to sort the domains according to whether they contain edge detail, textured details or are flat, solid regions. Once

sorted, when the search algorithm is trying to find a match for a particular range it is able to classify the range and then select the similarly classified domain set and thus avoiding repeatedly searching those domains that have been pre-classified as structurally dissimilar to the range itself.

Although it is likely to be reasonably computationally expensive in itself, domain classification only need to happen once for each level of the quadtree or for each domain pool, rather than for each range cover search, and is therefore a fixed compute cost that can be saved by reducing the size of the domain pool in an intelligent fashion when the algorithm attempts to cover a range.

Adoption of classified domains allows for more intelligent search schemes than pure brute-force comparison.

6.2.2.3 Optimising the Search with Early Termination

The quadtree algorithm allows for a domain to be mapped to a range in a number of ways – the domain can be rotated through 0, 90, 180 and 270 degrees; have its contrast increased or reduced by a number of levels; have the brightness of each pixel increased or decreased; etc.

The search algorithm itself will try every permutation of the above domain to range mappings to find the optimal domain to cover a particular range. Once this comparison is complete, the algorithm will evaluate how well the domain covers the range and, depending on whether a specified fitness level has been achieved, the candidate will be recorded as a mapping to cover the range or the range will be split into four smaller ranges and added to the quadtree range queue. In this guise, the algorithm doesn't attempt to terminate the computational expensive search function early if a suitably accurate domain to range cover has been found. The fitness evaluation is already carried out and therefore the computation needed to determine this has already taken place. The only loss in the case of early search termination would be that it's

possible a more optimal domain to range cover would not be encountered and subsequently recorded. However, terminating early could save significant computation during this step.

Additionally, initial results from the implementation of this particular quadtree algorithm suggest that some of these permutations tend to deliver better domain to range mappings than others – that is, for example, the computational effort behind adjusting domain brightness during the mapping search may statistically deliver more favourable results than the similar effort required to rotate the domain in order to match the domain. If this example was borne out in tests on a variety of images, it becomes possible to profile the performance of each domain transformation type involved in the search and to try statistically higher performing transformation types earlier in the search function.

If a fitness-based early search termination method was combined with an optimisation of the order in which the various transforms are applied to a domain, then it is likely that, with a suitable realistic fitness factor, significant savings could be made in encode time for the domain to range search.

6.2.2.4 Early Evaluation of Quadtree Fitness Success

In the previous section the concept of terminating the computationally intensive search as soon as a sufficiently good (but not necessarily optimal) domain is encountered allows for a reduction in encode time due to the early success of the search. Conversely, it could also be possible to terminate a search in a particular quadtree level early if an initial proportion of the search does not find a domain that covers a range with a fitness that represents a particular percentage of the fitness that the algorithm is attempting to achieve for a particular range at a particular level of quadtree. This would mean that hard to cover ranges, where, for example, after search 25% of the domain pool the algorithm did not find a domain that could cover the range with a fitness level of 50% of the target fitness level then it might be reasonable to assume that the other 75% of the

domain pool would be likely to fair little better and therefore may not be worth exploring due to the computational expense. In such a case, the algorithm would abandon the range it was trying to cover and would instead subdivide it for the next level of the quadtree. There are, as with early termination due to an early solution, pitfalls with not exploring the full domain pool in that it is possible that a suitable domain does exist to cover the range and therefore potential compression performance would be lost.

Just as in the previous section, early termination here would likely benefit from some prior analysis of domains and their likely success in covering a range. If domain classification were to be accurately employed, then this early termination method could be enhanced to ensure that only promising (from a classification perspective) domains were in the first 25% searched.

6.2.2.5 Improvements in Inter-Frame Encoding

In addition to improve the compression and encoding performance of the image compression engine, there are a number of improvements that could be made to the inter-frame encoding algorithm. The main area of focus for future investigations should be to move away from a square block-by-block analysis of the current frame with elements in the previous frame. Although this is clearly a very effective compression technique for removing redundancy between successive frames, it is not very intelligent nor content aware.

In this video compression implementation, the quadtree partitioning method (or image partitioning method) and the i-frame calculation method are closely related – both methods use square blocks, the i-frame process outputs a image of blocks that differ significantly to those of the preceding frame and these are then handled by the quadtree compression which partitions this image into blocks. A more modern approach to dealing simply with blocks for inter-frame redundancy removal is to identify actual elements or objects within a scene and track the mappings of these objects as the video proceeds. Thus, instead of describing a scene as regular

square blocks, it can be described as irregular, non-square objects that change positions, are transformed, can be reused, may overlap, etc. Newer video standards such as MPEG4 [MPEG] have shown that this technique can improve upon simpler block-oriented techniques. However, if the strong relationship between the partitioning method and the i-frame calculation method is to be maintained, a more flexible partitioning scheme than quadtree may need to be employed to handle the more irregular objects that could result. This adds to the benefits of more sophisticated partitioning schemes presented in 6.2.2.1.

6.3 Summary

The images presented in chapter 5 show that a working quadtree compression solution has been produced, though its current performance does not generally compete with JPEG. However, as discussed earlier in chapter 6, there are a number of improvements that can be made that would reduce the amount of computation required to compress an image whilst increasing the quality and level of compression achieved – these improvements to the base partitioning method are key to making fractal compression competitive in practical applications.

Fractal, rather than quadtree compression specifically, still offers the promise of increases in compression ratios that are an order of magnitude beyond those offered by JPEG, though the challenge of how to actually achieve these for arbitrary image sets has yet to be met. Quadtree and other partitioning schemes offer the potential for a compromise solution between the two methods and with further work and development it is possible that they will form part of the solution in compressing large digital media for storage and transmission.

7.0 References

AHO, A *et al.* ***Data Structures and Algorithms***. Addison-Wesley (1987)

BARNSLEY, M.F. ***Fractals Everywhere***, p5-105. Academic Press (1988)

BARNSLEY, M.F. ***Fractals Everywhere*** (2nd Edn.) , p210-405. Academic Press (1993)

BARNSLEY, M. ***Methods and Apparatus for Image Compression Using IFSs***. US PATENT #4941193

BARNSLEY, M. & HURD, L. P. Fractal Image Compression, p70-105. Wellesley, Mass., AK Peters (1993)

BEASLEY, D. ***Hitch-Hiker's Guide to Evolutionary Computation***, part 2. comp.ai.genetic. Joint Cardiff Computing Service (1998)

FISHER, Y. ***Fractal Image Compression***. Springer (1995)

FOGEL, L *et al.* ***Artificial Intelligence Through Simulated Evolution***. Wiley (1966)

GLEICK, J. ***CHAOS – Making a New Science***. Minerva (1988)

HELD, G. ***Data Compression: Techniques and Applications***. Wiley (1983)

HOLLAND, J.H. ***Adaptation in Natural and Artificial Systems***. University of Michigan Press (1975)

MANDELBROT, B. ***Fractals: Form, Chance and Dimension***. W.H. Freeman (1977)

MIANO, J. ***Compressed Image File Formats***. Addison Wesley (1999)

MONRO, D. & DUDBRIDGE, F. ***Rendering Algorithms for Deterministic Fractals***, p30-43. IEEE Computer Graphics and Applications (Jan 1995)

NETTLETON, D.J. ***Iterated Function Systems and Shape Representation***. PhD Thesis, p63-111, Chapter 5. University of Durham (1994)

NETTLETON, D. J. ***Representation and Generation of Graphs using Iterated Function Systems.*** Pergamon Applied Mathematics Letters (1996)

TANENBAUM, A.S. ***Computer Networks*** (3rd Edn.), p723-756. Prentice Hall (1996)

WATKINSON, J. ***The MPEG Handbook.*** Focal Press (2001)