University of Southern Queensland

Faculty of Engineering and Surveying

Multicore Algorithms for Image Alignment

A dissertation submitted by

Tristan James Ward

in fulfilment of the requirements of

Courses ENG4111 and ENG4112 Research Project

towards the degree of

Bachelor of Engineering (Software)

Submitted: October, 2011

ABSTRACT

Parallel processing is an emerging trend in modern computing. Traditional software development paradigms often forsake parallelism in their approach to produce algorithms. Applications developed then effectively relinquish any potential performance benefits gained by using multi–core processing hardware that is presently available. The fundamental idea of using parallel processing is applied to medical research and the results are reported in this dissertation. Advancements in technology within this field have the potential to greatly streamline processing, thereby directing scientific attention back to research.

Advances in medical microscopy are presently being hindered by the substantial time involved with the construction of panoramic imagery. The predominate purpose and focus of the project is to investigate and develop the automation of image alignment and noise reduction to a series of microscopy photographs, using the performance advantages of multiple processor cores. The output of the algorithms is the formation of a single microscopic panoramic image.

Pursuing the intention of parallelism, the implementation involves adaptation of certain recognized algorithms. Alignment of the images is achieved by correlation, a typical form of digital signal processing technique to measure the similarity between images. Reducing the noise in the photographs is accomplished by a computationally efficient median filter. The algorithms which were evaluated provide a means of automated batch image construction without the need for user intervention. This has the potential to save time by multi–threading on as many processor cores as possible for the system that it executes on.

University of Southern Queensland

Faculty of Engineering and Surveying

ENG4111 and ENG4112 Research Project

Limitations of Use

CERTIFICATION

I certify that the ideas, designs and experimental work, results, analyses and conclusions set out in this dissertation are entirely my own effort, except where otherwise indicated and acknowledged.

I further certify that the work is original and has not been previously submitted for assessment in any other course or institution, except where specifically stated.

Tristan James Ward

Student Number 0050086907

____

Signature

____

Date

ACKNOWLEDGEMENTS

I would like to thank my supervisor Dr. John Leis for his ongoing support, expertise, guidance and patience throughout the duration of this project. Without his assistance and direction, this project would not have come to fruition. I would also like to acknowledge the unconditional love and support of my family over this time. It was much appreciated.

T. WARD

University of Southern Queensland

October 2011

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Shortly after the turn of the 21st century, the frequency of single core processors had almost reached the maximum limit (Ramanathan 2006). Instead of increasing performance of a processor by raising the frequency, processor manufacturer giants Intel and AMD sought alternative measures to fulfil the escalating demands. The solution to improve performance was to produce a multi–core processor, to cater for the needs of running multiple process applications seemingly concurrently (Ramanathan 2006). In the last couple of years, multi–core processing hardware has become mainstream and drastically more affordable. With the advent of this hardware advancement, which is evidently here for the indefinite future (Ramanathan 2006), a shift in conventional programming paradigms is required to accommodate the full potential of the hardware.

Medical and forensic sciences are fields which benefit considerably with advancements in technology (Cooper, Huang & Ujaldon 2011). A specialisation of this sector is the technical field of microscopy; a research area which encompasses the use of microscopes to view small samples that would not otherwise be visible to human eye (Cooper, Huang & Ujaldon 2011, Rankov et al. 2005). Without computerised aid, microscopes offer a moderate insight into the sample. Physical characteristics of microscopic lenses, larger sample sizes and the restricted extent of human vision prevent the acquisition of even finer details. Digital photography is instrumental in inspecting smaller particulars and as a means of providing records. In the case of microscopy, digitised images are obtained from a camera mounted in the optical path of the microscope.

1.1 CONTEXT

Traditionally, photographic images were produced as hardcopy items, making panorama development difficult and restrictive. Since those times, digital cameras have almost entirely replaced the mature analog film counterparts for the majority of civil uses. Presently digital cameras are relatively inexpensive for resolutions below approximately 20 megapixels. They moreover represent convenience with the ease of transfer of imagery to other digital devices. However whilst still performing the same role, extremely high resolution digital cameras remain excessively expensive and for specific applications, may not exist. This facet represents an impediment for science in general, which often necessitates larger resolutions with specialised equipment to be effective.

Overcoming the exorbitant financial outlay and technical issues are managed equivalently to the domestic solution; by using smaller resolution capture devices and employing software to compensate (Rankov et al. 2005). The process of creating a finely detailed photograph is accomplished with a series of logical steps. Initially a sequence of images is taken at higher magnification using a lower resolution camera. Well constructed imagery will contain a slight overlap on the previous frame. These images are subsequently fed into image creation software, which seeks the overlaps to generate the desired product (Rankov et al. 2005). The result is known as a panorama and should closely resemble the output of high resolution capturing devices.

Microscopy notably utilises the concept of panoramas (Cooper, Huang & Ujaldon 2011). Microscopy is becoming increasingly important in the society of today and frequently plays an integral part in research projects of varying natures. The rapid incline of cancer and other diseases present in the populace has seen the need for detection of diseases and the formulation of cures escalate. A further instance where the application of microscopy is practical is for examining plant and other biological materials (Cooper, Huang & Ujaldon 2011). One such agency using microscopy is the Australian governmental research group, the Commonwealth Scientific and Industrial Research Organisation (CSIRO).

1.2 CURRENT SYSTEM

Distinctly different hardware and software setups exist between corporations and perhaps between interrelated departments of the same company. The current software that the CSIRO employs is the public domain system ImageJ. ImageJ was developed by Wayne Rasband while employed by the United States National Institutes of Health (NIH) (Collins 2007). Built on the Java runtime environment, the software is platform independent making it choice for many researchers. Since ImageJ furthermore provides mechanisms for expandability, the basic program can be improved by adding specifically designed plug–ins (Collins 2007).

The ImageJ application is not without restrictions however. At the core there are two fundamental limitations of foremost importance affecting the base program. One is that without support, the product does not feature automated generation of panoramas. Consequently manual involvement is required, directing time away from critical research. The second issue faced is that although ImageJ is built on Java for inter–platform compatibility, processing time cannot be accurately estimated due to the approach Java utilises for execution. Limited thread control management and the abstraction of the underlying Java binary code guarantees that the program forfeits capturing the full potential of the hardware and operating system (OS) (Moreira, Midkiff & Gupta 1998).

Although being reasonably precise, manual construction of panoramic imagery takes a substantial amount of time to finish. Opening photographs and moving them into the appropriate position in the final composition takes concentration in addition to time. Some of the time however is not purely spent on photographic arrangement. Delays due to the hardware and software combination consume periods of time, even with the fastest hardware available (Cooper, Huang & Ujaldon

2011). Image transformation programs by their nature exhibit memory dominating properties, notably having sizeable memory footprints when buffering high resolution imagery (Xiong & Pulli 2010).

When delays occur in graphical user interfaces because of memory or processor overloads, the system appears to the user as unresponsive. This unresponsiveness has been experienced by the microscopy scientists working for the CSIRO. Attempting simple operations such as image alignment becomes a challenging ambition, particularly when it is amalgamated with the waiting of extensive periods for computational intensive activities to be completed.

The choice of the Java language for programming moreover presents a unique problem. Developing in the Java environment has the advantage of portability among different operating systems and hardware configurations (Savitch 2010). The software distributed is compiled into Java binary format; a format necessary to function on the Java runtime environment. In this format, each instruction to be performed is interpreted in real time into native machine binary commands for the system that it is executing on (Savitch 2010). Evidently the real time translation procedure costs valuable processor time. Conversely programming in lower level languages facilitates the maximum processing speed to be achieved, at the cost of requiring recompilation for each system that the application will be used on (Savitch 2010).

1.3 DESIGN AIMS

Essentially the design of the project intends to eliminate the predominate issues associated with the existing system, namely task automation and the proper utilisation of a multi–core system. Ultimately the project seeks to:

I investigate practical parallelisation methods of previous image alignment and noise removal algorithms.

II implement the most efficient and proficient alignment and noise reduction approaches, with the intent to decrease processing time.

III output the manufactured panoramic image to a suitable and compatible format for the field of microscopy with minimal information loss.

IV evaluate the performance gains of successive cores on several differing platforms.

1.4 PROJECT OBJECTIVES

Accomplishing the design aims of the project will be realised through the subsequent objectives. The tasks the project will require include:

I research into existing image alignment techniques and how these can be achieved through parallelisation.

II research and critical analysis of current noise removal algorithms and how they can be implemented through parallelisation.

III investigate or otherwise evaluate the expected performance of the different approaches to ascertain the most efficient technique or techniques.

IV design and implementation of a working prototype based on the best processing scheme.

V review of the application performance on several differing types of machines and observe sections for improvement and optimisation.

CHAPTER 2

LITERATURE REVIEW

Algorithms for image alignment or noise reduction are not new innovations. Xia & Zhang (2010) acknowledge that composing a panorama from an image set has been explored before, with varying results. Each panoramic algorithm has precise design parameters that are to be considered when it is conceived, whether it be portability, execution speed or accuracy for a specific type of image (Rankov 2005, Szeliski 2006). The concept of multi–core based software development equally is not original. Whilst this concept matured, two differing approaches were proposed to take advantage of the hardware (Hughes & Hughes 2008). It is important to establish the gaps with these two prior developments, to comprehend how the multicore image alignment algorithm can be better applied microscopy.

2.1 EXISTING TECHNOLOGIES

There are numerous existing technologies boasting the faculty of automated panorama construction. Some of these panorama tools are integrated into the image editing program, whilst others are plug in packages to expand upon a base application. One renowned image editor in the industry is the Adobe Photoshop program. It features an automated panorama builder entitled Photomerge, which quickly assembles the panorama and displays the result on screen (O'Donohue et al. 2008). Unfortunately like most commercial products, the algorithms are patented and consequently the source code is not available to view (Adobe Systems Incorporated 2011). Entirely contrary is the Adobe Photoshop competitor GIMP, since in GIMP automated image stitching is processed only through dedicated plugins and the source code is available for all GIMP modules (Koponen 2006).

TomoJ is an ImageJ software plugin that allows semi–automated or manual panorama construction, working specifically with photographic imagery from transmission electron tomography (TET). Messaoudii et al. (2007) described TET as:

> *"... an increasingly common three–dimensional electron microscopy approach that can provide new insights into the structure of subcellular components. [TET] fills the gap between high resolution structural methods (X–ray diffraction or nuclear magnetic resonance) and optical microscopy."*

The statement by Messaoudii et al. gives an insight into the level of work conducted by microscopy researchers at the CSIRO and worldwide. However in documenting the TomoJ plugin, no mention is given of multicore algorithmic design. Without reference to multicore development and with the limitations of ImageJ as presented in the current system section (refer to 1.2), it can be assumed that the software is based on a single core process design. Often software with this capability will widely advertise this feature.

Searching online will result in the discovery of many panorama software applications, not just for computers in general but also the Apple iPhone and Android mobile phone markets. Of these products, two from large corporations are striking, which are the Autodesk Stitcher Unlimited and ArcSoft Panorama Maker Pro products. The interesting detail regarding these products is that they both claim on the boxed feature list to make panoramas with little effort from the user and have hardware optimisation algorithms (*Autodesk Stitcher Unlimited* 2011, *Panorama Maker 5 Pro* 2011). These algorithms apparently take advantage of the central processor unit (CPU) or graphics processor unit (GPU). With the CPU processor, some form of multi–threading is expected to use multiple processor cores simultaneously to decrease processing time. According to Zhang, Wang & Chen (2010), the GPU processing is of interest as the GPU is greater than ten times faster than CPU processing. The GPU is the hardware responsible for rendering the graphics on the monitor. Offloading the image processing functionality to the GPU logically will decrease the processing time for dedicated hardware (Zhang, Wang & Chen 2010).

Several problems exist with utilising the GPU approach to panorama construction. The first drawback is that the two leading GPU manufacturers, ATI and nVidia, have different programming interfaces to develop with. As cited by Wang et al. (2009), although nVidia has dubbed their technology Compute Unified Device Architecture (CUDA) and ATI has named the hardware ATI Stream, the technologies are similar. Development of GPU functions will be more challenging if two sets of interfaces have to be maintained. Communication of instructions to and from the processor to the GPU is already a complex task (Zhang & Wang & Chen 2010).

A second problem is the assumption that the computer systems utilised for microscopy at the CSRIO and elsewhere have these hardware advancements. Computers can and have been built without a dedicated graphics card for many years (Blythe 2008). There is furthermore no guarantee that a researcher that has a dedicated graphics card will support the CUDA or ATI Stream instruction sets. Yuffe et al. (2011) reveals that Intel has recently released the CPU with an integrated GPU on a single die, codenamed Sandy Bridge. This release introduces other issues, such as developers needing to program for Intel GPU processing in addition to the aforementioned technologies.

As cited by Blythe (2008), another issue with GPU rendering is the data transfer cost. Transferring data between the processor and GPU is an expensive operation and one which increases the overall processing time. Having the GPU close to the CPU on the Intel solution reduces the latency when copying or sharing data. The Intel product makes GPU processing more attractive and should be more feasible in the future, however at the present time it is a new hardware device that needs mainstream adoption.

The monetary cost of freeware applications is of significant consideration, as finances can be employed for research instead of outlay on software tools. A problem exists where freeware products oriented towards microscopy research or fields of similar nature are not designed to take advantage of multi–core hardware. Eytani & Ur (2004) implies that it is less difficult to implement and maintain singular threaded applications and that developers occasionally use this excuse to avoid spending time on producing multi–core algorithms. It is not uncommon for freeware projects to rely on the support of volunteer developers or donations to continue the expansion and improvement of particular programs (Cubranic & Booth 1999). This is one reason why features that are deemed unessential such as multi–core algorithms are overlooked in freeware software.

Fogel (2006) implies that commercial applications do not have this limitation to the same extent. Businesses have funding which they can spend on paying developer salaries and on pioneering algorithms. The objective of corporations consuming finances is that the expenses are expected to be redeemed in the profits from the sale of the software. This financial backing has a benefit in that the developers are generating income, so the design and implementation of innovative approaches becomes a higher priority than it would otherwise.

The component that is absent between freeware and commercial applications is the disclosure and distribution of designs for the purpose of education. Freeware applications have the benefits of no monetary outlay for the users and can be readily expended due to the availability of source code (Cubranic & Booth 1999). Commercial products have the advantage of innovative and computationally efficient designs. Presently researchers appear to favour freeware products, since they utilise the freeware product ImageJ and its extension, TomoJ. It would appear that not only the cost that is considered, but furthermore the specific functional layout directed for scientific use. The intention of this project is to produce an automated image stitching program that satisfies the combination of the former advantages. This would benefit researchers in the technical field of microscopy.

2.2 MULTI–CORE DESIGN

Multi–core algorithm design is an integral concept of the project. The anticipated efficiency of present multi–core algorithms undoubtedly is imperative in the potential outcome of the project. The research of Liu et al. (2010) into the performance of multicore hardware systems establishes valuable conclusions. Liu et al. (2010) tested the decrease in processing time relative to the number of hardware processors utilised. Throughout the trials, the algorithm used was the Adaptive Differential Pulse Code Modulation (ADPCM). Yatsuzuka et al. (1998) outlines that ADPCM has widespread usage in public telephone networks for reducing the bandwidth required for both telephone conversations and internet traffic. The results of Liu et al. determined that for large values of data, the performance increase approached the number of cores. This conclusion is understandable in that whilst it is acknowledged that there are processing overheads in the creation of threads and assigning tasks (Silberschatz, Galvin & Gagne 2009), these actions can be diminished when compared to a large overall processing time. When the data is small, the algorithm is not as efficient.

Unlike the approximate double times increase observed with two threads and a large data set, Liu et al. only obtained an almost triple improvement in processing times with four threads. Data access contention is one explanation as to why processing times do not reach theoretical values. Sun, Byna & Holmgren (2009) describe memory access contention as a major performance bottleneck in computing with multiple processors. Data access contentions occur when multiple processors request the same resource, such as memory bandwidth or cache memory. Only one competing processor can control a given resource at a time, causing delays for the other processors requiring the resource. Another of the tests conducted by Liu et al. on multicore algorithms was the impact of increased bus width. Using the Global Standard for Mobile Communications (GSM) encoding algorithm, the results concluded that higher bandwidth does increase the system efficiency to a point. Since the project does not have control over the hardware design and more specifically the bus width and memory layout, only the impacts of the hardware on the execution performance are examined.

2.3 IMAGE STITCHING ALGORITHMS

Image stitching is the process of creating a panorama from a set of related images, each with a slight overlap on the next. Xing & Miao (2007) defines image stitching to produce a panorama as:

> *"... a technique to merge a sequence of images with limited overlapping area into one blended picture."*

To accomplish this task, Hsieh (2003) describes the generic process of image stitching as:

> *"... recovering the existing camera motion parameters between [the various] images and then compositing them together."*

Hsieh (2003) essentially depicts the image stitching process as encompassing two major steps. The first is image registration, which involves determining a point in which to join the photographs either from the features in the images or from the image similarities. Once this coordinate is known, the two photographs can be merged into a single image. The process of finding the join point and merging is then repeated for the number of images to be processed. In advanced algorithms, any distortions, rotations or mild scaling errors are corrected before merging (Szeliski 2006).

There are numerous image alignment schemes that are available. Xing & Miao (2007) categorise image registration techniques under two broad types: direct methods; and feature based methods. Direct methods often are the simplest to develop, comparing the images pixel to pixel (Szeliski 2006). Exhaustively trying all combinations of alignment locations is known as a full search. A full search will be the most accurate of searches however it will incur a performance penalty for the significant number of computations required (Chen 1998). The easiest approach for utilising a direct method to image alignment is to shift one image relative to a template image. At intervals the two images are evaluated to calculate the sum of squared differences (SSD) (Szeliski 2006). Over all the movements of the shiftable image, the sought after point is where the SSD function is at a minimum. The median of absolute differences (MAD) is one direct based approach that follows this methodology (Szeliski 2006).

Rankov et al. (2005) disclose Correlation as an example of a direct method that is often used for image alignment and that differs in the approach taken. Although correlation still iterates over all the shiftable locations, it relies on the discovery of the cross product maximum of the two images. The Fourier transform based alignment is another direct method. The Fourier technique operates on the detail that the signal of the shiftable image has the same magnitude as the template image, but with a linearly adjusting phase. This phase can detect the appropriate join coordinate. Szeliski (2006) suggests that the Fourier Transform calculation can additionally be utilised to estimate rotations and scaling differences in the images. However since the Fourier Transform involves the calculation of the correlation algorithm, the Fourier Transform approach is overlooked in this project due to performance concerns.

The second category of image alignment algorithms is the feature detectors. Jia & Tang (2008) list several common variants including: scale invariant feature transform (SIFT); Harris corner detector; and random sample consensus (RANSAC). The list presented by Jia & Tang is confirmed by Hua, Li & Li (2010), who outline the same set of algorithms whilst exploring alternative means of image alignment. Feature detection algorithms differ in the means of identifying what pixels correspond to a feature. Ryu, Lee & Park (2011) mention some algorithms such as the Harris corner detector which focus on the corners present in an image. Other algorithms may attempt to identify edges or blobs within images. The SIFT approach regularly is applied in systems due to its generic feature detection abilities and library referencing (Hsieh 2003). After the features have been identified in all the images, matching of these features must be performed.

Feature detection algorithms are advantageous over traditional direct methods when there are image acquisition problems. Chen (1998) addresses some of the typical image acquisition related issues including: variations in the light illumination; contrast dissimilarities caused by reflections; movements in the scene between shots; and general lens distortions. Rankov et al. (2005) expressed that image capture issues aside, cross–correlation was the second fastest method they had tested, after the principle axis method which was considerably less accurate. Rankov et al. (2005) subsequently consider correlation as the preferred method. It was discovered that the calculation time of correlation could be reduced by directing the search points in the photographs to anticipated overlapping regions. However this required use of an automatic stage for capture to decrease acquisition differences (Rankov et al. 2005). Since a motorised, automatic stage could not be assumed in practical use with this project, the proposal is not of benefit.

2.4 NOISE REDUCTION ALGORITHMS

The term noise refers to imperfections in the original signal, such that certain sections of the signal no longer represent the true value. Thangavel, Manavalan & Aroquiaraj (2009) raise several distinctive types of noise found in images: Gaussian noise; Speckle noise; Rician noise; and Poisson noise. Gaussian noise is a random additive found in natural images, while Rician noise is image noise that affects Magnetic Resonance Image (MRI) photographs. Speckle noise is otherwise known as 'Salt 'n' Pepper' noise (Leis 2011) and is often present in ultrasound images (Thangavel, Manavalan & Aroquiaraj 2009). Poisson noise is the noise introduced by the camera or capture equipment. Research by Srivastava (2010) returned the same noise types present in microscopic imagery as Thangavel, Manavalan & Aroquiaraj (2009), thereby confirming the various sorts. Furthermore Srivastava (2010) confirms some of the factors producing noise, as briefly inspected for image alignment. Srivastava (2010) outlines the factors that induce noise in fluorescence microscopy photographs which include, but are not limited to:

I lens miss–focus.
II environmental factors.
III instrumental error.
IV dark current.
V electronic noise.
VI photon limited scientific charge–coupled device (CCD) cameras.

Since image capture recommendations are out of the scope of this project, noise reduction techniques will have to be designed to remove as much noise from the photographic files without reducing clarity. Thangavel, Manavalan & Aroquiaraj (2009) describes numerous approaches to remove noise from images. In most instances, the pixel and its neighbours are assessed to receive the noise reduction result. This set of pixels is known as a window, with the centremost pixel being the one to be replaced (Leis 2011). Each pixel of the image is evaluated, with the window shifting relative to the pixel being considered. Some of the approaches listed by Thangavel, Manavalan & Aroquiaraj (2009) and Leis (2011) include:

I the minimum filter. The lowest value in the window is taken as the selected value for replacement. This darkens the overall image.
II the maximum filter. The highest value in the window is taken as the selected value for replacement. This lightens the overall image.
III the moving average filter. The pixel values in the window are summed and divided by the number of pixels in the window. It is simple to implement, but the output image will be marginally blurry.
IV the median filter. The pixel values in the window are sorted and the centre value selected.
V the midpoint filter. The midpoint between the highest and lowest values is computed and selected for replacement. It is known that this approach has

shortcomings in that it: slightly blurs the image; is not robust against impulse noise; and it does not keep the image borders.

VI high boost filter. Low frequency content is removed from the image. The result is that the background detail is improved and the sharpness and brightness of the image is enhanced.

VII trace means filter. The values on the diagonal of the window are summed and divided by the number of pixels on the diagonal. It is not as computationally expensive as the moving average filter.

VIII trace median filter. The values along the diagonal of the window are sorted and the centre value selected. It is not as accurate or computationally expensive as the median filter.

IX the correlation filter. The autocorrelation of an image is computed to remove intense colour variations between pixels, which may correlate to noise.

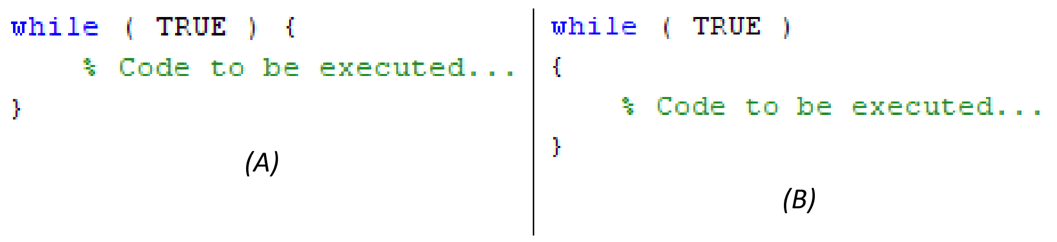
X the M3 filter. This filter is a hybrid scheme between the moving average and median filters. The maximum of both filters is selected as the value for replacement. High frequency components of the image are preserved, making it suitable for ultrasound imagery.

From the selection of algorithms possible, Thangavel, Manavalan & Aroquiaraj (2009) concluded that the M3 filter was the best on a performance basis. The illustration provided however visually shows the M3 filter loses an arguably significant amount of contrast and clarity. Without this filter, Leis (2011) suggests the median filter as the preferred choice, as it produces fewer artefacts than the moving average filter.

2.5 CODING STYLE

The coding style of the software developer has a sizeable impact on the effectiveness of the project (Kemerer & Paulk 2009). Boogerd & Moonen (2008) regards reliability, portability and maintainability as three desirable qualities that software should be built upon. These qualities reduce the cost of code maintenance and the number of faults associated with the system. Whilst there are various tools to enforce the use of a particular standard in widespread use, no universal coding standard exists. Research by Boogerd & Moonen (2008) reveals a reason that is cited for not using the software is that the developers are bombarded with warnings of non conformance. Kremenek et al. (2004) tested this claim and observed that every software tool produced false positives when enforcing coding conventions. The number of false positives recorded in the tests by Kremenek et al. ranged from 30 % to 100 %. With no formal standard on how to write applications, the style of the program that is composed is purely related to the opinions and craftsmanship of the author (Fang 2001).

There are numerous programming manuals that endeavour to present guidelines on common and accepted programming styles. Naming conventions, indentations and commenting depth and frequency are just some of the guidelines these manuals will attempt to have developers adhere to. Yet since these are merely guidelines and not rules, a programmer can legitimately disregard such suggestions (Wang et al. 2010). A classic example is where to place the opening { symbol in the C language. Two accepted styles exist, but whichever technique is chosen it is expected that the developer is consistent across all modules. Figure 2.1 shows the differences in style with the parenthesis symbol, both of which are syntactically valid.



```
while ( TRUE ) {
    % Code to be executed...
}
```

(A)

```
while ( TRUE )
{
    % Code to be executed...
}
```

(B)

Figure 2.1 Different styles of valid programming

(A) The opening parenthesis on the same line (B) The parenthesis on the proceeding line.

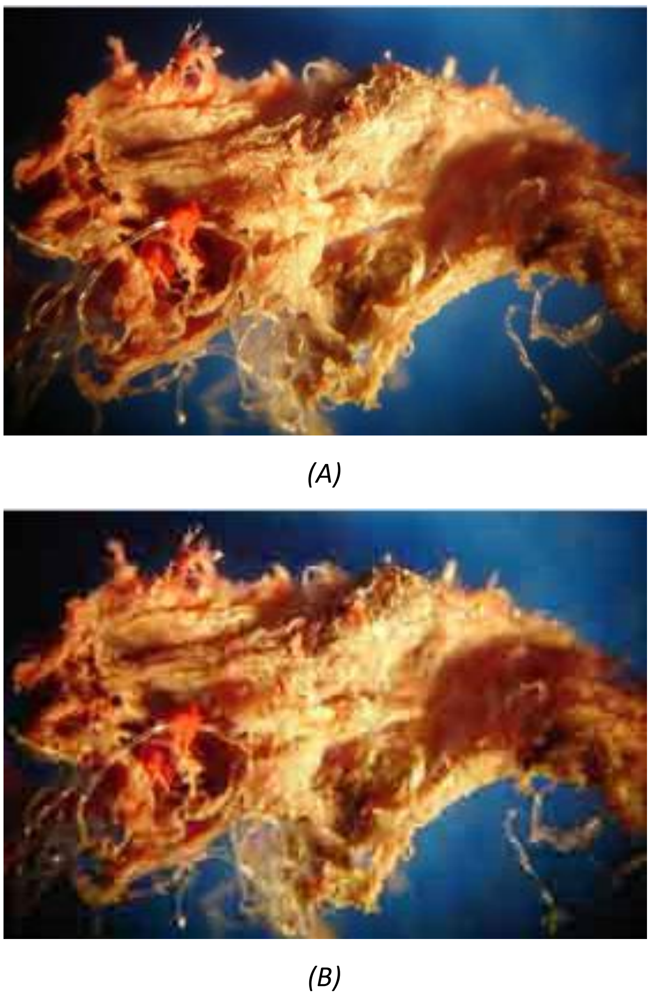
Using Figure 2.1 as an example, there are arguments for both versions. The proponents of Figure 2.1 (A) state that one less line of code is used (Mark 2009), whilst supporters of Figure 2.1 (B) claim that the code is more readable since it is not as compressed (Mark 2009). In either case, the convention chosen by the developer should be reflected throughout the differing type constructs and the work in general for consistency and professional appearance.

2.6 FILE FORMATS

There are numerous file formats presently available for the storage of image data. Work by Bell Laboratories in the late 1940's began research into compression methods, originally relating to textual communications. Salomon (2002) indicates that there is currently two generic categories for image, video or audio files. The first of these two categories is the lossless compression method, where the data is the same at the decoder as it was originally at the encoder (Salomon 2002). Between the encoder and decoder, the data may be stored in some form of compressed state to reduce the file size or is otherwise stored as raw data. According to Salomon (2002), the second category of compression is the lossy format, whereby the data is different between the encoder and decoder. In an effort to save storage space, some of the information in the original file is lost. The lossy algorithm will remove information, with consideration such that the output of the file visually or audibly appears the same as the original to the user (Xin 2009, Salomon 2002).

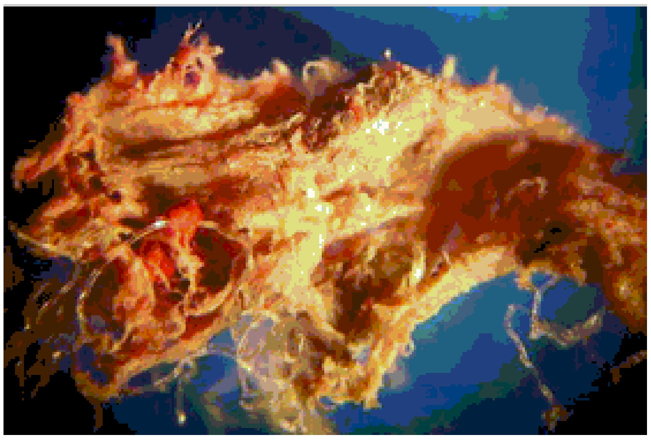
Selection of the image file format is of importance for the project. Selecting an inappropriate format to implement may result in rejection of the project as a whole. According to Xin (2009) some of the popular image file formats include: Graphics Interchange Format (GIF); Bitmap (BMP); Joint Photographic Experts Group (JPEG); and Tagged Image File Format (TIFF). GIF and BMP image files are limited in their scope for medical science as they often only have 8 bit colours, meaning that the maximum number of different colours that can be referenced is 256 (Jackson &

Hannah 1993). Although both BMP and GIF are restricted to 8 bit colour, they handle this limitation in dissimilar approaches as illustrated by Figure 2.2 (CompuServe Incorporated 1990). Another representation of the source image is with the JPEG format, as displayed in Figure 2.2 (B). Neelamani et al. (2006) indicates that JPEG is known as a lossy format and is utilised as such, even though it is acknowledged that lossless JPEG algorithms exist. In the field of microscopy, information loss in photographic files is unacceptable as researchers require as much detail as possible to properly examine samples (Rankov et al. 2005). In this context, lossy file formats are undesirable.

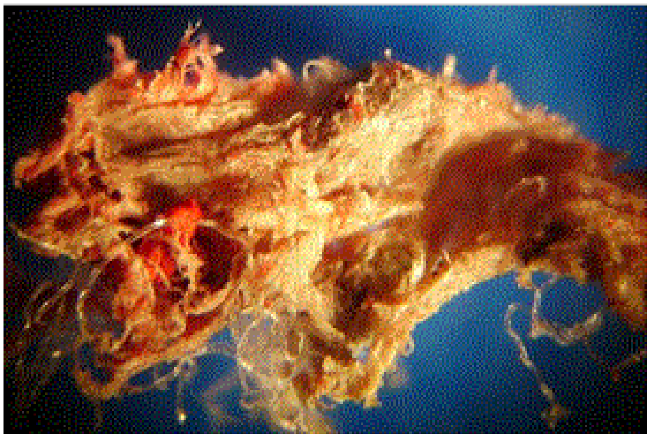


(A)

(B)



(C)



(D)

Figure 2.2 Differences in the output of image file formats (A) Original image, but also representative of lossless algorithms (B) JPEG Image (C) BMP Image (D) GIF Image.

One of the other formats in circulation is TIFF. Available in lossy and lossless, TIFF is a container for both. This makes TIFF prime for many uses, including the lossless storage of microscopy imagery. ImageJ documentation (Collins 2007) however reveals that the Digital Micrograph (DM3) format is format for microscopy research. External correspondence with the CSIRO confirms the statement that DM3 is used in microscopy. According to Jefferis (2004), unfortunately the DM3 format is a proprietary algorithm from Gatan Incorporated and it is not known if this format is universally supported by microscopy researchers. Since DM3 can be easily converted into TIFF by at least ImageJ (Collins 2007), TIFF remains the choice of file format so that the project is unanimously received.

CHAPTER 3

METHODOLOGY

At its core, this project is about the parallelisation of software tasks to entirely use current hardware. On another level, the project involves the technology used by microscopy researchers and how it can be improved as to direct the focus towards scientific endeavours. Ultimately the automation of tasks and reductions in processing time to produce panoramic imagery represent significant milestones. The first stage of design and development entails background research, followed by selection of certain design parameters.

3.1 RESEARCH AND DESIGN

Research is an imperative step in innovative design. In the context of algorithm development, any existing technologies and algorithms in circulation will provide a foundation to advance upon. Research was conducted in several areas (refer to 2.0), with much of the research into existing technologies and algorithms either being integrated into or influencing the final product. From this research, proposed designs are fabricated and a prototype application developed in the chosen language. The prototype serves two purposes: the most prominent being the capacity to test the effectiveness of the implemented design; and the second is that the project is a basis that could be improved upon if the application were to be progressed further. Modules of the final product additionally can be reviewed for use in different applications provided the programming language is known.

3.2 PROGRAMMING LANGUAGE

The selection of programming is crucial in the design of the project. Obviously use of higher level languages would decrease the development time due to their simpler syntax, allowing testing to commence more quickly. In various instances this arrangement would be portable amongst different OS environments, as some high level languages are written platform independent (Savitch 2010). A well known and used example of this is Java. The issue with high level languages such as Java is that the native compiled code is often not optimised as assembly (Moreira, Midkiff & Gupta 1998). Execution speed is however forefront to the success of the project. Similarly several high level languages including Java abstract the implementation details from the developer, restricting certain imperative functions such as the capability to fine tune multi–processing aspects.

For best executable performance, the project should be programmed in assembly language (MacKenzie 1988). Unlike high level languages, assembly is dedicated to specific hardware and is not rapidly portable. More crucially, programming in assembly involves extensive knowledge of the intended hardware design layout and substantial time to develop the appropriate program. Since this project is limited by the development duration and it is known that the hardware used may vary, assembly language is not the most suitable.

The optimal trade off between executable performance and development time is attained with use of the programming language C. It is a well recognized programming language with support in the majority of OS environments and hardware configurations. Furthermore with a cooperative compiler, generic assembly code optimisations can be performed without any intervention of the developer (Moreira, Midkiff & Gupta 1998). Extensive instruction configuration is incorporated into the design of the language. For these reasons, C is the language of choice throughout the project.

3.3 HARDWARE AND SOFTWARE PLATFORMS

Originally the project started without any constraints on the hardware or software utilised. As the design and development progressed, real restrictions on the software became apparent. The development of the prototype is restricted to the Microsoft Windows OS by a few Windows dependent Application Programming Interface (API) function calls. Without amendments to these sections of code to be more universal, the OS must be at least Microsoft Windows XP or capable of running surrogate Windows instructions. This prerequisite reduces the software requirements for testing significantly.

Hardware limitations are introduced by the obligation to run the project on Windows compatible and capable systems. For Windows XP, the minimum hardware system requirements are specified by Microsoft (2007):

"● *Pentium 233-megahertz (MHz) processor or faster (300 MHz is recommended)*
● *At least 64 megabytes (MB) of RAM (128 MB is recommended)*
● *At least 1.5 gigabytes (GB) of available space on the hard disk*
● *CD-ROM or DVD-ROM drive*
● *Keyboard and a Microsoft Mouse or some other compatible pointing device*
● *Video adapter and monitor with Super VGA (800 x 600) or higher resolution ... "*

Evidently these are the absolute minimum hardware specifications that the project will operate on. However these requirements are obsolete by the unofficial standards of present computers. To properly appreciate the performance advantages of this project, a multi–core processor system is compulsory. All computers powered by at least a dual core processor will be sufficiently adequate to run the project application. Newer machines with faster frequency processors and larger caches will clearly observe a greater benefit with condensed processing times compared to older hardware.

3.4 PERFORMANCE TESTING

Performance testing is central to the evaluation and analysis of the designed approaches for image alignment and noise reduction. Testing facilitates deductions to be formed regarding the successfulness of the project. The test results of the prototype project application must be recorded in suitable units and obtained with a degree of accuracy to be of value. It is of no benefit to have the technical representation of the results reported in the number of instructions processed, as it is meaningless for assessments in its end use. Considering these rationales, time was selected as the preferred unit for its relevance and understandable comparisons in modern society.

There are numerous methods that could be used to gauge the duration of the image alignment algorithm. Traditionally the counting of the seconds or minutes passed is one approach that could be used without much deliberation. A considerably better approximation is acquired with a stopwatch. A stopwatch could be a mechanical or electronic device. Conveniently Microsoft Windows has a reasonable clock that could be used for timing, since the computer must already be on to execute the program. All of these approaches share a common oversight, in that the tester must be concentrating on the computer until the tests have terminated. Likewise the accuracy of the timing is relative to the human response time, which accumulates an indeterminate amount twice for each test.

To counteract these shortcomings, variations to the anticipated calculation and render time could be achieved by increasing or decreasing the number of files or file dimensions. Clearly this solution is unacceptable in production, but moreover discrete issues are produced in estimating the processing time. If the images used as the test are sized too small, the outcomes will not be sufficiently invariant to deduce conclusions and the human timer might not record adequate differences. Similarly, small deviations in nominal execution such as a simple context switch mid processing would obscure the result. Nevertheless if image sizes or the number of files are set too high, the human timer would be spending large amounts of time waiting. Because of this the timer may not be as responsive to halting the stopwatch at the end of the test, again leading to the inaccuracies as described.

Rectifying the issue of accurate timing is resolved with an inbuilt counter in the application. The time is recorded from the function `clock()` as an integer representing the number of clocks of the hardware. Comparing the start and end clock values divided by the number of clocks per second gives the time in seconds. After each processing stage and at the conclusion of the program a timed value is printed to the terminal screen. This value is as accurate as practically useable.

The tests are carried out on a set of separate hardware to ensure that different combinations of processor frequencies, cache and software environments do not drastically alter the results. Initially the project is started with one thread, mimicking a single core machine. It is then gradually stepped up one thread at a time to the maximum number of threads, which is equivalent to the total quantity of hardware cores. At each thread count, the test is carried out on the same set of images, the result recorded and the test repeated for consistency.

CHAPTER 4

MULTI–CORE COMPUTING

Before the advent of the multi–core processor, single core machines dominated with persistently increasing frequencies (Ramanathan 2006). Once the multi–core processor became mainstream, there was a delay in the development of software applications to utilise the hardware entirely. Software continued to be designed on previous generation models and programming languages that only considered the now superseded hardware of the day. This led to sequential programming approaches, much of which is still in existence (Bridges et al. 2007). Designing programs for multi–core processors is not an automated, instinctive approach. Rather careful design strategies contribute to a thoroughly efficient use of the hardware (Bridges et al. 2007). Although some OS environments provide several methods to accomplish multi–tasking, only two generic methods are considered that are reasonably consistent across a diverse range of operating systems. Processes and threads are the aforementioned mechanisms.

4.1 PROCESSES

In terms of computing, a process is defined by the Oxford Dictionary as:

"A series of actions or steps taken in order to achieve a particular end."

In essence this is logically the intended outcome of a process. However this definition better represents the notion of a program. A program is a passive entity that often resides on non volatile storage as an executable file. Known to many as a computer program or application, an executable file is merely a container for a list of processor instructions (Silberschatz, Galvin & Gagne 2009). Since a program is not allocated any hardware resources, the instructions form a series of steps that provide a means to solve a problem if followed or run. The transition from the definition of a program to a process follows after the program is loaded in memory, ready for execution.

Considering the implementation in software, an amended classification to the Oxford definition of a process can be altered to accommodate the impact on memory and processors. A software process contains all resources required for operation with an operating system. Examples of the types of resources a process possesses and has control over are: the program counter; hardware processor registers; a stack for temporary data; and a section for dynamic memory provision known as the heap. All of these resources consume system memory and processor time to perform the instructions imbedded in the process. Subsequently, Silberschatz, Galvin and Gagne (2009) informally define the computing process as:

> *"... a program in execution ... [and which] is an active entity, with a program counter specifying the next instruction to execute and a set of associated resources."*

In practice, at least one process is essential for the program instructions to be performed. Hughes and Hughes (2008) outline some of the primary reasons that multiple processes are used in multi–core systems. The first beneficial rationale is that each process comprises of a separate address space. Isolating a list of the location of variables stored in memory is valuable as it provides a barrier between rouge and badly programmed applications from modifying data that they should not access (Hughes & Hughes 2008). Moreover, an impression of redundancy is created in the circumstance of an errant process. In this instance, the other processes might still be able to fulfil the functions successfully without the problematic process crashing the entire program. Another motivation for developing multi–process programs is for the expansion of memory allocations (Hughes & Hughes 2008). Each process is assigned a limited quota of resources by the OS. The amount of resources and files for prospective utilisation noticeably increases relative to the number of processes active. Without otherwise resorting to shared memory or other means to expand the maximum quantity of resources, multi–process applications are an alternative.

The benefits of processes do not come without tradeoffs. The OS accordingly uses the model of processes to manage hardware resource usage appropriately. Without any form of context or relationship between several executing processes, typically the OS will manage processor time by some form of pre–emptive time slicing and will govern system memory by paging infrequently used memory to disk (Silberschatz, Galvin & Gagne 2009). Whilst these forms of hardware management can be effective in many circumstances, they are often inefficient in the context of multi–core programming.

Dependant on the application, a developer is permitted to initiate multiple processes to utilise the various hardware cores (Bridges et al. 2007). A direct disadvantage of using multiple processes to utilise the available hardware is that multiple processes take more computational power. Processes host referencing data that is central for correct OS operation. When the OS decides that the process has had enough time on the hardware, a context switch between multiple processes occurs. When the OS reinstates hardware privileges, data from the presently executing process such as register states have to be copied to storage so that the process can continue execution (Bovet & Cesati 2006). Once complete, the opposite is applied for the process about to begin operation. Data is transferred from storage to the appropriate locations and the process continues from where it was interrupted. Evidently context switching between processes is a timely feat.

Multiple processes unnecessarily duplicate information in memory. Using processes in a multi–core system requires one process for every hardware core that is to be utilised. In Microsoft Windows OS, each process can be forced on a specific hardware processor by the `SetProcessAffinityMask()` API function (MSDN 2011). Similar API calls are available in most OS's. However since each process contains a unique set of resources, at least the instructional code of the process is duplicated. Overlooking the situations where the data duplication of multi–process could be tolerated, this type of design is inefficient, producing both slight performance penalties and resource overheads. Figure 4.1 graphically illustrates the issue of memory duplication on a dual core machine which requires two processes.

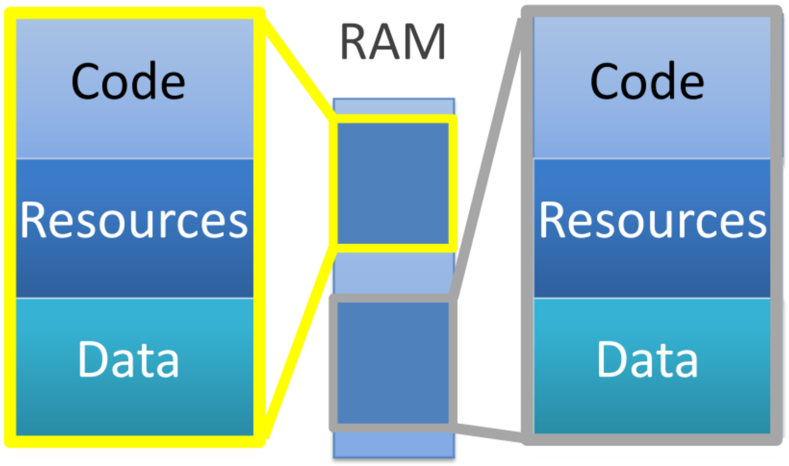


Figure 4.1 Representation of memory usage of two processes

This shortcoming is additionally compounded once consideration is given to the image data stored for the specific application of this project. The simplest possibility is that each process encompasses its own copy of the image data. Performance would be degraded from the need to read the same file multiple times, according to the number of processes. Similarly, the size of the image files and the amount of data duplicated may exceed usable thresholds, leading to undesirable events such as disk thrashing. An alternative possibility is to create a region of shared memory in which to store the common image data. Shared memory would reduce duplication and alleviate potential issues associated with the transfer and composition of the panorama data between the various processes.

4.2 THREADING

One of the predominate resources a process includes is that of at least one thread. A thread is defined by Akhter and Roberts (2006) as:

> *"... a discrete sequence of related instructions that is executed independently of other instruction sequences."*

Threads are consequently a series of instructions, known as a function, which executes on a hardware unit. The disparity between multiple threads and multiple processes is that threads of the same process inherently share most of the common process–wide resources. Threads are therefore a lighter–weight approach to processes in that threads are less demanding on memory by eliminating the majority of the code and resource duplication observed with processes. The minimum resource requirement for threading includes data items such as the stack (Hughes & Hughes 2008), where local variables and function pointers are situated. Figure 4.2 exhibits the same dual core machine as Figure 4.1, although with multiple threads.

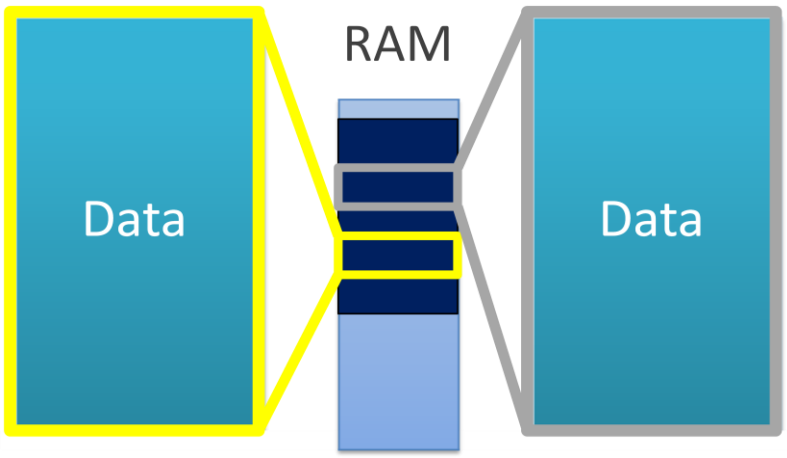


Figure 4.2 Representation of memory usage of two threads in a single process

Employing threads in a multi–core system has several benefits. A paramount advantage is that it is more computationally efficient for a changeover between threads than a process context switch (Akhter & Roberts 2006). Threads of the same process share the same distinguishing process attributes, so there is less storage and retrieval of state information when the OS negotiates hardware scheduling. Furthermore threads avert the need for contact between sibling threads to the degree apparent with inter–process communications. Since a thread is perceived to be merely a function, no inter–thread communication is required besides the control of resources and timing. Most common resources are shared by design in threads, so exchange of elements such as global variables are unnecessary (Akhter & Roberts 2006). Savings in both execution and development time and expenses are realistic.

4.3 PROPOSED DESIGN

Drafting of the multi–core method involved several distinct design choices. The design commenced with the criteria for selecting the multi–core processing technique. The criteria for the project was based on two elements, execution speed and memory usage. The fundamental facet of the project is to have the time of image stitching and construction reduced. Clearly the performance of the chosen approach is a significant and influential factor. Likewise the memory overheads are a consideration, as the extent of the resource used in the proceeding algorithms are expected to be sizeable. The more resources used along with larger sized resources will result in memory constraints on all modules, limiting the algorithms to the minimum memory footprint feasible without disrupting routine operation. This is to ensure that adequate processing potential was provided to a diverse range of hardware. Evaluating the alternatives, threading was preferred as it was most applicable for the aforementioned criteria. This approach follows current development conventions and ideologies for multi–core systems. Subsequently, an approach to thread invocation is required.

Conventionally threads are created and destroyed as required, however this induces a performance penalty. Although this method is easier to produce (Lee et al. 2011), each time a thread is created or destroyed an allocation or reclamation of memory and system resources occurs respectively. These transactions consume valuable processor time that could be productively used for processing. Instead the project establishes the threads once at program initialisation. Threads are suspended when in idle state and resumed when there is a task to process.

Generally initialisation of the threads is performed before the threads are used. The threading function in the project prepares the number of threads according to either the number of hardware cores or the input entered as an argument to the process. If present, the input number of threads is capped at the number of hardware cores. Each of the threads is given a dedicated hardware core to operate on, which is unused by any other thread in the process. The rationale is that peak performance is obtained if threads have separate cores and are not in direct contention for equivalent processor resources. A thread will commence by making a call to the OS API to suspend itself, as there are no tasks to process. The API is a set of predefined methods that perform a tested sequence of instructions, without having to develop anything from scratch.

When the point arrives in an algorithm for it to be multi–threaded, a function named `assignThreadFunction` is called with five arguments. The purpose of this method is to instigate another function to begin operation on one of the threads. The arguments of `assignThreadFunction` comprise of:

I an integer representing a core on which the proceeding routine should execute on.

II a pointer to a function to process. The method must have a prototype of `void functionName (IMAGE_LIMITS, IMAGE_LIMITS, char*)`

III two `IMAGE_LIMITS` structures containing:

A an integer indicative of the image to be operated on.

B a pair of structures that encompass the minimum and maximum coordinates that the processing algorithms can use for boundaries.

IV a pointer to a character array where output data will be stored, if applicable.

Verification that a thread is not formerly processing when it is called and the synchronisation of multiple threads for a particular task involves the analogous concept of a semaphore. Semaphores are an OS level construct that offers a technique of autonomous mutual exclusion (Silberschatz, Galvin & Gagne 2009). Essentially a semaphore is an ordinary variable owned by the OS, which is incremented and decremented atomically. When the value of the semaphore is zero, a process endeavouring to utilise the resource that the semaphore locks must wait until the value of the semaphore is positive. In this project, the first use of the semaphore is to be waited on when trying to task a thread as a precaution. The thread cannot process two items simultaneously; therefore the lock supplies a means of verification that only a singular function can process.

Synchronisation is the second use for the semaphore. Various algorithms require that all threads accomplish their assignment before moving onto the next phase of instructions. This can be illustrated in examples such as averaging across multiple threads. It is evident that if the control thread did not wait until all threads were adequately complete before progressing, an incorrect value for the average could be attained. Even worse, some variables might not be created leading to a segmentation fault. In the project, a dedicated semaphore exists for the number of threads. As each thread is freed of its previous responsibilities, the `waitForAllCores` function collects the semaphore of the thread. Once the `waitForAllCores` function has decremented the semaphore count to zero, all threads have completed their respective assignments and the waiting method can continue.

CHAPTER 5

IMAGE ALIGNMENT

Image alignment is the process of engineering a common coordinate system that is shared among a set of interrelated photographs (Hsieh 2003). It is a technique of calculating the similarities in two images and devising a position based system relative to both images. Presently the image alignment algorithms in use consume a tremendous amount of time to complete, as they are a computationally intensive task involving a calculation for nearly every pixel (Chen 1998). In this project, image alignment is employed not purely to formulate a coordinate structure, but to discover alignment points between all of the images. Figure 5.1 shows the point of join with two images. After the points have been resolved, the algorithm must join the images at the predetermined location. Lowering the time the image alignment algorithms take to complete the task can be achieved through multi–processing on a multi–core system.

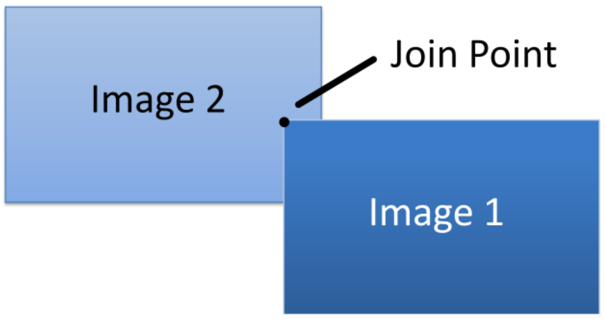


Figure 5.1 Image alignment of two images showing the join coordinate.

Comprehensive image alignment is a complex exercise and is influenced by numerous factors which are out of control of the developer. Brown (1992) suggests that the photographs could be taken at varied times and from different sensors or viewpoints. Although this is feasible, in actuality photos that are intended for producing panoramic images are often taken successively, with only a slight variation in the viewpoint. The intent is for no deficiencies to emerge between each shot, though problems arising are inevitable. Alignment of one image to the next is certainly an arduous task, since even small variations in the capture parameters of the image can misguide the alignment position.

5.1 ISSUES AND ASSUMPTIONS

Ideally the panoramic photograph would be captured on a single camera and through a wide angle lens, to negate the time and capital involved in constructing a panorama. Clearly acquiring or obtaining access to high resolution camera technology equipment is not practical in the majority of situations. The next best is to have the simultaneous acquisition of two overlapping photographs, with no differentiation in perspectives or with any form of distortion. Whilst technically this is not impossible, it is just as improbable as the former proposition. Consequently it is acknowledged that some deformation will be present in the images input into the alignment algorithm. The specific application of this project pertaining to image alignment facilitates several assumptions regarding the input files to be made. Image alignment related issues include (Szeliski 2006):

I the parallax error in each image. Derived from the Greek meaning alteration, parallax error relates to the difference between the perceivable inclinations of an object at varying viewpoints (Zeilik & Gregory 1998). These viewports are the images in the set to be joined. As a capture device sweeps around, objects nearby tend to appear moving with respect to the distant background. If the parallax error occurs on entities that are to be blended in the overlap, the join stem in the resultant panorama will be blurry. Szeliski (2006) suggested 2D optical flow motion estimation as a method to compensate for radial distortion and parallax. Radial distortion is another similar issue that can have origins with the CCD orientations in the camera.

Radial distortion is the curvature of the edges of an image so that a rectangular image warps in a circular profile (Szeliski 2006). Both of these are presumed to be negligible in the images produced by microscopy, as the device is particularly close to the object and the microscope lenses should have low parallax tolerances. No parallax compensation is accounted for in the project.

II the rotations of any image. Unless the images are taken with a tripod or similar apparatus that is absolutely level, some rotations will be introduced. In domestic photography, the rotation might not be enough to be conspicuous and would probably go unnoticed. With medical imagery however, rotations may represent a large issue with the diagnosis. Misaligned images in a panorama could perceivably be misleading to the identification and analysis of the object. Without any prior medical qualification and since image rotations are a per image attribute, applying an autocorrecting rotation to each image is out of the scope of this project.

III the perspective and distortion found in images. Objects that are skewed in the 3D plane can be repaired by affine transformation. Affine transformation is the mathematical properties that allow the vectors of the image in all dimensions to be rotated and skewed as to reproduce the non skewed version. The microscope is nominally calibrated to capture images on a horizontal surface that is parallel to the microscope camera device. Because of this, it is extremely implausible that affine transformations will occur and need to be accounted for.

5.2 SCALE–INVARIANT FEATURE TRANSFORM

Scale–invariant feature transform is a type of object recognition system with a wide range of applications. Developed by Lowe in 1999, SIFT is one of many feature detection algorithms available that can depict or outline various details within a photograph. The Oxford Dictionary defines a feature as:

> *"... a distinctive characteristic of a linguistic unit ... that serves to distinguish it from others of the same type."*

By this definition, a feature is in essence a point of interest. Whilst being direct in that a feature must be a differentiating component, it is unclear from the description of the exact specifics of what constitutes such a distinction. Due to varied applications where feature detection is utilised (Lowe 1999), many unique forms of feature detection algorithms have been developed. Some such systems include: feature description; edge detection; corner detection; and blob detection. Each algorithm plays a considerable role in the field for which it originates. The SIFT algorithm is part of the set of feature descriptors.

The SIFT algorithm begins by first extracting key points from a collection of reference images (Lowe 1999). These key vector points are stored in a library. When an image is input into the algorithm to have its features identified, the algorithm cycles each pixel generating feature map. The feature map is the vectors of interest which are compared to the library. If a matching candidate is found, the key vectors in the input image are classified and indexed accordingly. The principle benefit of this approach is that overall, detection is invariant with respect to image: scaling; orientation; position; and with minimal effect, noise and slight distortions (Lowe 1999, Hua, Li & Li 2010). Key features are based on an array of vector points and are scrutinised under these attributes. Positive matches discovered are transferred for subsequent analysis which seeks to discard outlier vector objects. The vectors remaining relate to detectable characteristic.

Xing & Miao (2007) state in research on the SIFT algorithm that it is a complex technique. Xing & Miao outline their steps to perform the SIFT calculation and merge the images of the panorama:

“I Choose an image as referenced one.

II Find the feature matched in the neighboring images.

III Calculate the homography H of the two images.

IV Apply H to warp and project the image 2 to the same coordinate system as the image 1, and then process image 2 and stitch them seamlessly.”

A substantial issue with the SIFT approach and all equivalent subsets is that the features present in the images for alignment have not always been identified. The technical field in science of microscopy researches into both existing and undiscovered substances. In the context of this project, the SIFT algorithm is not practical. Maintaining reliable library records to ensure accurate image alignment joins is not convenient, as it leads to microscopy researchers again focusing on technology rather than science. Likewise updating the catalogue of items every time a new object is found would slow progress down in this application.

5.3 CORRELATION

Correlation is a commonly used digital signal processing technique to filter noise from electrical and audio signals (Leis 2011). Noise refers to disturbances in the original signal, such that certain parts of the signal no longer represent the true value. For a number of reasons, signals often gather noise through transmission mediums. Comparing a signal buried in noise with the original will conclude with a negative result. Correlation forms an output waveform based on two inputs, which are the known original signal, and an acquired signal that contains noise (Leis 2011). The correlation algorithm then strives to repair the corrupted signal so that the best waveform that resembles the original is produced. Notably the technique of correlation can be applied to image alignment.

Using correlation for image alignment involves a marginally adapted methodology. With images, one image is arbitrarily selected as being stationary and the second image is shifted relative to the first (Rankov et al. 2005). The shiftable image is incremented from a one pixel overlap in the top left corner of the stationary image, to a one pixel overlap in the bottom right corner of the stationary image. At each increment, the similarity of the overlap of the two images is calculated. It is noteworthy that all computations are only performed on the overlap region of the two images, dubbed the viewport. After all calculations are performed, the algorithm seeks the highest peak in output waveform. The coordinate offset with this highest similarity is selected as the point to join (Rankov et al. 2005). The plot in Figure 5.2 displays the highest peak in the output, which is appropriately positioned at offset (0,0) since the diagram is the correlation of the same image.

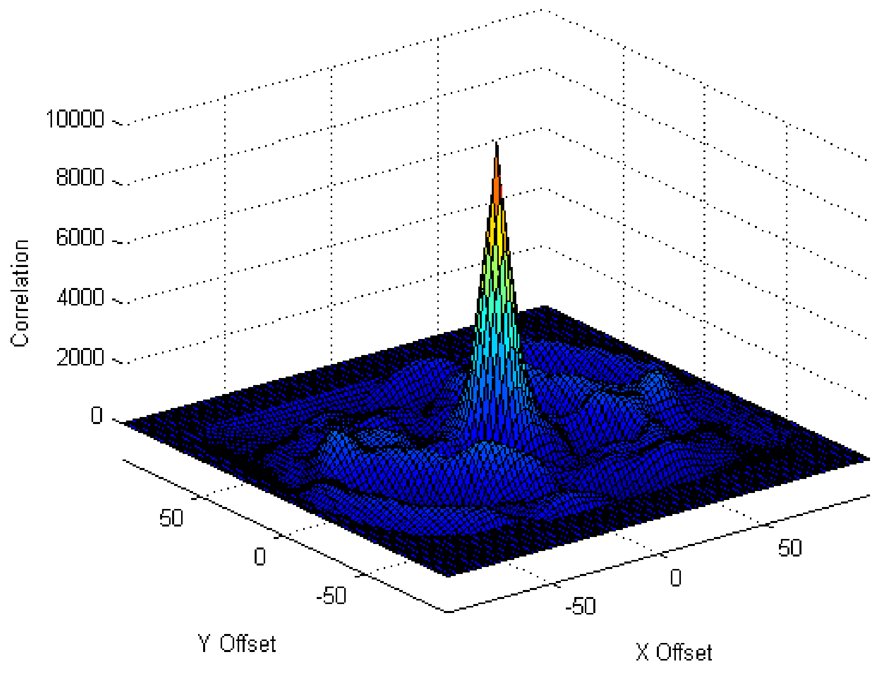


Figure 5.2 Surface plot of the correlation of the same image with dimensions 100 × 100 pixels.

```
% Compute the following Mean Absolute Deviation (MAD) twice;
% once each for the two images to compare.
ArrayMean = Mean ( ImagePixelArray )
ImagePixelArray = ImagePixelArray - ArrayMean
MAD = SquareRoot ( Mean ( ImagePixelArray * ImagePixelArray ) )

% Evalute the Correlation instruction once.
Correlation = Sum ( ( Image1PixelArray * Image2PixelArray ) /
                    ( MAD1 * MAD2 ) )
```

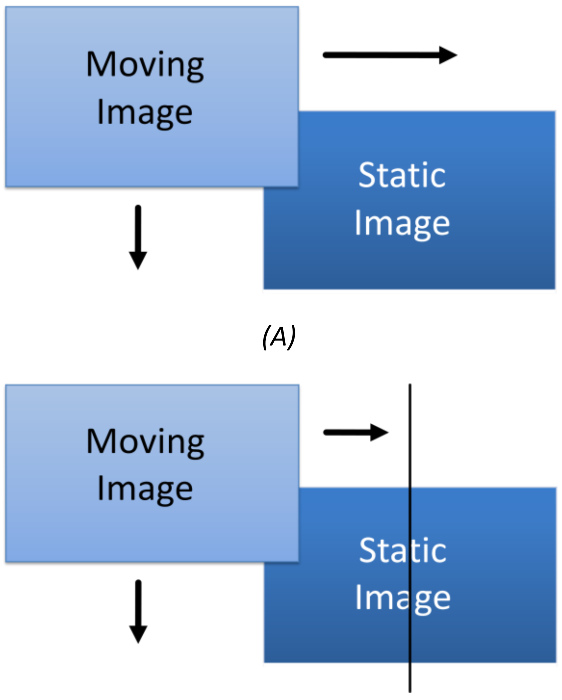
Figure 5.3 Pseudo code instructions for the calculation of correlation

The calculation of correlation is performed in a number of stages. The sequence for evaluation of the correlation is presented as a codified list of pseudo code instructions in Figure 5.3.

The correlation algorithm has several benefits over SIFT. A significant advantageous factor is that no library scheme is mandatory. The requirement to have existing items in storage for comparisons against in conjunction with maintenance time necessary for library upkeep, increases the prerequisites of the SIFT algorithm. Eliminating these founding prerequisites saves both time and capital. Another advantage of the correlation algorithm is that it is relatively easy to implement in the chosen development environment.

5.4 THE DESIGN

Composition of the image alignment algorithm involved numerous phases. The initial phase was to select the appropriate algorithm to ascertain a suitable join point. Correlation was selected for this purpose. However the correlation algorithm disclosed in Figure 5.3 is expected to be iterated repeatedly for every horizontal and vertical position that the shifting image can take. Conventionally this is performed through a nested loop, which begins with the top–leftmost location and concludes with the bottom–rightmost (Rankov et al. 2005). Figure 5.4 (A) illustrates this aspect diagrammatically. The looping construct consequently requires modification to utilise a multi–threaded programming procedure.



(B)

Figure 5.4 Image alignment approach for similarity calculation

(A) Traditional single–core type (B) Division for a dual core machine.

A solution to this problem is illustrated in Figure 5.4 (B). In the case of the dual core machine, the boundary of movement is halved. The moving image is now restricted from shifting across all horizontal and vertical combinations of the static image, to being able to move up to the vertical line. What this means for processing is that essentially the processing time theoretically is halved. The first thread executing on the first hardware core would process all the correlation calculations on the left side of the static image. Likewise the second thread operating on the second core would process the right half. In a perfect arrangement, both threads would start concurrently with no initialisation overhead and would complete simultaneously. A prime gain of applying this approach for the division of threading tasks is that it is scalable. Any quantity of threads can equally execute segments of the image alignment algorithm. This is in contrast to the many existing implementations that are hardcoded.

The multi–core alignment methodology developed falls short when the algorithm offers two offsets as the join point; one offset for each thread. This issue is overcome by the controlling thread saving all the calculated values from each thread in an array. Once all threads complete the computations, the control thread selects the highest correlation value from the array and uses the corresponding horizontal and vertical offset as the join point. At this point, the offset for this image combination is stored in a separate array. This second array which contains all of the offset values is subsequently normalised, so that one image will start with either a horizontal or vertical offset of zero. Negative offsets will cause corruption when the images are compiled into a single panorama, as the location is used directly for its position in the final image. Undoubtedly image files cannot have negative dimensions. The cycle of calculating similarities on multiple threads, determining the largest and storing the offset result, repeats until all image files are processed.

In spite of the multi–threaded algorithm for correlation evaluation, the calculations remain computationally intensive. Further reductions in the processing time are fulfilled by regulating the number of calculations that the processor performs. Instead of evaluating the similarity at every increment of the moveable image, the default is calibrated to every third. This is configurable on the command line with the switch `--align-step=value`. At every third increment, the processing time for this function is theoretically reduced by 66%. Having the step size set at every third increment nonetheless introduces its own issues. In the example presented in Figure 5.2, the offset value of (0,0) is only obtainable when the step size coincides with zero horizontal and vertical coordinates. The default maximum offset error is ±3, but this could be higher if a larger increment size is chosen.

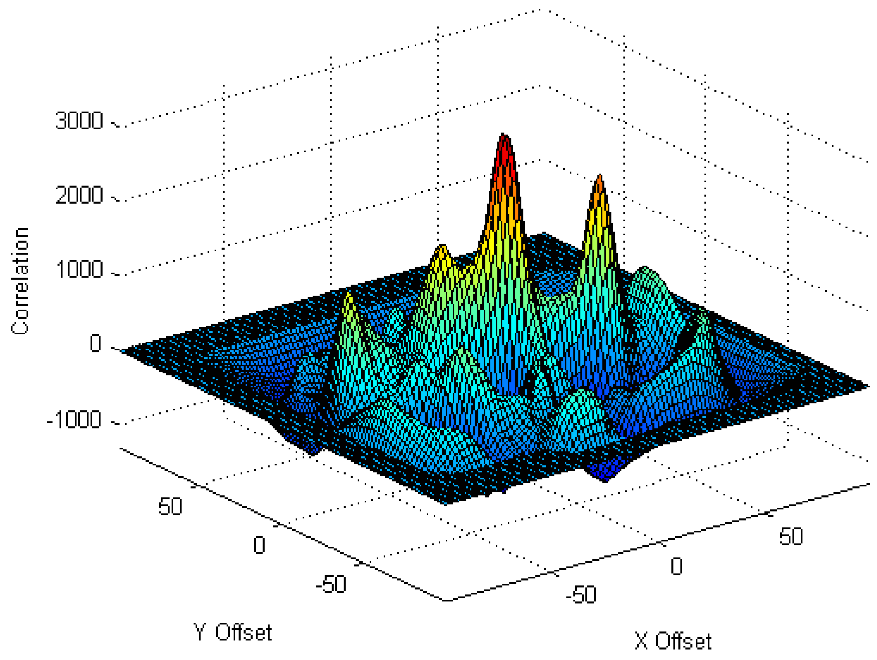


Figure 5.5 Surface plot of the correlation of different images with dimensions 100 × 100 pixels.

Figure 5.5 graphically represents the step size problem. When referring to two different images it is common for there to be several crests in the output plot. In any given image, there can only be one peak that is classified by the algorithm as being the highest. If the increment size is set too high in order to save processing time, the genuine highest peak may not be selected as the join point. To rectify this exception, the correlation function repeats itself over the range between the value it has selected as the highest and the neighbours of this selected point. Every increment between both neighbours of the designated peak is calculated with the optimism that any higher peaks could be valid in this range. In the instance of Figure 5.5, if the crest on the right side were to be determined as the highest and the step size sufficiently large, the authentic ultimate peak would be detected during the second iteration.

After obtaining the image alignment coordinates for all images the next phase of the algorithm is proceeded. Preparations begin on compiling the final panorama. Using the information in the array on the offsets of the images and the height and width of each image, the output image dimensions are ascertained. Local memory is allocated according to this size. Utilising the same approach used for division of the image alignment task, the activities of image compilation and the transfer of local data is multi–processed. The output image data is transferred to a location of choice; namely the output structure in memory that is written to disk.

A deficiency of the image compilation method is the lack of blending. Initially the designs included fading between images. Merges were calculated using relative coordinates to the second image and blending was performed in the correct direction. This functionality was removed late in the development due to the routine being incapable to perform the operation consistently and without disturbance to general image compilation.

CHAPTER 6

NOISE REDUCTION

Noise refers to the amount of errors or imperfections that are enclosed in an image compared to what is present in the original exposure. Stroebel and Zakia (1993) describe image noise as:

> *"... random variations, associated with detection and reproduction systems, that limit the sensitivity of detectors and the fidelity of reproductions ..."*

There are numerous reasons why noise exists in images. Some examples of the causes include: dust or particles developed between the camera and the object; light reflections across the lens introducing graininess; or transmission errors altering the intended values (Srivastava 2010). Indeed combinations of these issues are probable which further compounds the incapacity of a photograph to perfectly represent the subject. The challenge of noise reduction ideally is to remove all of these indicated defects and improve clarity in an image.

Various algorithms have been developed to resolve noise affected images, each with differing objectives and benefits. The type of noise, its frequency and the context of the image all contribute to the diverse assortment of algorithms developed. Defining what parts of an image constitute noise is the prevalent problem. Developing an algorithm based on a limited subset of images is likely not to be as effective or precise as diminishing the noise on a dissimilar style of image. Only some of the common approaches for noise reduction will be discussed in this section.

6.1 MOVING AVERAGE FILTER

The moving average filter is a particularly straightforward noise reduction algorithm. Every pixel of the image is iterated and the mean of the neighbouring pixels are calculated (Mather 2004). Harnessing a 3 × 3 pixel window size, a single pixel around each extremity of the centremost pixel is summated, including the value of the centremost pixel itself. The centre pixel is the pixel designated for noise reduction. It is subsequently substituted with the mean.

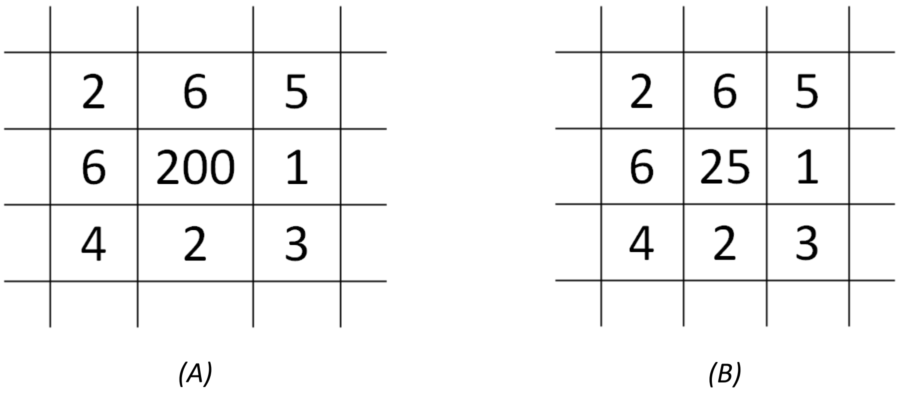


Figure 6.1 Moving average filter (A) Original window (B) Window after filtering.

Usage of the moving average filter with a 3 × 3 pixel window is illustrated with an example in Figure 6.1. It is worth noting that greyscale images are generally represented in files as numbered quantities, therefore this depiction is apt. In this instance, the square for noise reduction is listed as holding the value of 200. Amid the context of the surrounding pixels, it can be perceived that the number 200 is out of perspective. Conveniently, the majority of all noise encountered in this project follows a similar nature to this, with pixel values either being too high or too low for the region. Calculation of the mean results in a slightly more appropriate figure of approximately 25, as visible in Figure 6.1 (B).

One of the disadvantages associated with the moving average filter is the effectiveness of the noise reduction (Leis 2011). The size of the window plays a significant role in the amount of noise removed. The performance of the filter is restricted to approximately $O(n\ )$. Whilst a smaller window size will increase the throughput of the algorithm, too small a window size results in the noise not being thoroughly removed. Conversely, window dimensions that are too large yield poor comparative performance and inferior image clarity. Incidentally, testing seemingly demonstrated that the 3×3 pixel window offers the best ratio of performance to noise removal accuracy.

Window sizing is not the only limiting factor on the effectiveness of the moving average filter. The approach to filtering tends towards instability when the noise is vastly different to the anticipated value. Figure 6.1 displays such a case. It could be supposed that the expected value to be replaced in Figure 6.1 would be no higher than perhaps ten, as apparent by those adjacent to it. Any pixels that are vastly opposing to the predicated will not have noise entirely reduced and will in effect contribute to incongruous values applied in the mean of neighbouring squares. The image thus becomes visually blurry to the viewer (Leis 2011). It is acknowledged that successive revisions of the algorithm will gradually reduce noise, at the expense of the loss of image clarity and perceived blurriness. Likewise the ubiquitous issue arises as with all image filters, which lies in the definition of noise. In this circumstance, it is not clear what actual value should replace the 200 of Figure 6.1, if any. If the algorithm were to be adapted to suit this characteristic, it is very unlikely to be a fitting attribute of all images.

6.2 MEDIAN FILTER

Median filtering follows many of the same processes that encompass the moving average filter to reduce the noise in images. The window based system remains, as does the need to iterate through every pixel of the image. Equivalently the result is stored in the centre pixel, which is the pixel designated for noise reduction. The distinction of the median filter lies in how the replacement value is designated. Instead of summating and calculating a mean, all of the pixels of the window grid are sorted in ascending order (Thangavel, Manavalan & Aroquiaraj 2009). Sequencing the grid degrades performance moderately to O($n \log n$) (Sedgewick 1978). The median filter however is affected by the window sizing for exactly the same rationale as the moving average filter.

2	6	5
6	200	1
4	2	3

(A)

2	6	5
6	4	1
4	2	3

(B)

Figure 6.2 Median filter (A) Original window (B) Window after filtering.

Figure 6.2 exhibits the same test case as Figure 6.1 however with the median filter methodology. As in the former example, the value of 200 appears to be inconsistent in the context. The median of the nine cells of the window is the number four, which once more replaces the original value in Figure 6.2 (B). One leading motivation for using the median filter over the moving average filter is that it lessens the undesirable effects present in the latter filter (Leis 2011). Since the median filter ranks the neighbouring pixels rather than averaging them, a more precise result is formed. The outcome is as what was expected in the moving average filter; that is a value below ten. With this approach, much of the detail is preserved in both the replaced square and neighbouring pixels. Since the neighbouring pixels of the image are not adversely affected, the image retains its sharpness and clarity. In addition, an added benefit is that since the median filter is nonlinear, no setting of a threshold is necessary in an attempt to filter noise frequencies and thereby produce better results (Leis 2011).

6.3 ASSUMED DESIGN

Design aspects of the noise reduction filter build upon those constructed in the image alignment algorithm (refer to 5.4). Resembling the alignment algorithm, the choice of noise reduction algorithm was the foremost decision. The median filter was opted for as it offers satisfactory performance and of the options, provides the most precision reduction without adversely affecting the overall image clarity. The pixel values that constitute the window are gathered into local memory, with one array for each of the red, green and blue colour channels. The integrated Quicksort algorithm in C is utilised to sort the pixels in ascending order. The median of each of the arrays after sorting is complete and overwrites the existing values held for the image. Therefore alterations are immediate and global for all proceeding operations. Division among threads and consequently cores were conducted identically to the distribution for image alignment, whereby the vertical divider is calculated according to the number of cores. Each thread is given boundaries which it can process within.

Upon commencing development of the median filter algorithm, several issues are encountered. One issue confronted is the computation of the median filter around the edges of the image. In the example of the extreme leftmost side, three pixels down the left column of the median window are not accounted for. Three schemes intend to solve this, including:

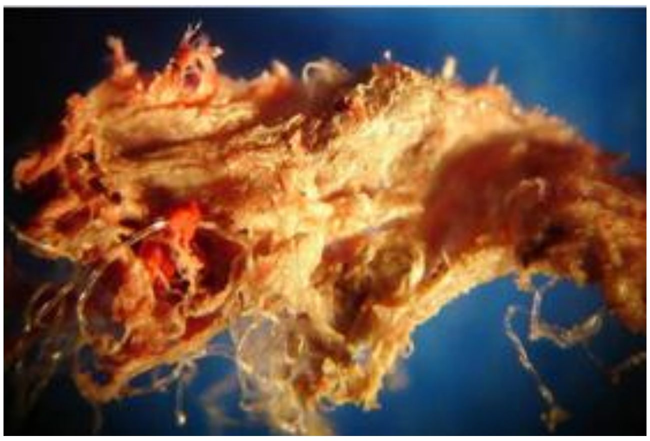
I avoiding the boundaries. This is achieved by commencing calculation at the first complete window. In the case of a larger 5×5 pixel window, the first complete window is formed at horizontal and vertical pixel 3. Computationally this is the fastest option, however the trade off is that the boundary does not have noise removed.

II shrinking the window at the boundaries. This involves gradually reducing of the window dimensions as it approaches the boundary, until the boundary is hit. At the edge, the window is condensed to a singular pixel. When this occurs, the median filter reduces to option I as there is only one pixel in the array. No noise can be reduced when there are not at least two values; a noisy one and a valid one.

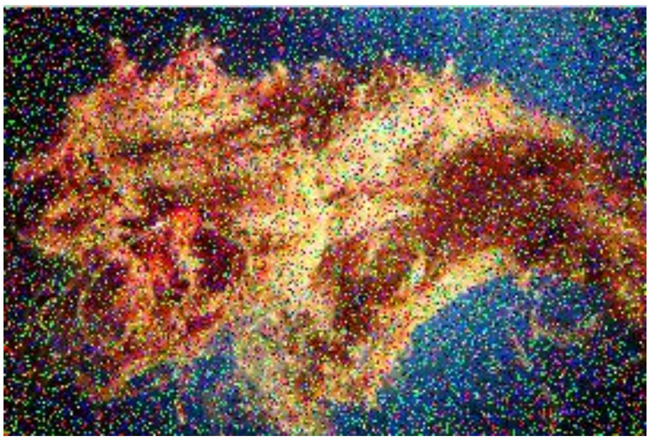
III fetching values from elsewhere to fill the window. Entries on the horizontal or vertical wrap around are prime targets. In the case of filtering the leftmost side, values on the rightmost side may be selected. Visibly this is only favourable when the images are to some extent palindromic at the edges. Otherwise the two sides may exhibit absolutely no similarity to each other and this could distort the output.

Taking into account that the end use is for medical science, the former of these options was desired. Primarily no inadvertent colourations or textures are introduced by blending two edges of an image together. Any peculiar patterns induced by the algorithm could potentially impair: the correct diagnosis; or the discovery of new materials. Clearly noise on the boundaries of the images is more acceptable to human users than discoloured or disfigured photographs. Conversely as Figure 6.3 (C) illustrates, often not all noise will entirely be removed by the filter across the image. Noise remnants remain in regions of high noise and will not be removed as the median of the window may in fact be noise itself. Overall the image preserves much of the clarity of the original image, found in Figure 6.3 (A).

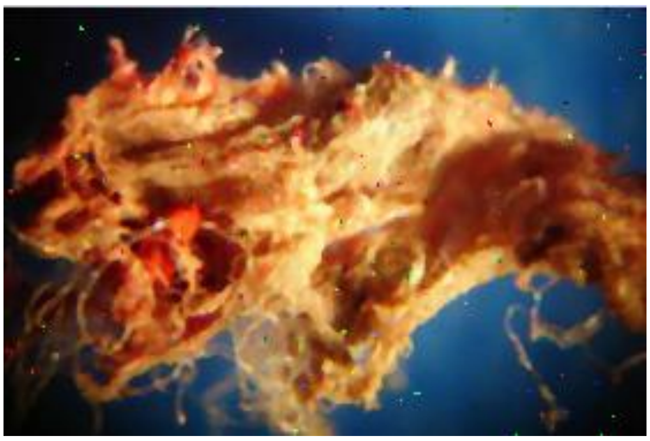
A disadvantage for the preference of the second option is that it is not the most computationally efficient of the alternatives. Efficiency is highly esteemed in this project as the various processes consume copiousness amounts of time. This facet is particularly applicable when it is considered that unlike image alignment, reductions in time cannot be developed by skipping iterations. Accuracy of the noise reduction in the images however takes precedence over the processing speed. Without accuracy, the project is not useful for its purpose. A direct advantage of not skipping computations is that it is not necessary to pass back over the samples to ensure the proper outcome.



(A)



(B)



(C)

Figure 6.3 Median filtering for noise reduction (A) Original image (B) Image disrupted with 'Salt 'n Pepper' style noise (C) Result of image after median filtering.

CHAPTER 7

PROJECT LIMITATIONS

It is a given that the majority of projects constructed will have functional, design or logistical limitations. The time involved in preparation, research and design are often critical factors restricting the overall outcome of the project. Breadth of the project scope and the extent of innovative developments furthermore add complexity to the project. Without exception, this project comprises several limitations. Constrained development time coupled with the concentration of the project relating to the parallelisation of image alignment and noise reduction left various prospective features overlooked. The major limitations are detailed below.

7.1 FILE FORMATS

It is evident that the algorithms developed will expect that the data acquired from the image files be in a universal arrangement in memory, so that the data can be readily accessed for manipulation. Whilst there are two types of acceptable input and output file formats developed throughout the project, only one is considered viable for quality and compatibility reasons. The foremost file type used is the Tagged Image File Format (TIFF). Designed in 1986 by Aldus Corporation, the purpose of TIFF was to have a unified format for images amongst different manufacturers (Adobe Developers Association 1992). It is informally known for the capability to house lossless photographic information in a common format, thereby leading to applications where quality loss is intolerable. There have been no major updates to the TIFF specification since 1992 and presently the TIFF format is widely supported by image manipulation programs and internet browsers alike. Nevertheless TIFF files can accommodate an almost unbounded number of compositions, making support for this type overly complex. The design of the TIFF module in this project is such that only selected configuration options are available. Restrictions include, but are not limited to:

I uncompressed, interlaced image data.
II greyscale or red–green–blue colour interpretations.
III 8 bits per colour channel.
IV fourth and subsequent colour channels ignored.

Simple modifications of the file input–output module would permit further third party file modules to be integrated. Provided that the additional file input–output module satisfies the functional conditions below, there should be no difficulties. A file format module must have methods to:

I read an image, given an open file pointer.
II write an image, given an open file pointer and an image object.
III return the width and height of the image.
V get and set a pixel at a given coordinate.
VII resize the image to given dimensions.

7.2 IMAGE SEQUENCING

The image alignment algorithm capitalises on the order the photographs are input to acquire additional performance. The path to each of the images is entered as separate arguments on the command line when the application is run. One photograph must be distinctively related to the next in the sequence entered, over any edge. Vast reductions in the amount of processing are to be reclaimed as the project does not have to compute the correlation for every combination of images. Inputting in a logical order results in performance of O$(n - 1)$, whereas the project would be penalised with a performance of O$(n !)$ to produce the image arrangement automatically. Figure 7.1 provides guidance on prospective ordering of a nine set of images. Note the arrangement of the centre row relative to the first and last rows.

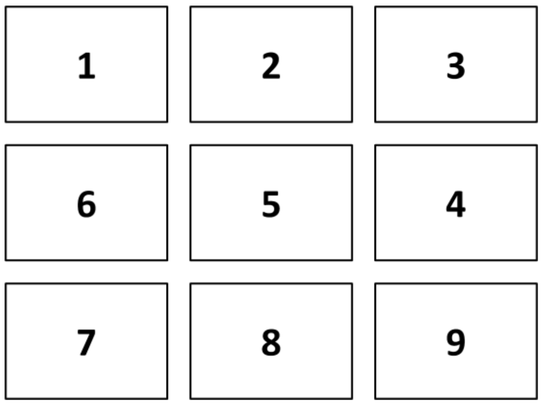


Figure 7.1 Sample ordering of a 3 X 3 image set

7.3 FILE PATHS

The path to files entered into the application represents a limitation. As the file paths are input on the command line with the `--input=file` switch, any spaces in the file path will be considered by the parser as multiple separate arguments. A file with two spaces in the file path will be registered by the parser as three arguments and so on. The program will therefore not accept files will spaces in the file path and will complain with a file not found error if present.

7.3 GRAPHICAL USER INTERFACE EXPANDABILITY

The project presently relies on the command line to interface with the user. Verbose status updates and errors are printed on the text console by default, unless deactivated with the `--quiet` option. Original intentions were to have an incorporated graphical user interface (GUI) to preview the output and make any fine adjustments before saving the panorama. Unfortunately there are no methods currently designed for implementing a GUI layer.

7.4 OPERATING SYSTEMS AND COMPLIATION

The implementation of the project application is restricted by a few isolated OS dependent function calls. Until further development redefines these OS dependant methods, it is mandatory that the operating system be Microsoft Windows XP or greater, or capable of running surrogate Windows instructions. It is also expected that the compiler be C standards compliant and evidently able to compile Windows executables. This limitation is acceptable in this setting since there are several free and compatible compilers and that Windows is installed on the majority of machines.

Primarily there are three OS dependant functions that prevent the software from being portable. The first developed function with the Windows OS limitation is the `getNumberOfProcessorCores()` method. The purpose of this function is to return the number of hardware cores, so that the corresponding number of threads can be initialised. In Windows to retrieve the number of cores, the `dwNumberOfProcessors` attribute of the `SYSTEM_INFO` structure must be accessed (*SYSTEM_INFO structure* 2011). In Linux, the corresponding command to list the number of processor cores is:

```
cat /proc/cpuinfo|grep processor|wc -l
```

Notably this instruction could be dissimilar on variants of Linux, however the concept remains. The final OS dependant function calls are found in the method `createThreadOnCore()`. Threads and semaphores for control are created with the Windows versions rather than using POSIX modules. Whilst calling POSIX modules is similar to the Windows counterparts and POSIX components are portable across OS's, Windows versions were selected since they were easier to read and maintain. Portability was deliberated extensively, but the prior limitation with the number of cores meant that portability was already constrained and as such POSIX threads were not necessary in this context.

CHAPTER 8

PERFORMANCE REVIEW

The overall performance of the prototype built in this project is a major factor in the effectiveness for use in a production situation. Besides reliability and consistency, comparative execution duration is the only measurable assessment for the realisation of the project objectives. As reasoned in the methodology (refer to 3.4), time is the unit selected for assessing the performance of the algorithms. The general populace can often relate to time in modern society. This makes comparison of performance straightforward for those outside of the fields of computers and electronics and thereby do not appreciate technical jargon.

8.1 TESTING SCENARIO

The performance of the project will be tested on various computer systems. In the practical performance testing of this system, only Windows Vista and Windows 7 machines could be utilised. There was no access to machines running older versions of the Windows OS to test. Periodically computer systems get updated and it becomes harder to find machines that continue to employ older software versions. Each system experimented on features an Intel processor and chipset, as unfortunately there were no AMD processors that could be tried for reference, since the majority of AMD chips are in low end machines (Burgess et al. 2011). Likewise, all systems had different levels of system memory size, frequency, latency and processor cache, so no direct conclusions can be drawn from the results concerning which of these hardware attribute confers the best processing times. This is acceptable since testing intends to demonstrate the performance benefits of multi–core algorithms on multi–core hardware, rather than provide an in depth overview of which hardware characteristics are the most advantageous.

Table 8.1 outlines the hardware used to test the project. Of note is the core counts row of Table 8.1 that displays the difference in hardware architectures. The newer Intel processors have the technology known as Hyper–Threading (HT). Essentially HT is hardware multi–threading. Within Windows, the core count returned is eight cores for System 1, as Windows does not make a discernable distinction between hardware processor cores and hardware threads.

Table 8.1 Hardware utilised for testing the project.

	System 1	**System 2**	**System 3**	**System 4**
Processor	Intel Core i7 930 @ 4.0 GHz	Intel Core i5 2400 @ 3.1 GHz	Intel Core i3 M330 @ 2.1 GHz	Intel Core 2 Duo T9300 @ 2.5 GHz
Core Counts	4 Processors 8 Threads	4 Processors 4 Threads	2 Processors 4 Threads	2 Processors 2 Threads
Memory	12 GB	4 GB	4 GB	2 GB
OS	Windows 7	Windows 7	Windows 7	Windows Vista

The test used two specifically modified images to repeatedly stress the algorithm and the hardware. The images both measure 250 × 250 pixels in dimension and as seen in Figure 8.1, the two arcs should combine to form a semicircle. This is ideal for testing of the project, as output errors are immediately obvious in the final panorama.

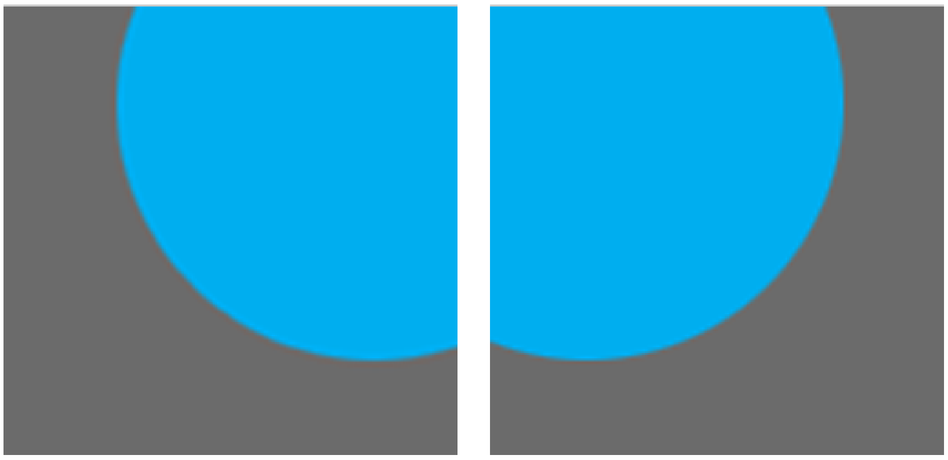


Figure 8.1 The two images used to stress the algorithm (A) The first arc (B) The second arc.

The procedure for testing the project is as follows:

I select the hardware to on which to perform the trials. The machine must have at least Microsoft Windows XP OS or better as discussed in the methodology (refer to 3.3). The only other requirement is that the test system must have a multi–core processor, lest the system not be able to properly assess the performance advantages.

II copy the binary program and image files to the test system. Placing the image files in the same directory as the binary file is recommended. The directory must not have spaces in the filename.

III call the software with the appropriate arguments. The appropriate call is

```
appname --input=arc1.tiff --input=arc2.tiff --threads=x
```

where

A `appname` is the application name. This will depend on the settings in the compiler when producing the binary file.

B `--input=x` is the image files to process. In this example, the files are named `arc1.tiff` and `arc2.tiff`. Each image file must be entered separately with a `--input=` preceding the filename.

C `--threads=x` is the number of threads to create. Replace `x` with the numeral of the desired value.

IV begin with a value of one for the number of threads. Record the time in seconds printed on the terminal at the end of processing.

V repeat the sequence III to IV various times to confirm consistency.

VI increment the amount of threads by one. Repeat the entire sequence III to V until the four threaded test is completed.

VII calculate the mean for each set of threads after all trials have been performed.

8.2 RESULTS

Table 8.2 is the tabulated data from testing the project over five iterations of each of the thread counts. Avoiding skewing the final conclusions, system 4 was not included in the mean time in the final row of Figure 8.3. The reason system 4 was not tested for three and four threads was that the computer was a dual core item and as such could not run the extra threads without conflicts in scheduling. The times for system 4 are included for reference.

Table 8.2 Time in seconds to render the alignment of the test images.

	Time (Seconds)			
	1 Thread	2 Threads	3 Threads	4 Threads
System 1	137.108	92.442	92.866	67.752
	137.347	100.386	92.914	69.501
	135.542	100.418	92.442	66.376
	136.664	100.387	93.068	67.168
	135.919	100.403	93.040	68.583
System 2	166.340	118.030	109.528	104.104
	165.491	119.608	112.245	105.211
	163.169	120.378	109.548	105.599
	163.893	119.585	109.717	105.370
	164.317	118.634	110.577	104.191
System 3	274.686	209.496	184.766	131.059
	273.409	206. 714	183.915	131.252
	274.089	207.408	184.085	129.331
	274.196	207.279	184.805	129.711
	273.800	207.028	183.774	129.432
System 4	246.382	157.700	–	–
	244.834	156.359	–	–
	246.016	156.127	–	–
	245.231	158.642	–	–
	245.097	156.322	–	–
Mean (No System 4)	**191.731**	**137.249**	**129.153**	**100.976**

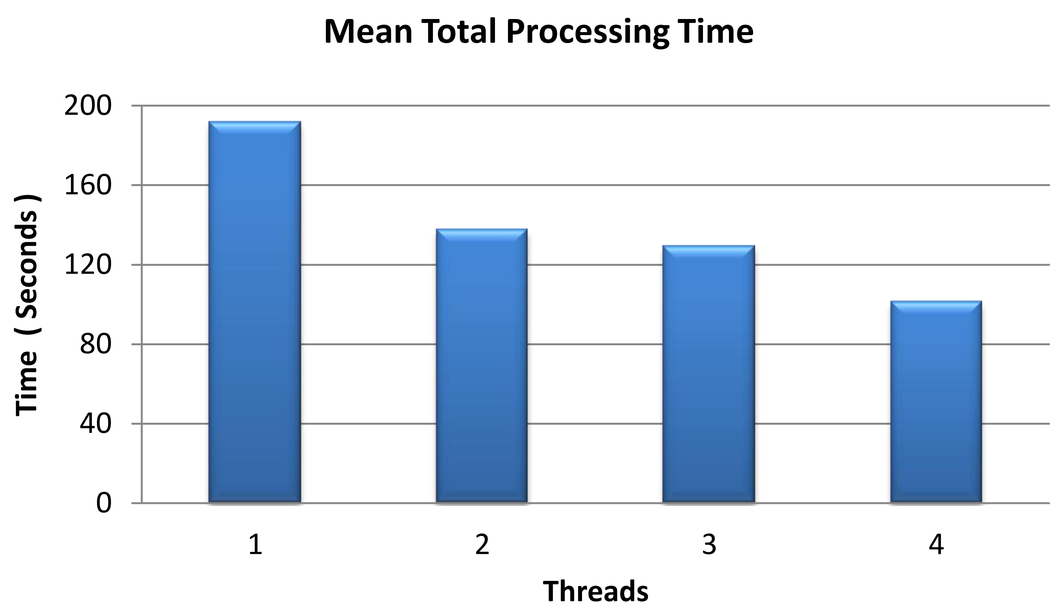


Figure 8.2 Graph of the tabulated data from Table 8.2.

8.3 DISCUSSION

The times that were drawn from the project application running on the test hardware were predicable. Adding successive threads to the program furnishes performance accelerations, thereby reducing the overall processing time. Unsurprisingly, the increases in processing times were disproportionate to the number of threads used. Table 8.2 illustrates that two threads delivered the largest divergence between threads, yielding 28.4 % faster processing time over one core. This translates to approximately 1.4 times the performance; not exactly the double that is instinctively thought would occur when using two cores. It is not until four threads are utilised that double performance is realised and the render time almost halves. Primarily the processing time is not linear and corresponding to the number of threads due to overheads. Nearly all of these are from initialisation practices. Some of these overheads include, but are not limited to:

I calculating the boundaries of a thread. A thread is not called until the range of values that the thread can process up to is determined. For simplicity, a starting value is assumed and an increment computed. The increment is the division of the task over the number of threads. The initial thread takes these parameters. Subsequent thread bounds are found by taking the previous bounds and adding the increment. This means that there is a delay between when the first threads are called and the latter threads are summoned.

II obtaining access to task a thread. Each thread is protected with a semaphore to prevent multiple functions attempting to be executed on the same thread concurrently. The semaphore creates a performance penalty by blocking the second and subsequent functions and making each wait. The time to signal the semaphore or set the semaphore to a blocking state moreover takes processor cycles.

III resuming or suspending a thread. During times where multiple threads are not utilised, the threads are suspended to save wasting processor cycles in idle. The OS scheduler will consequently pass over these threads, allowing processor allocations to be better managed for the other processes or threads. When the time comes to assign a function to evaluate, the thread must be resumed. The call to resume the thread consumes time.

IV the allocation of function resources. Each function that is multi–threaded requires a separate memory space to operate in. Local variables that are exclusive to the function must be duplicated for every thread, so that each thread does not overwrite the values of another thread. Time is used to perform variable creation and destruction of dynamic variables and verification that the construction of dynamic variables succeeds (Silberschatz, Galvin & Gagne 2009). The calling of the multi–threaded function from the threading module similarly increases the execution time. The cost in execution time of requesting the function, passing the parameters and managing the semaphores and other control variables all accumulates.

The results of testing the project permit inferences to be developed regarding the performance advantages of multi–core systems with high levels of processors. From previous annotations, it is reiterated that there will be an absolute minimum processing time that could be attained due to initialisation overheads. Considering the worst case of one pixel images, this inhibits the maximum useful thread count to one. With more threads than one, the results of all threads will be same. In this circumstance, the graph of the results would become horizontally linear for any amount of threads. The value of the line would be consistent with the absolute minimum processing time. Predicably this principle could be applied to hardware containing high levels of processing cores. Once the number of hardware cores exceeds the number of pixels in the image, no performance benefits can indisputably be obtained with the multi–threading algorithms presented in this project.

Adding further hardware cores moreover exhibits economies of scale. As established, two cores are faster than one. Depending on the application, four cores should have a lower processing time than one. The difference in the processing times of the additional cores progressively decreases, as observed with the processing times extracted from Figure 8.2. There becomes a point where whilst the further cores will decrease the processing times, the comparative time saving coupled with an escalation in hardware expenses does not justify the outgoings.

The computationally intensive image alignment task established some seemingly logical results concerning the hardware most appropriate to the fabrication of a panorama with the project. Whilst there are a number of different variables that contribute to the performance findings, all tests exhibited a decrease in processing time that was consistent with an increase in the processor frequency. For a known processor bound algorithm, this inference is reasonably sensible to conclude. Yet this deduction is rather speculation, as the fastest clocked processor in system 1 in the test moreover sported: the largest processor cache; the largest memory size;

and the lowest memory timings and latency. Any of these variables or combinations of them could respectively contribute to a decrease in the processing time.

The last computer listed in Table 8.2 as system 4, demonstrated a divergence from the results of the other machines. System 4 ran an approximate 10% improvement from one core to two than the next slowest system, system 3. It is unfortunate that the two machines do not share the same chipset and processor set, so that deductions could be made. In any case, it could be proposed that the choice of Windows OS may vary the render times. The hardware and software of system 4 is older, yet with both limitations the computer managed a better difference in processing times for the multi–core algorithm.

Untested is the impact of large image sizes on the performance of the application. Unless the hardware used has an abundance of physical memory, the memory allocations needed for the images in the algorithms will quickly exceed the amount of physical memory available. When this occurs, some of the data stored in memory will be paged to disk by the OS, decreasing system wide performance. The larger the images are and the more images used will increase the probability of exceeding physical memory. Evidently the algorithm has yet to consider these aspects. A conceivable approach to resolving low memory is to perform all functions progressively; however this alteration will influence algorithm performance negatively.

Time furthermore is not the best measurement for performance. As discussed in the methodology (refer to 3.4), time is the real world measurement with which the users of the project will judge the performance. The duration of the application is captured by the internal counter of the program for accuracy; however this counter is a simple utility. The counter does not account for the time where the OS scheduler has selected another process to operate the hardware. The result is that when the project spends periods of time not processing, the counter continues totalling the timing. In addition, the amount of time to process the same instructions in the project repeatedly will vary dependant on a number of factors. These factors include, but are not limited to:

I the instructions available to the processor. Hardware is unique and the architecture of the processor and memory subsystems is significant. Running a certain combination of software on specific hardware might produce different results than expected. The arrangement of particular series of commands or the use of a particular assembly routine might either aid or hinder performance on two analogous systems. Each hardware chipset has its own set of instructions. To achieve a typical function, some programming may be more efficient. If the system has hardware assistance for the commands to be utilised, the developer would instantaneously reduce the number of processor cycles to complete the same task, without further input.

II the number of actively competing processes and threads. The more processes waiting for time on the hardware, the slower the project will perform. The OS scheduler will reduce the allocated time permitted on the hardware for each thread according to the number of prepared and waiting processes or threads (Silberschatz, Galvin & Gagne 2009). Reducing the amount of active processes enables all processes to spend more processor cycles on the hardware and this is returned to the user by faster image alignment times.

III the effectiveness of the OS process scheduler. A scheduler that consistently has to fetch instructions paged on disk or that switches between processes too often will provide an environment that is relatively inefficient (Silberschatz, Galvin & Gagne 2009). In this instance, the overall performance of the project would not be equal to a system with a fast scheduler. Sections that are not properly optimised in the project will have the inefficiencies amplified in slower systems. These inefficiencies in both the OS scheduler and the project will detrimentally affect the processing time of the program. Observably these impacts will be more prevalent and recognisable in slower systems.

A marginal increase in performance is achieved by eliminating the inter–image blending module. Aesthetically the final panorama warrants such a feature, as the definitive contour around select images is visually unappealing. Furthermore differences in the light intensity or slight pattern mismatches may make the overlap seem rigid and abrupt. Due to compatibility issues, the project is void of such blending between photographs. Figure 8.3 displays the output from the tests. It is worth noting the non smooth semicircle particularly to the left of centre, as the two arcs join slightly off the midpoint. Blending in this situation would be beneficial. The advantage of not having such a feature is that execution time is slightly reduced.

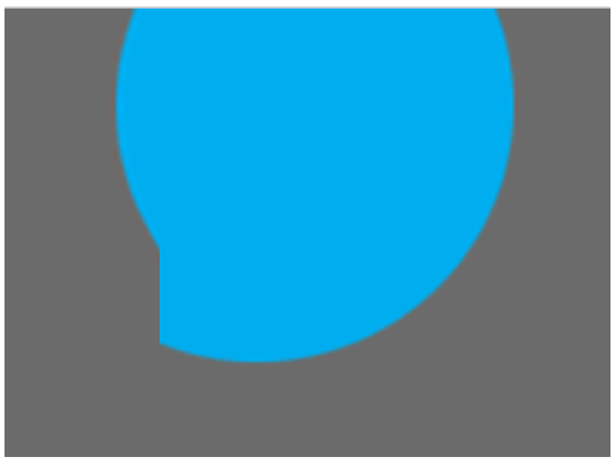


Figure 8.3 The output panorama after executing the program.

CHAPTER 9

CONCLUSIONS

Ultimately the project intended to resolve the issues associated with the existing system, which included the need to manually construct panoramas and the existing software not properly utilising multi–core processing capabilities of the hardware. In a generic sense, the project has satisfied the objectives. Throughout the duration of the design of the project, a potential solution has been developed to automate image alignment on multiple hardware processors. Furthermore the devised algorithms performed expectantly, contributing a reduction in the processing times for each additional thread. On a more thorough level, each of the objectives was completed to differing degrees.

9.1 SUMMARY OF DEVELOPMENTS AND FINDINGS

The design of the project is based on several computing concepts. The first component necessary for a multi–core panorama creation system is an approach to utilise the hardware processors. For this objective, there are two typical models: processes and threads. As outlined in the multi–core computing chapter, a process is a stream of instructions coupled with the resources to complete a task. Of the resources that a process contains, is at least one thread. A thread is a set of interrelated instructions known as a function, which performs a precise task. The instructions of a thread run on a singular hardware processor.

To utilise the performance advantages of multiple hardware processors, either multiple threads or multiple processes must be programmed. For a number of performance associated reasons, the project is multi–threaded. The scheme designed included using one thread per processor and dividing the computations of the algorithms equivocally across the threads. A theoretical acceleration close to a factor of the number of threads could be assumed by this approach; however research by Liu et al. (2010) concluded that initialisation overheads severely reduced the performance on small data sets. It is notable that the use of multi–threading is compliant with present programming ideologies.

The next component of the microscopy image stitching application was the decision of which algorithm to use for image alignment. The preferred algorithm was correlation, which numerically measures the similarity between two images. It was chosen out of the necessity for a self–contained calculation to determine similarity between two images without reliance on exterior resources. Accomplishment of the correlation function is done in three sections. First the correlation workload is divided, by taking the size of the images and dividing by the number of threads. Following this is the distribution of the task to each of the threads, which individually execute the correlation algorithm and return a result. Finally the best location is determined from all the responses of the threads and the images are collated into a single panoramic image.

The final component of the multi–core panoramic program is the noise reduction function. Noise relates to any variations between the captured pixels and the original subject. Obviously this component needs to be performed before any of the images enter the image alignment method and are collated. For noise reduction, the median filter algorithm was chosen over the moving average filter due to its clarity. The median filter operates by taking a window of pixels and arranging them in ascending order. The centre value is then selected to replace the designated pixel. Similar to the image alignment algorithm, the median filter is composed of a couple of sections. First the median filter workload is divided amongst the various threads. The tasks are subsequently distributed to all threads, which execute the respective assignment.

The results of the project are fitting according to prior work, as examined in the literature review (refer to 2.0). The trials of the application were carried out on varying hardware, but with the same two images. Nearly double the speed in processing time was found when utilising four threads compared to one. In this case, the average processing time decreased from approximately 192 seconds to 101 seconds, with no adjustments to the images. Two cores yielded a mere 28.4 % improvement in processing time over one core.

The performance difference between one core and two, confirmed the concept of overheads particularly with regard to the initialisation of functions. When a function starts, storage space in memory has to be allocated, costing valuable processing time. This cost can be perceived where the fastest acceleration in processing time was experienced with two cores over one. Multi–threading algorithms therefore do not produce performance advantages exactly equivalent to the number of hardware processors.

The results of the project displayed some fundamental characteristics of multi–core processing hardware and the associated algorithms. Programs that are designed with multi–threaded algorithms generate practical performance advantages on multi–core hardware over single–threaded algorithms. Moreover, the performance advantages can be gained without additional expenditure, whether it is hardware, software or monetary outgoings. This is the most noticeable benefit. The shortcoming of multi–threaded algorithms is however that the performance increase is by no means equal to the number of processors that the computer system has. In the days where processor accelerations were obtained by increasing the clock frequency, the software benefits were much more profound.

9.2 INITIAL RESEARCH OBJECTIVES

The first two objectives encompassed the research and investigation into existing image alignment and noise reduction algorithms. These objectives stated:

"I research into existing image alignment techniques and how these can be achieved through parallelisation.

II research and critical analysis of current noise removal algorithms and how they can be implemented through parallelisation."

The research conducted into these existing techniques was adequate however it lacked the comprehensiveness of a full algorithmic review. The sheer number of approaches meant that not every technique was covered in detail, primarily due to time and page constraints. Both the image alignment and noise reduction algorithms were to be multi–threaded, so devising an approach to adapt these algorithms to achieve parallelisation was essential. Techniques that restricted the ability of the algorithm to be multi–threaded or were known to be not as computationally efficient as counterparts were excluded from contention in the project. Whilst these decisions enabled development of a prototype to commence earlier, the elimination process may have left some potential candidates overlooked.

9.3 EVALUATION OF THE ALGORITHMS

In a similar manner to the first two objectives, the third objective was realised. This objective listed the evaluation of the performance of the researched approaches, stating:

"III investigate or otherwise evaluate the expected performance of the different approaches to ascertain the most efficient technique or techniques."

The performance of the algorithms shortlisted for development was outlined briefly in relation to the big–O notation. Big–O notation is a recognised system to approximate the number of repetitions in an algorithm, thus being a representative for the performance of the approach. The algorithms developed ultimately were not compared with other approaches in the same programming language, as time and resources were prohibitive. Nevertheless the algorithm designs could have been evaluated in this manner for the best results.

9.4 IMPLEMENTATION OBJECTIVE

The accomplishment of the fourth objective is somewhat subjective. The fourth objective was to design and implement a working prototype:

> *"IV design and implementation of a working prototype based on the best processing scheme."*

Undoubtedly a functional prototype was designed and compiled, allowing results to be gathered and analysed. In this regard the objective is realised. The subjectiveness is presented when considering the degree to which the prototype satisfies the current software practices. With no formal standard on how to write applications, the style of the program that is composed is purely related to the opinions of the author (refer to 2.5). The accomplishment of the coding of the project is deemed to be appropriate and complete. Programming constructs such as the placement of the opening `{` symbol in the C language are consistently coded throughout all the modules in the project. The project likewise is intended to preserve other respectable programming practices, such as minimising the code duplication.

9.5 REVIEW OBJECTIVE

The final objective of the project was to review the performance of the program on several computers:

"V review of the application performance on several differing types of machines and observe sections for improvement and optimisation."

Evidently the Intel processor hardware and the Microsoft Windows 7 OS were thoroughly tested. AMD processor hardware and Microsoft Windows XP were furthermore not tested at all. Unfortunately there were no AMD processors that could be tried for reference, since the majority of AMD chips are in low end machines. Windows XP was not trialled since computer systems are periodically updated and it becomes harder to find machines that continue to employ older software versions.

One prominent area to improve upon was the algorithm for image alignment. Whilst the alignment algorithm chosen fairly accurately determined the position of the two images, the performance of the algorithm was only satisfactory. The image alignment function was the most time consuming operation of the project when executing. Development of a custom algorithm dedicated to microscopy image alignment was desired, to reduce the total processing time. However this would require a longer timeframe for designing the algorithm and specific background knowledge of the subjects being researched at the CSIRO.

Although areas for improvement in the coding of the project can be clearly seen from the testing done, additional trails on more platforms could have revealed further sections for development. One of the points that may have been discovered is whether certain functions were dependant on hardware, software or a combination of both. Increased performance would have been achieved with a program written in assembly language.

A specially crafted algorithm suiting the alignment of specific images to hone the image join location could have been built. The purpose of such an algorithm would be to consume less processor cycles than the current implementation. Out of all functions developed, the image alignment algorithm is the most expensive in terms of computations. Any variations in this method will significantly alter the processing time. However as noted in the literature review (refer to 2.3), this outcome could only be accomplished with limited or zero error between captures. The simplest way of reducing the error is with an automated, motorised stage for image acquisition. Image acquisition recommendations however are beyond the scope of this project.

9.6 FURTHER WORK

Several aspects of the project could be developed further to enhance the application, enrich the experience for the user and contribute to pioneering research. Prevailing topic sections are outlined as follows.

I Fabrication of a GUI. As distinguished by the user, the addition of a GUI is paramount. It is anticipated that many computer users will fail to familiarise with a text based program. Usability of the project relies on the acceptance of the interface by the user and with a straightforward layout the project should complement the objectives of the consumer. Conversely the appendage of a GUI might degrade specific attributes of the project, in particular the overall efficiency of the program. Rendering graphics in real time on the monitor to display a preview of the panorama will produce memory and processor overheads, slowing the processing time accordingly.

II Refinements of algorithms. The alignment and noise reduction algorithms devised are a synthesis and evolution of recognised approaches that have been used for many years. Remodelling these algorithms or developing replacements to achieve faster processing times or a smaller memory size would prove beneficial. The correlation algorithm particularly consumes large periods of time. Performance improvements in this section will decrease the processing time considerably.

III Support for more file formats. This is advantageous if the image was not in a compatible format for this project. With more file format support, photographic data would not have to be transformed elsewhere before use in the project, thereby saving time. Besides convenience, commercially the project could be better positioned in the marketplace since it would be more flexible in the workflow of a diverse clientele.

LIST OF REFERENCES

Ramanathan, R 2006, *Intel Multi–Core Processors: Making the Move to Quad–Core and Beyond*, Intel Corporation, United States, accessed 27 August 2011, <http://www.intel.com/technology/architecture/downloads/quad-core-06.pdf>.

Cooper, L, Huang, K & Ujaldon, M 2011, ‘Parallel Automatic Registration of Large Scale Microscopic Images on Multiprocessor CPUs and GPUs’, *2011 IEEE International Parallel & Distributed Processing Symposium*, 16–20 May, pp. 1367–1376.

Collins, T 2007, ‘ImageJ for microscopy’, *BioTechniques*, vol. 43, no. 1, pp. 25–30.

Moreira, J, Midkiff, S & Gupta, M 1998, ‘A Comparison of Java, C/C++, and FORTRAN for Numerical Computing’, *IEEE Antennas and Propagation Magazine*, vol. 40, no. 5, pp. 102–105.

Xiong, Y & Pulli, K 2010, ‘Fast Image Stitching and Editing for Panorama Painting on Mobile Phones’, *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, 13–18 June, pp. 47–52.

Savitch, W 2010, *Absolute Java*, 4th edn, Pearson Education Incorporated, New Jersey, United States.

Oxford University Press 2010, ‘process’, Oxford Dictionaries, United States, accessed 24 August 2011, <http://oxforddictionaries.com/definition/process?rskey=kAayIh&result=1>.

Xia, Q & Zhang, Y 2010, ‘A Roust Algorithm of Constructing Panorama’, *2010 2nd International Conference on Networking and Digital Society*, vol. 2, 30–31 May, pp. 143–146.

Rankov, V, Locke, R, Edens, R, Barber, P & Vojnovic, B 2005, 'An algorithm for image stitching and blending', *Proceedings of SPIE*, vol. 5701, March, pp. 190-199.

Szeliski, R 2006, 'Image Alignment and Stitching: A Tutorial', *Foundations and Trends in Computer Graphics and Vision*, vol. 2, no. 1, January, pp. 1–104.

O'Donohue, D, Mills, S, Kingham, S, Bartie, P & Park, D 2008, 'Combined Thermal–LIDAR Imagery for Urban Mapping', *23rd International Conference on Image and Vision Computing New Zealand*, 26–28 November, pp. 1–6.

Adobe Systems Incorporated 2011, *Photoshop CS3 End User License Agreement*, Adobe Systems Incorporated, United States, accessed 20 July 2011, <http://labs.adobe.com/technologies/eula/photoshopcs3.html>.

Koponen, T 2006, 'Evaluation Framework for Open Source Software Maintenance', *International Conference on Software Engineering Advances*, October, pp. 52.

Messaoudii, C, Boudier, T, Sánchez–Sorzano, C & Marco, S 2007, 'TomoJ: new tools for electron tomography', *Proceedings of the Conference on ImageJ*, pp. 151–161.

ArcSoft Incorporated 2011, *Panorama Maker 5 Pro*, ArcSoft Incorporated, United States, accessed 15 April 2011, <http://www.arcsoft.com/estore/software_title.asp?ProductCode=PMK5PRO>.

Autodesk Incorporated 2011, *Autodesk Stitcher Unlimited*, Autodesk Incorporated, United States, accessed 15 April 2011, <http://usa.autodesk.com/adsk/servlet/pc/index?id=11390049&siteID=123112>.

Zhang, N & Wang, J & Chen, Y 2010, 'Image Parallel Processing Based on GPU', *2010 2nd International Conference on Advanced Computer Control*, vol. 3, 27–29 March, pp. 367–370.

Wang, B & Wu, T & Yan, F & Li, R & Xu, N & Wang, Y 2009, ‘RankBoost Acceleration on both NVIDIA CUDA and ATI Stream Platforms’, *2009 15th International Conference on Parallel and Distributed Systems (ICPADS)*, 8–11 December, pp. 284–291.

Blythe, D 2008, ‘Rise of the Graphics Processor’, *Proceedings of the IEEE*, vol. 96, no.5, May, pp.761–778.

Yuffe, M, Knoll, E, Mehalel, M, Shor, J & Kurts, T 2011, ‘A fully integrated multi–CPU, GPU and memory controller 32nm processor’, *2011 IEEE International* on *Solid-State Circuits Conference Digest of Technical Papers,* 20–24 February, pp. 264–266.

Eytani, Y & Ur, S 2004, ‘Compiling a benchmark of documented multi-threaded bugs’, *Proceedings of the 18th International Parallel and Distributed Processing Symposium,* 26–30 April, pp. 266.

Cubranic, D & Booth, K 1999, ‘Coordinating Open–Source Software Development’, *Proceedings of the IEEE 8th International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pp.61–66.

Fogel, K 2006, *Producing Open Source Software: How to Run a Successful Free Software Project*, O’Reilly Media Incorporated, United States.

Liu, T, Ji, Z, Wang, Q & Zhu, S 2010, ‘Research on Efficiency of Signal Processing on Embedded Multicore System’, *2010 First International Conference on Pervasive Computing Signal Processing and Applications,* 17–19 September, pp. 907-911.

Yatsuzuka, Y, Hosoda, K, Iizuka, S, Kawaguchi, S & Shinbo, A 1988, ‘High–performance ADPCM codec for voice and voiceband data and its application to DCME’, *IEEE International Conference on Communications*, vol. 1, 12–15 June, pp.400–407.

Sun, X, Byna, S & Holmgren, D 2009, ‘Modeling Data Access Contention in Multicore Architectures’, *2009 15th International Conference on Parallel and Distributed Systems*, 8–11 December, pp. 213–219.

Xing, J & Miao, Z 2007, ‘An Improved Algorithm on Image Stitching based on SIFT features’, *2nd International Conference on Innovative Computing, Information and Control*, 5–7 September, pp. 453.

Hsieh, J 2003, ‘Fast Stitching Algorithm for Moving Object Detection and Mosaic Construction’, *Proceedings of the 2003 International Conference on Multimedia*, vol. 1, 6–9 July, pp. 85–8.

Chen, C 1998, ‘Image Stitching - Comparisons and New Techniques’, *Technical Report CITR-TR-30*, Computer Science Department, The University of Auckland, New Zealand.

Jia, J & Tang, C 2008, ‘Image Stitching Using Structure Deformation’, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 30, no. 4, April, pp. 617–631.

Hua, Z, Li, Y & Li, J 2010, ‘Image Stitch Algorithm Based on SIFT and MVSC’, *2010 Seventh International Conference on Fuzzy Systems and Knowledge Discovery*, vol. 6, 10–12 August, pp. 2628–2632.

Ryu, J, Lee, C & Park, H 2011, ‘Formula for Harris corner detector’, *Electronics Letters*, vol. 47, no. 3, February 3, pp. 180–181.

Thangavel, K, Manavalan, R & Aroquiaraj, L 2009, ‘Removal of Speckle Noise from Ultrasound Medical Image based on Special Filters: Comparative Study’, *ICGST-GVIP*, vol. 9, no. 3, June, pp. 25–32.

Srivastava, R 2010, ‘Restoration of fluorescence microscopic images using a nonlinear PDE based filter’, *2010 Annual IEEE India Conference,* 17–19 December, pp. 1–4.

Kemerer, C & Paulk, M 2009, ‘The Impact of Design and Code Reviews on Software Quality: An Empirical Study Based on PSP Data’, *IEEE Transactions on Software Engineering,* vol. 35, no. 4, July–August, pp. 534–550.

Boogerd, C & Moonen, L 2008, ‘Assessing the value of coding standards: An empirical study’, *IEEE International Conference on Software Maintenance*, 28 September–4 October, pp. 277–286.

Kremenek, T, Ashcraft, K, Yang, J & Engler, D 2004, ‘Correlation Exploitation in Error Ranking’, *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, vol. 29, no. 6, November, pp. 83–93.

Wang, Y, Wang, S, Li, X, Li, H & Du, J 2010, ‘Identifier Naming Conventions and Software Coding Standards: A Case Study in One School of Software’, *2010 International Conference on Computational Intelligence and Software Engineering*, 10–12 December, pp. 1–4.

Fang, X 2001, ‘Using a coding standard to improve program quality’, *Proceedings of the 2nd Asia–Pacific Conference on Quality Software*, pp. 73–78.

Mark, D 2009, *Learn C on the Mac*, Springer–Verlag, New York, United States.

Salomon, D 2002, *A Guide to Data Compression Methods*, Springer–Verlag, New York, United States.

Xin, C 2009, ‘Music and Image Applications of Mobile Phone Serious Game’, *International Conference on Environmental Science and Information Application Technology*, vol. 2, 4–5 July, pp. 510–513.

Jackson, D & Hannah, S 1993, ‘Comparative Analysis of Image Compression Techniques’, *Proceedings of the Twenty-Fifth Southeastern Symposium on System Theory*, 7–9 March, pp. 513–517.

Neelamani, R, de Queiroz, R, Fan, Z, Dash, S & Baraniuk, R 2006, ‘JPEG Compression History Estimation for Color Images’, *IEEE Transactions on Image Processing*, vol. 15, no. 6, pp. 1365–1378.

Jefferis, G 2004, *DM3 Image Format*, National Institutes of Health, United States, accessed 5 Semptember 2011, <http://rsbweb.nih.gov/ij/plugins/DM3Format.gj.html>.

CompuServe Incorporated 1990, *GRAPHICS INTERCHANGE FORMAT*, World Wide Web Consortium (W3C), United States, accessed 14 July 2011, <http://www.w3.org/Graphics/GIF/spec-gif89a.txt>.

MacKenzie, S 1988, ‘A structured approach to assembly language programming’, *IEEE Transactions on Education*, vol. 31, no. 2, pp. 123–128.

Microsoft Support 2007, *System requirements for Windows XP operating systems*, Microsoft Corporation, United States, accessed 4 July 2011, <http://support.microsoft.com/kb/314865>.

Microsoft Developer Network 2011, *clock*, Microsoft Corporation, United States, accessed 10 September 2011, <http://msdn.microsoft.com/en-us/library/4e2ess30%28v=vs.71%29.aspx>.

Bridges, M, Vachharajani, N, Zhang, Y, Jablin, T & August, D 2007, *Revisiting the Sequential Programming Model for Multi-Core*, Princeton University, United States, accessed 7 July 2011, <liberty.princeton.edu/Publications/micro40_scale.pdf>.

Silberschatz, Galvin and Gagne (2009), *Operating System Concepts*, 8th edn, John Wiley & Sons Incorporated, Jefferson City, United States.

Hughes, C & Hughes, T 2008, *Professional multicore programming: Design and implementation for C++ developers*, Wiley Publishing, United States.

Bovet, D & Cesatí, M 2006, *Understanding the Linux Kernel*, 3rd edn, O'Reilly Media Incorporated, United States.

Microsoft Developer Network 2011, *SetProcessAffinityMask function*, Microsoft Corporation, United States, accessed 16 June 2011, <http://msdn.microsoft.com/en-us/library/windows/desktop/ms686223%28v=vs.85%29.aspx>.

Akhter, S & Roberts, J 2006, *Multi-core programming: Increasing performance through software multithreading*, Intel Corporation, United States.

Lee, K, H, Pham, Kim, H, Youn, H & Song, O 2011, 'A Novel Predictive and Self–Adaptive Dynamic Thread Pool Management', *2011 IEEE 9th International Symposium on Parallel and Distributed Processing with Applications*, 26–28 May 2011, pp. 93–98.

Brown, LG 1992, 'A Survey of Image Registration Techniques', *ACM Computing Surveys*, vol. 24, no. 4, December, pp. 325–376.

Zeilik, M & Gregory, S 1998, *Introductory Astronomy and Astrophysics*, Saunders College Publishing, United States.

Lowe, D 1999, 'Object Recognition from Local Scale–Invariant Features', *The Proceedings of the Seventh IEEE International Conference on Computer Vision*, vol. 2, pp. 1150–1157.

Stroebel, L & Zakia, R 1993, *The Focal Encyclopedia of Photography*, 3rd edn, Butterworth–Heinemann, Woburn, United States.

Mather, P 2004, *Computer processing of remotely sensed images: an introduction*, John Wiley & Sons Incorporated, West Sussex, England.

Sedgewick, R 1978, 'Implementing Quicksort programs', *Communications of the ACM*, vol. 21, no. 10, October, pp. 847–857.

Leis, J 2011, *Digital Signal Processing Using MATLAB for Researchers and Students*, John Wiley & Sons Incorporated, New Jersey, United States.

Adobe Developers Association 1992, *TIFF Revision 6.0*, Adobe Systems Incorporated, United States, accessed 1 August 2011, <http://partners.adobe.com/public/developer/en/tiff/TIFF6.pdf>.

Microsoft Developer Network 2011, *SYSTEM_INFO structure*, Microsoft Pty Ltd, United States, accessed 5 October 2011, <http://msdn.microsoft.com/en-us/library/windows/desktop/ms724958%28v=vs.85%29.aspx>.

Burleson, D 2002, *Oracle9i High–Performance Tuning with STATSPACK*, McGraw–Hill/Osborne, California, United States.

Burgess, B, Cohen, B, Denman, M, Dundas, J, Kaplan, D & Rupley, J 2011, 'Bobcat: AMD's Low-Power x86 Processor', *IEEE Micro*, vol. 31, no. 2, pp. 16–25.

APPENDIX A

SPECIFICATIONS

University of Southern Queensland

FACULTY OF ENGINEERING AND SURVEYING

ENG4111 / ENG4112 Research Project
PROJECT SPECIFICATION

FOR:	**TRISTAN WARD**
TOPIC:	MULTICORE ALGORITHMS FOR IMAGE ALIGNMENT
SUPERVISORS:	Dr. John Leis
PROJECT AIM:	This project seeks to investigate and implement the parallelising of image alignment and noise removal algorithms on a multicore CPU, for the purpose of forming prompt microscopy panoramic images.

PROGRAMME: **Issue A, 10th March 2011**

1. Research image alignment techniques and how these can be achieved through parallelisation; using a multicore processor.
2. Research and critically investigate existing noise removal algorithms and how they can be implemented through parallelisation.
3. Evaluate the performance of the different approaches to determine the most efficient process.
4. Design and implement a working prototype based on the best processing technique.
5. Review the performance of the application on several differing types of machines, and observe areas for improvement and optimisation.

As time permits:

6. Design a simple user interface to manually adjust the image alignment if processing is not exact or fails.

AGREED:

_________________ (Student) _________________ (Supervisor)

___ / ___ / ___ ___ / ___ / ___

APPENDIX B

CODE LISTINGS

B1 MAIN.C

```
//------------------------------------------------------------------------------
// Description:   Multicore microscopy panorama image processing
//-------------------------------------------------------------------------------

#include <stdarg.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include "alignment.h"
#include "boolean.h"
#include "error.h"
#include "fileio.h"
#include "noisereduction.h"
#include "threads.h"

//------------------------------------------------------------------------------
int main(int argc, char* argv[]) {
//------------------------------------------------------------------------------
    unsigned short numberOfThreads = 100;
    unsigned short processorOffset = 0;
    short alignmentStep = -1;
    short alignmentTolerance = -1;
    short noiseFilterSize = -1;
    unsigned short argIterator = 1;
    unsigned short numberOfInputs = 0;
    BOOLEAN_TYPE outputFileSet = FALSE;
    clock_t startProgram = clock();
    clock_t startSection;
    char statusBuffer[100];

    // Start the program with verbose output
    setVerboseMode(TRUE);
    setFileOutput("compiled_image.tiff");
    // Determine which command line arguments are given and process them
    if (argc >= 2) {
        while (argIterator < argc) {
            if (strncmp(argv[argIterator], "--help", 6) == 0) {
                argc = 0;
            }
            else if (strncmp(argv[argIterator], "--input=", 8) == 0) {
                if (setFilePath(strndup(argv[argIterator] + 8, 1024))) {
                    numberOfInputs++;
                }
            }
            else if (strncmp(argv[argIterator], "--output=", 9) == 0) {
                setFileOutput(strndup(argv[argIterator] + 9, 1024));
                outputFileSet = TRUE;
            }
            else if (strncmp(argv[argIterator], "--threads=", 10) == 0) {
                numberOfThreads = atoi(strndup(argv[argIterator] + 10, 5));
            }
            else if (strncmp(argv[argIterator], "--offset=", 9) == 0) {
                processorOffset = atoi(strndup(argv[argIterator] + 9, 5));
            }
            else if (strncmp(argv[argIterator], "--align-step=", 13) == 0) {
                alignmentStep = atoi(strndup(argv[argIterator] + 13, 5));
            }
            else if (strncmp(argv[argIterator], "--tolerance=", 12) == 0) {
                alignmentTolerance = atoi(strndup(argv[argIterator] + 12, 5));
            }
            else if (strncmp(argv[argIterator], "--filter-size=", 14) == 0) {
                noiseFilterSize = atoi(strndup(argv[argIterator] + 14, 5));
            }
            else if (strcmp("--quiet", argv[argIterator]) == 0) {
                setVerboseMode(FALSE);
            }
            else {
                sprintf(statusBuffer, "Unknown argument \"%s\". Ignoring...\n",
                    argv[argIterator]);
                warning(statusBuffer);
            }
            argIterator++;
        }
    }
    // Display help if requested or no files input
    if (argc < 3 || numberOfInputs < 2) {
        warning("  \nUsage: app --input=file --input=file\n");
        warning("  Manatory arguments:");
        warning("    --input=file       The image \"file\" to process.\n");
        warning("  Optional arguments:");
        warning("    --output=file      Save the image output as \"file\".");
        warning("    --threads=x        Limit the number of threads to x.");
        warning("    --offset=x         Offset the first thread by x "
            "processors.");
        warning("    --align-step=x     Adjust the increment between alignment "
            "tests to x pixels.");
        warning("    --tolerance=x      Apply the tolerance of x pixels to "
            "record as valid.");
        warning("    --filter-size=x    Set the noise reduction filter to x "
            "pixels.");
        warning("    --help             Show this usage help.\n");
        return 0;
    }
    // Setup threads
    startSection = clock();
    status("Setting up core features...");
    createMultipleThreads(numberOfThreads, processorOffset);
    sprintf(statusBuffer, "Core setup complete in %g seconds.\n",
            (double)(clock() - startSection) / CLOCKS_PER_SEC);
    status(statusBuffer);
    // Read in and buffer images
    startSection = clock();
    status("Starting read of image files...");
    readAllImageFiles();
    sprintf(statusBuffer, "Image imports complete in %g seconds.\n",
            (double)(clock() - startSection) / CLOCKS_PER_SEC);
    status(statusBuffer);
    // Remove noise from the images
    if (noiseFilterSize != 0) {
        startSection = clock();
        status("Initiating image noise reduction...");
        reduceNoise(noiseFilterSize);
        sprintf(statusBuffer, "Noise reduction complete in %g seconds.\n",
                (double)(clock() - startSection) / CLOCKS_PER_SEC);
        status(statusBuffer);
    }
    // Begin to align the images
    startSection = clock();
    status("Commencing alignment of images...");
    alignImages(alignmentStep, alignmentTolerance);
    sprintf(statusBuffer, "Alignment of images complete in %g seconds.\n",
            (double)(clock() - startSection) / CLOCKS_PER_SEC);
    status(statusBuffer);
    // Save the finished file
    startSection = clock();
    status("Exporting compiled output image...");
    writeImageFile();
    sprintf(statusBuffer, "Saving output complete in %g seconds.\n",
            (double)(clock() - startSection) / CLOCKS_PER_SEC);
    status(statusBuffer);
    // Return the status
    sprintf(statusBuffer, "All functions complete in %g seconds.",
            (double)(clock() - startProgram) / CLOCKS_PER_SEC);
    status(statusBuffer);
    return 0;
}
```

B2 ALIGNMENT.H

```
//--------------------------------------------------------------------------------
// alignment.h - header file for detecting and positioning images
//--------------------------------------------------------------------------------
#ifndef ALIGNMENT_H
#define ALIGNMENT_H
//--------------------------------------------------------------------------------

void alignImages(int stepping, int toleranceSize);
// Description:    Finds the alignment point and compiles all of the images
// Inputs:         The size of the increment steps between correlation calculation
//                 amount of tolerance to record a specific value
// Returns:        None

#endif
```

B3 ALIGNMENT.C

```
//--------------------------------------------------------------------------------
// alignment.c - implementation file for detecting and positioning images
//--------------------------------------------------------------------------------

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include "alignment.h"
#include "axis.h"
#include "fileio.h"
#include "threads.h"
#include "boolean.h"
#include "error.h"

#define DEFAULT_TOLERANCE 10
#define DEFAULT_STEP_SIZE 3
#define INDEX_BASE 0
#define INDEX_SHIFT 1

//--------------------------------------------------------------------------------
// Structures
//--------------------------------------------------------------------------------

typedef struct {
    COORDINATE offset;
    double matchQuality;
}
MATCH_COORDINATE;

typedef struct {
    GENERIC_RGB* imageData;
    unsigned long imageHeight;
    unsigned long imageWidth;
    COORDINATE overlap;
    unsigned long xOverlaps;
    unsigned long yOverlaps;
    COORDINATE base;
}
```

```
LOCAL_IMAGE;

//------------------------------------------------------------------------------
// Global Variables
//------------------------------------------------------------------------------

static LOCAL_IMAGE localImage;
static int steppingSize;
static int tolerance;

//------------------------------------------------------------------------------
double average(GENERIC_RGB value) {
//------------------------------------------------------------------------------
    return (double)(value.red + value.green + value.blue) / 3;
}

//------------------------------------------------------------------------------
void seekOffset(IMAGE_LIMITS base, IMAGE_LIMITS shift, char* resultData) {
//------------------------------------------------------------------------------
    MATCH_COORDINATE bestMatchPoint;
    char* bestMatchPointPtr = (char*)&bestMatchPoint;
    RECTANGLE viewport;
    LIMITS loopLimits;
    unsigned long height[2] = {getFileHeight(base.imageIndex),
        getFileHeight(shift.imageIndex)};
    unsigned long width[2] = {getFileWidth(base.imageIndex),
        getFileWidth(shift.imageIndex)};
    unsigned long xOverlaps = 0;
    unsigned long yOverlaps = 0;
    unsigned long viewportSize;
    short stepSize = steppingSize;
    double baseMean;
    double baseMeanDeviation;
    double* baseDeviation;
    double shiftMean;
    double shiftMeanDeviation;
    double* shiftDeviation;
    double meanDeviation;

    // Allocate the arrays
    baseDeviation = malloc(height[INDEX_BASE] * width[INDEX_BASE] *
        sizeof(double));
```

```
shiftDeviation = malloc(height[INDEX_BASE] * width[INDEX_BASE] *
    sizeof(double));
if (baseDeviation == NULL || shiftDeviation == NULL) {
    errorAndTerminate("Insufficient memory to allocate for alignment "
        "buffers.", INSUFFICIENT_MEMORY);
}
// Initialise variables
bestMatchPoint.matchQuality = 0;
bestMatchPoint.offset.x = 0;
bestMatchPoint.offset.y = 0;
loopLimits.min.x = shift.limits.min.x;
if (abs(loopLimits.min.x) == width[INDEX_SHIFT]) {
    loopLimits.min.x++;
}
loopLimits.min.y = shift.limits.min.y;
if (abs(loopLimits.min.y) == height[INDEX_SHIFT]) {
    loopLimits.min.y++;
}
loopLimits.max.x = base.limits.max.x;
if (loopLimits.max.x == width[INDEX_BASE]) {
    loopLimits.max.x--;
}
loopLimits.max.y = base.limits.max.y;
if (loopLimits.max.y == height[INDEX_BASE]) {
    loopLimits.max.y--;
}
for (short depth = 0; depth < 2; depth++) {
    for (long lineX = loopLimits.min.x; lineX < loopLimits.max.x;
        lineX += stepSize) {
        // Determine the number of overlaps in the x axis
        xOverlaps = width[INDEX_SHIFT];
        viewport.upperLeft.x = 0;
        viewport.lowerRight.x = width[INDEX_BASE] - 1;
        if (lineX < 0) {
            xOverlaps = width[INDEX_SHIFT] + lineX;
            viewport.lowerRight.x = xOverlaps;
        }
        else if (width[INDEX_SHIFT] + lineX > width[INDEX_BASE]) {
            xOverlaps = width[INDEX_BASE] - lineX;
            viewport.upperLeft.x = lineX;
        }
        for (long lineY = loopLimits.min.y; lineY < loopLimits.max.y;
            lineY += stepSize) {
            // Determine the number of overlaps in the y axis
            yOverlaps = height[INDEX_BASE];
            viewport.upperLeft.y = 0;
            viewport.lowerRight.y = height[INDEX_BASE] - 1;
            if (lineY < 0) {
                yOverlaps = height[INDEX_SHIFT] + lineY;
                viewport.lowerRight.y = yOverlaps;
            }
            else if (height[INDEX_SHIFT] + lineY > height[INDEX_BASE]) {
                yOverlaps = height[INDEX_BASE] - lineY;
                viewport.upperLeft.y = lineY;
            }
            // Calculate the mean of the current overlap region
            baseMean = 0;
            shiftMean = 0;
            for (long x = viewport.upperLeft.x;
                x < viewport.lowerRight.x; x++) {
                for (long y = viewport.upperLeft.y;
                    y < viewport.lowerRight.y; y++) {
                    // Sum the base image
                    baseMean += average(getPixel(base.imageIndex, x, y));
                    // Sum the shiftable image
                    shiftMean += average(getPixel(shift.imageIndex,
                        x - lineX, y - lineY));
                }
            }
            // Divide by the size of the viewport to obtain the mean
            viewportSize = (viewport.lowerRight.x - viewport.upperLeft.x) *
                (viewport.lowerRight.y - viewport.upperLeft.y);
            baseMean /= viewportSize;
            shiftMean /= viewportSize;
            // Evaluate the mean absolute deviation
            baseMeanDeviation = 0;
            shiftMeanDeviation = 0;
            for (long x = viewport.upperLeft.x; x < viewport.lowerRight.x;
                x++) {
                for (long y = viewport.upperLeft.y;
                    y < viewport.lowerRight.y; y++) {
                    long arrayOffset = (y - viewport.upperLeft.y) *
                        xOverlaps + x - viewport.upperLeft.x;
```

```
                baseDeviation[arrayOffset] = average(getPixel(
                    base.imageIndex, x, y)) - baseMean;
                baseMeanDeviation += baseDeviation[arrayOffset] *
                    baseDeviation[arrayOffset];
                shiftDeviation[arrayOffset] = average(getPixel(
                    shift.imageIndex, x - lineX, y - lineY)) - shiftMean;
                shiftMeanDeviation += shiftDeviation[arrayOffset] *
                    shiftDeviation[arrayOffset];
            }
        }
        baseMeanDeviation = sqrt(baseMeanDeviation / viewportSize);
        shiftMeanDeviation = sqrt(shiftMeanDeviation / viewportSize);
        // Compare the similarity of the mean deviations
        meanDeviation = 0;
        if (baseMeanDeviation * shiftMeanDeviation > tolerance) {
            for (long x = viewport.upperLeft.x;
                x < viewport.lowerRight.x; x++) {
                for (long y = viewport.upperLeft.y;
                    y < viewport.lowerRight.y; y++) {
                    long arrayOffset = (y - viewport.upperLeft.y) *
                        xOverlaps + x - viewport.upperLeft.x;

                    meanDeviation += (baseDeviation[arrayOffset] *
                        shiftDeviation[arrayOffset]) /
                                (baseMeanDeviation * shiftMeanDeviation);
                }
            }
            // Store the offsets if a greater match than existing
            if (bestMatchPoint.matchQuality < meanDeviation){
                bestMatchPoint.matchQuality = meanDeviation;
                bestMatchPoint.offset.x = lineX;
                bestMatchPoint.offset.y = lineY;
            }
        }
    }
}
// Adjust parameters to search around the best point for a better match
loopLimits.min.x = bestMatchPoint.offset.x - stepSize + 1;
if (loopLimits.min.x < 0) {
    loopLimits.min.x = 0;
}
loopLimits.min.y = bestMatchPoint.offset.y - stepSize + 1;
```

```
        if (loopLimits.min.y < 0) {
            loopLimits.min.y = 0;
        }
        loopLimits.max.x = bestMatchPoint.offset.x + stepSize - 1;
        if (loopLimits.max.x > width[INDEX_BASE]) {
            loopLimits.max.x = width[INDEX_BASE];
        }
        loopLimits.max.y = bestMatchPoint.offset.y + stepSize - 1;
        if (loopLimits.max.y > height[INDEX_BASE]) {
            loopLimits.max.y = height[INDEX_BASE];
        }
        stepSize = 1;
        if (depth == 0) {
            status("Potential alignment region found.");
        }
    }
    // Copy the results
    for (short size = 0; size < sizeof(MATCH_COORDINATE); size++) {
        resultData[size] = bestMatchPointPtr[size];
    }
    // Free dynamic memory
    free(baseDeviation);
    free(shiftDeviation);
}

//--------------------------------------------------------------------------------
void compile(IMAGE_LIMITS base, IMAGE_LIMITS shift, char* reserved) {
//--------------------------------------------------------------------------------
    long xMin[2] = {base.limits.min.x, shift.limits.min.x};
    long xMax[2] = {base.limits.max.x, shift.limits.max.x};
    long yMin[2] = {base.limits.min.y, shift.limits.min.y};
    long yMax[2] = {base.limits.max.y, shift.limits.max.y};

    for (long x = xMin[INDEX_SHIFT]; x < xMax[INDEX_SHIFT]; x++) {
        for (long y = yMin[INDEX_SHIFT]; y < yMax[INDEX_SHIFT]; y++) {
            long arrayOffset = (y + abs(localImage.overlap.y))  *
                localImage.imageWidth + (x + abs(localImage.overlap.x));

            // Copy the image data blocks
            localImage.imageData[arrayOffset] = getPixel(shift.imageIndex,x,y);
        }
    }
```

```
}

//------------------------------------------------------------------------------
void transferLocalData(IMAGE_LIMITS base, IMAGE_LIMITS shift, char* reserved) {
//------------------------------------------------------------------------------
    unsigned long height = getFileHeight(shift.imageIndex);
    unsigned long width = getFileWidth(shift.imageIndex);

    // Sanity check the bounds
    if (shift.limits.min.x < 0 || shift.limits.min.y < 0 ||
        shift.limits.max.x < 0 || shift.limits.max.y < 0 ||
        shift.limits.min.x > width || shift.limits.min.y > height ||
        shift.limits.max.x > width || shift.limits.max.y > height) {
        return;
    }
    // Copy the image data
    for (long x = shift.limits.min.x; x < shift.limits.max.x; x++) {
        for (long y = shift.limits.min.y; y < shift.limits.max.y; y++) {
            setPixel(shift.imageIndex, x, y,
                localImage.imageData[y * localImage.imageWidth + x]);
        }
    }
}

//------------------------------------------------------------------------------
void fillAndSmooth(COORDINATE* orderedOffsets, int numberInArray) {
//------------------------------------------------------------------------------
    unsigned short numberOfThreads = getNumberOfAvailableThreads();
    GENERIC_RGB* imageData;
    COORDINATE overlap;
    unsigned long xOverlaps;
    unsigned long yOverlaps;
    unsigned long imageHeight;
    unsigned long imageWidth;
    COORDINATE baseIncrement;
    COORDINATE shiftIncrement;
    IMAGE_LIMITS base;
    IMAGE_LIMITS shift;
    char statusBuffer[100];

    // Calculate the dimensions of the output image
    imageHeight = getFileHeight(INDEX_BASE);
```

```
imageWidth = getFileWidth(INDEX_BASE);
for (short offset = 0; offset < numberInArray; offset++) {
    if (imageWidth < orderedOffsets[offset].x + getFileWidth(offset)) {
        imageWidth = orderedOffsets[offset].x + getFileWidth(offset);
    }
    if (imageHeight < orderedOffsets[offset].y +
        getFileHeight(offset)) {
        imageHeight = orderedOffsets[offset].y + getFileHeight(offset);
    }
}
// Prepare the local image for writing new data
imageData = (GENERIC_RGB*)malloc((imageHeight + 1) * (imageWidth + 1) *
    sizeof(GENERIC_RGB));
if (imageData == NULL) {
    sprintf(statusBuffer, "Insufficient memory to allocate %d bytes for "
        "image contruction.", imageHeight*imageWidth*sizeof(GENERIC_RGB));
    errorAndTerminate(statusBuffer, INSUFFICIENT_MEMORY);
}
// Set the image to all black
memset(imageData, 0, imageHeight * imageWidth * sizeof(GENERIC_RGB));
// Gather the data to be globally available
localImage.imageData = imageData;
localImage.imageHeight = imageHeight;
localImage.imageWidth = imageWidth;
status("Constructing the final image.");
for (int file = 0; file < numberInArray; file++) {
    overlap = orderedOffsets[file];
    if (file > 0) {
        // Determine the number of overlaps in the x axis
        xOverlaps = getFileWidth(file);
        if (overlap.x < orderedOffsets[file - 1].x) {
            xOverlaps = getFileWidth(file) + overlap.x;
        }
        else if (overlap.x + getFileWidth(file) >
            getFileWidth(file - 1) + orderedOffsets[file - 1].x) {
            xOverlaps = getFileWidth(file - 1) - overlap.x;
        }
        // Determine the number of overlaps in the y axis
        yOverlaps = getFileHeight(file);
        if (overlap.y < orderedOffsets[file - 1].y) {
            yOverlaps = getFileHeight(file) + overlap.y;
        }
        else if (overlap.x + getFileHeight(file) >
            getFileHeight(file - 1) + orderedOffsets[file - 1].y) {
            yOverlaps = getFileHeight(file - 1) - overlap.y;
        }
        // Gather the data to be globally available
        localImage.overlap = overlap;
        localImage.xOverlaps = xOverlaps;
        localImage.yOverlaps = yOverlaps;
        localImage.base.x = orderedOffsets[file].x -
            orderedOffsets[file - 1].x;
        localImage.base.y = orderedOffsets[file].y -
            orderedOffsets[file - 1].y;
        // Assign the threads to compile the image
        base.imageIndex = file - 1;
        resetLimit(&base,&baseIncrement);
        shift.imageIndex = file;
        resetLimit(&shift,&shiftIncrement);
    }
    else {
        // Gather the data to be globally available
        localImage.overlap = orderedOffsets[file];
        localImage.xOverlaps = 0;
        localImage.yOverlaps = 0;
        localImage.base.x = 0;
        localImage.base.y = 0;
        // Reset the limits
        base.imageIndex = file;
        resetLimit(&base,&baseIncrement);
        shift = base;
        shiftIncrement = baseIncrement;
    }
    // Add the image to the panorama
    for (short thread = 0; thread < numberOfThreads; thread++) {
        assignThreadFunction(thread, &compile, base, shift, NULL);
        incrementLimit(&base,baseIncrement);
        incrementLimit(&shift,shiftIncrement);
    }
    waitForAllCores();
}
// Resize the last image to transfer the data
setFileSize(numberInArray - 1, imageHeight, imageWidth);
// Assign the threads to transfer the local compiled image
```

```
    shift.imageIndex = numberInArray - 1;
    resetLimit(&shift, &shiftIncrement);
    status("Transferring the local image buffer to output.");
    for (short threads = 0; threads < numberOfThreads; threads++) {
        assignThreadFunction(threads, &transferLocalData, shift, shift, NULL);
        incrementLimit(&shift,shiftIncrement);
    }
    waitForAllCores();
    // Free the dynamic memory
    free(imageData);
}

//--------------------------------------------------------------------------------
void normaliseOffsets(COORDINATE* coordinateArray, unsigned int numberInArray) {
//--------------------------------------------------------------------------------
    COORDINATE minOffset = {0};

    // Find the minimum values for both axis
    for (short offset = 0; offset < numberInArray; offset++) {
        if (minOffset.x > coordinateArray[offset].x) {
            minOffset.x = coordinateArray[offset].x;
        }
        if (minOffset.y > coordinateArray[offset].y) {
            minOffset.y = coordinateArray[offset].y;
        }
    }
    // Add the minimum value to all values to ensure positive offsets
    for (short offset = 0; offset < numberInArray; offset++) {
        coordinateArray[offset].x += abs(minOffset.x);
        coordinateArray[offset].y += abs(minOffset.y);
    }
}

//--------------------------------------------------------------------------------
void alignImages(int stepping, int toleranceSize) {
//--------------------------------------------------------------------------------
    unsigned short numberOfThreads = getNumberOfAvailableThreads();
    MATCH_COORDINATE match[numberOfThreads];
    COORDINATE alignments[getNumberOfFiles()] = {0};
    COORDINATE increment;
    COORDINATE incrementOffset;
    unsigned long height[2];
    unsigned long width[2];
    IMAGE_LIMITS base;
    IMAGE_LIMITS shift;
    short bestMatch = 0;
    char statusBuffer[100];

    // Define the agressiveness of the filters by the size
    tolerance = toleranceSize;
    if (toleranceSize < 3) {
        tolerance = DEFAULT_TOLERANCE;
    }
    else if (toleranceSize > 25) {
        tolerance = 25;
    }
    steppingSize = stepping;
    if (stepping < 2) {
        steppingSize = DEFAULT_STEP_SIZE;
    }
    else if (stepping > 10) {
        steppingSize = 10;
    }
    for (short fileIndex = 1; fileIndex < getNumberOfFiles(); fileIndex++) {
        // Initialise the match coordinate variables
        for (short thread = 0; thread < numberOfThreads; thread++) {
            match[thread].offset.x = 0;
            match[thread].offset.y = 0;
            match[thread].matchQuality = 0;
        }
        // Get the dimensions of the images
        height[INDEX_BASE] = getFileHeight(fileIndex - 1);
        height[INDEX_SHIFT] = getFileHeight(fileIndex);
        width[INDEX_BASE] = getFileWidth(fileIndex - 1);
        width[INDEX_SHIFT] = getFileWidth(fileIndex);
        // Prepare the nonstandard limits
        increment.x = ceil((double)(width[INDEX_BASE] +
            width[INDEX_SHIFT]) / numberOfThreads);
        increment.y = height[INDEX_BASE] + height[INDEX_SHIFT];
        incrementOffset.x = 0 - width[INDEX_SHIFT];
        incrementOffset.y = 0 - height[INDEX_SHIFT];
        resetLimitPreCalculated(&shift, increment, incrementOffset);
        increment.y = 0;
        base = shift;
```

```
        // Split up the workload by coordinates
        base.imageIndex = fileIndex - 1;
        shift.imageIndex = fileIndex;
        // Assign the threads to seek appropriate overlap coordinates
        status("Seeking the alignment poisiton.");
        for (short thread = 0; thread < numberOfThreads; thread++) {
            assignThreadFunction(thread, &seekOffset, base, shift,
                (char*)&match[thread]);
            incrementLimit(&base, increment);
            incrementLimit(&shift, increment);
        }
        // Wait until all threads complete and update the status
        waitForAllCores();
        sprintf(statusBuffer, "Matching analysis complete for images %d "
            "and %d.", fileIndex, fileIndex + 1);
        status(statusBuffer);
        // Select the best match quality
        for (short thread = 0; thread < numberOfThreads; thread++) {
            if (match[bestMatch].matchQuality < match[thread].matchQuality) {
                bestMatch = thread;
            }
        }
        // Add the best match position to the image alignment array
        alignments[fileIndex].x = match[bestMatch].offset.x +
            alignments[fileIndex - 1].x;
        alignments[fileIndex].y = match[bestMatch].offset.y +
            alignments[fileIndex - 1].y;
        normaliseOffsets(alignments, getNumberOfFiles());
        // Update the status
        sprintf(statusBuffer, "Image selected for joining at (%d,%d).",
            alignments[fileIndex].x, alignments[fileIndex].y);
        status(statusBuffer);
    }
    // Join all of the images
    fillAndSmooth(alignments, getNumberOfFiles());
    status("All images joined.");
}
```

B4 AXIS.H

```
//------------------------------------------------------------------------------
// axis.h - header file for axis, coordinate and geometric based activities
//------------------------------------------------------------------------------
#ifndef AXIS_H
#define AXIS_H
//------------------------------------------------------------------------------

//------------------------------------------------------------------------------
// Structures
//------------------------------------------------------------------------------

typedef struct {
    long x;
    long y;
} COORDINATE;

typedef struct {
    COORDINATE upperLeft;
    COORDINATE lowerRight;
} RECTANGLE;

typedef struct {
    COORDINATE min;
    COORDINATE max;
} LIMITS;

typedef struct {
    short imageIndex;
    LIMITS limits;
} IMAGE_LIMITS;

//------------------------------------------------------------------------------
// Functions
//------------------------------------------------------------------------------

COORDINATE calculateIncrement(short imageIndex);
// Description:    Divides the given image into an incremental range for the
//                 number of threads
// Inputs:         Index to the image file
// Returns:        The incremenation structure with the values inserted
//--------------------------------------------------------------------------------

void resetLimit(IMAGE_LIMITS* image, COORDINATE* incrementor);
// Description:    Resets variables to start processing again
// Inputs:         A pointer to the image limits structure,
//                 a pointer for the coordinates of the incrementor
// Returns:        None
//--------------------------------------------------------------------------------

void resetLimitPreCalculated(IMAGE_LIMITS* image,
                             const COORDINATE incrementor,
                             const COORDINATE offset);
// Description:    Resets variables to start processing again using the
//                 precalculated incrementor and offset
// Inputs:         A pointer to the image limits structure,
//                 a pointer for the coordinates of the incrementor,
//                 a coordinate offset amount to start the limit with
// Returns:        None
//--------------------------------------------------------------------------------

void incrementLimit(IMAGE_LIMITS* image, COORDINATE incrementor);
// Description:    Adds a fixed incremented value to the image limits structure
// Inputs:         A pointer to the image limits structure,
//                 the increment structure containing the values to add
// Returns:        None

#endif
```

B5 AXIS.C

```
//--------------------------------------------------------------------------------
// axis.c - implementation for axis, coordinate and geometric based activities
//--------------------------------------------------------------------------------

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "axis.h"
#include "fileio.h"
```

```
#include "threads.h"

//-----------------------------------------------------------------------------
COORDINATE calculateIncrement(short imageIndex) {
//-----------------------------------------------------------------------------
    COORDINATE increment;

    increment.x = ceil((double)getFileWidth(imageIndex) /
        getNumberOfAvailableThreads());
    increment.y = getFileHeight(imageIndex);
    return increment;
}

//-----------------------------------------------------------------------------
void resetLimit(IMAGE_LIMITS* image, COORDINATE* incrementor) {
//-----------------------------------------------------------------------------
    *incrementor = calculateIncrement(image->imageIndex);
    image->limits.min.x = 0;
    image->limits.min.y = 0;
    image->limits.max = *incrementor;
    // Assure that the increments are not total dimension of the image
    if (incrementor->x == getFileWidth(image->imageIndex)) {
        incrementor->x = 0;
    }
    if (incrementor->y == getFileHeight(image->imageIndex)) {
        incrementor->y = 0;
    }
}

//-----------------------------------------------------------------------------
void resetLimitPreCalculated(IMAGE_LIMITS* image,
                             const COORDINATE incrementor,
                             const COORDINATE offset) {
//-----------------------------------------------------------------------------
    image->limits.min.x = offset.x;
    image->limits.min.y = offset.y;
    image->limits.max.x = offset.x + incrementor.x;
    image->limits.max.y = offset.y + incrementor.y;
}

//-----------------------------------------------------------------------------
void incrementLimit(IMAGE_LIMITS* image, COORDINATE incrementor) {
```

```
//------------------------------------------------------------------------------
    image->limits.min.x += incrementor.x;
    image->limits.max.x += incrementor.x;
    image->limits.min.y += incrementor.y;
    image->limits.max.y += incrementor.y;
    // Ensure that the maximum size of image is not exceeded
    if (image->limits.max.x > getFileWidth(image->imageIndex)) {
        image->limits.max.x = getFileWidth(image->imageIndex);
    }
    if (image->limits.max.y > getFileHeight(image->imageIndex)) {
        image->limits.max.y = getFileHeight(image->imageIndex);
    }
}
```

B6 BMP.H

```
//------------------------------------------------------------------------------
// bmp.h - header file for reading and writing bitmap files
//------------------------------------------------------------------------------
#ifndef BMP_H
#define BMP_H
//------------------------------------------------------------------------------

//------------------------------------------------------------------------------
// Structures
//------------------------------------------------------------------------------

typedef struct {
    unsigned int fileSize;
    short reserved, reserved2;
    unsigned int offsetToData;
} BITMAP_FILE_HEADER;

typedef struct {
    unsigned int headerSize;
    unsigned int width;
    unsigned int height;
    unsigned short numberOfPlanes;
    unsigned short bitsPerPixel;
    unsigned int compressionType;
    unsigned int imageSize;
    unsigned int xPixelsPerMeter;
    unsigned int yPixelsPerMeter;
    unsigned int numberOfColours;
    unsigned int numberOfImportantColours;
} BITMAP_INFO_HEADER;

typedef struct {
    unsigned char blue;
    unsigned char green;
    unsigned char red;
} BITMAP_RGB;

typedef struct {
    BITMAP_FILE_HEADER fileHeader;
    BITMAP_INFO_HEADER infoHeader;
    BITMAP_RGB* imageData;
} BITMAP;

//------------------------------------------------------------------------------
// Functions
//------------------------------------------------------------------------------

BITMAP readBitmap(FILE* imageFile);
// Description:    Reads the BITMAP structure from disk
// Inputs:         A FILE pointer to a previously opened file
// Returns:        The bitmap structure containing the file data
//------------------------------------------------------------------------------

void writeBitmap(FILE* imageFile, BITMAP bitmap);
// Description:    Writes the BITMAP structure to disk
// Inputs:         An open FILE pointer,
//                 a BITMAP image to write to disk
// Returns:        None
//------------------------------------------------------------------------------

BITMAP_RGB getBitmapPixel(BITMAP* bitmap, int x, int y);
// Description:    Returns the pixel at the point x,y
// Inputs:         A pointer to the image data to be read,
//                 coordinates of a point
// Returns:        The RGB structure correlating to the coordinate
//------------------------------------------------------------------------------
```

```
void setBitmapPixel(BITMAP* bitmap, int x, int y, BITMAP_RGB value);
// Description:    Sets the pixel at the point x,y
// Inputs:         A pointer to the image data to be read,
//                 coordinates of a point,
//                 the value to save
// Returns:        None
//--------------------------------------------------------------------------------

unsigned int getBitmapWidth(BITMAP* bitmap);
// Description:    Returns the width of the bitmap
// Inputs:         A pointer to the image data to be assessed
// Returns:        Width of the image
//--------------------------------------------------------------------------------

unsigned int getBitmapHeight(BITMAP* bitmap);
// Description:    Returns the height of the bitmap
// Inputs:         A pointer to the image data to be assessed
// Returns:        Height of the image
//--------------------------------------------------------------------------------

void setBitmapSize(BITMAP* bitmap, unsigned long width, unsigned long height);
// Description:    Sets the size of the bitmap
// Inputs:         A pointer to the image data to be read,
//                 the new width of the image,
//                 the new height of the image
// Returns:        None

#endif
```

B7 BMP.C

```
//--------------------------------------------------------------------------------
// bmp.c - implementation file for reading and writing bitmap files
//--------------------------------------------------------------------------------

#include <stdio.h>
#include <stdlib.h>
#include "bmp.h"
#include "boolean.h"
```

```
#include "error.h"

//--------------------------------------------------------------------------------
BITMAP readBitmap(FILE* imageFile) {
//--------------------------------------------------------------------------------
    BITMAP bitmap;
    char statusBuffer[100];

    // Read the file headers
    if (fread((char *)&bitmap.fileHeader, sizeof(BITMAP_FILE_HEADER), 1,
            imageFile) == 0) {
        errorAndTerminate("Unable to read image file header.", IMAGE_IO_ERROR);
    }
    if (fread((char *)&bitmap.infoHeader, sizeof(BITMAP_INFO_HEADER), 1,
            imageFile) == 0) {
        errorAndTerminate("Unable to read image info header.", IMAGE_IO_ERROR);
    }
    status("Acquired the bitmap headers.");
    // Check the compression status
    if (bitmap.infoHeader.compressionType != 0) {
        errorAndTerminate("Bitmap compression unsupported.", UNSUPPORTED_TYPE);
    }
    // Prepare to read the bitmap image data
    fseek(imageFile, bitmap.fileHeader.offsetToData, SEEK_SET);
    bitmap.imageData = (BITMAP_RGB*)malloc(bitmap.infoHeader.imageSize);
    if (bitmap.imageData == NULL) {
        errorAndTerminate("Insufficient memory to allocate for bitmap image "
            "data.", INSUFFICIENT_MEMORY);
    }
    // Notify of status and read bitmap image data
    sprintf(statusBuffer, "Beginning read of %d bytes.",
        bitmap.infoHeader.imageSize);
    status(statusBuffer);
    if (fread(bitmap.imageData, bitmap.infoHeader.imageSize, 1,
            imageFile) == 0) {
        errorAndTerminate("Unable to read image data.", IMAGE_IO_ERROR);
    }
    status("Successfully buffered image data.");
    return bitmap;
}

//--------------------------------------------------------------------------------
```

```
void writeBitmap(FILE* imageFile, BITMAP bitmap) {
//-------------------------------------------------------------------------------
    char statusBuffer[100];

    // Write the file headers
    if (fwrite("BM", sizeof(char)*2, 1, imageFile) == 0) {
        errorAndTerminate("Image signature not written.", IMAGE_IO_ERROR);
    }
    if (fwrite((char *)&bitmap.fileHeader, sizeof(BITMAP_FILE_HEADER), 1,
            imageFile) == 0) {
        errorAndTerminate("Image file header not written.", IMAGE_IO_ERROR);
    }
    if (fwrite((char *)&bitmap.infoHeader, sizeof(BITMAP_INFO_HEADER), 1,
            imageFile) == 0) {
        errorAndTerminate("Image info header not written.", IMAGE_IO_ERROR);
    }
    status("Written image headers.");
    // Write the bitmap image data
    fseek(imageFile, bitmap.fileHeader.offsetToData, SEEK_SET);
    if (fwrite(bitmap.imageData, bitmap.infoHeader.imageSize, 1,
            imageFile) == 0) {
        errorAndTerminate("Image data not written.", IMAGE_IO_ERROR);
    }
    status("Written image data.");
}

//-------------------------------------------------------------------------------
BITMAP_RGB getBitmapPixel(BITMAP* bitmap, int x, int y) {
//-------------------------------------------------------------------------------
    return bitmap->imageData[(bitmap->infoHeader.height - y - 1) *
            bitmap->infoHeader.width + x];
}

//-------------------------------------------------------------------------------
void setBitmapPixel(BITMAP* bitmap, int x, int y, BITMAP_RGB value) {
//-------------------------------------------------------------------------------
    bitmap->imageData[(bitmap->infoHeader.height - y - 1) *
            bitmap->infoHeader.width + x] = value;
}

//-------------------------------------------------------------------------------
unsigned int getBitmapWidth(BITMAP* bitmap) {
```

```
//-----------------------------------------------------------------------------
    return bitmap->infoHeader.width;
}

//-----------------------------------------------------------------------------
unsigned int getBitmapHeight(BITMAP* bitmap) {
//-----------------------------------------------------------------------------
    return bitmap->infoHeader.height;
}

//-----------------------------------------------------------------------------
void setBitmapSize(BITMAP* bitmap, unsigned long width, unsigned long height) {
//-----------------------------------------------------------------------------
    bitmap->infoHeader.width = width;
    bitmap->infoHeader.height = height;
    // Readjust the image size in memory
    bitmap->imageData = (BITMAP_RGB*)realloc(bitmap->imageData,
        width * height * sizeof(BITMAP_RGB));
    if (bitmap->imageData == NULL) {
        errorAndTerminate("Insufficient memory to reallocate for bitmap image "
            "data.", INSUFFICIENT_MEMORY);
    }
}
```

B8 BOOLEAN.H

```
//-----------------------------------------------------------------------------
// boolean.h - header file for boolean definition type
//-----------------------------------------------------------------------------
#ifndef BOOLEAN_H
#define BOOLEAN_H
//-----------------------------------------------------------------------------

typedef enum {FALSE = 0, TRUE = 1} BOOLEAN_TYPE;

#endif
```

B9 ERROR.H

```
//------------------------------------------------------------------------------
// error.h - header file for errors encountered
//------------------------------------------------------------------------------
#ifndef ERROR_H
#define ERROR_H
//------------------------------------------------------------------------------

#define UNKNOWN_ERROR 1
#define THREAD_ERROR 2
#define SEMAPHORE_ERROR 3
#define IMAGE_IO_ERROR 4
#define UNSUPPORTED_TYPE 5
#define INSUFFICIENT_MEMORY 6

void warning(char message[]);
// Description: Outputs the passed message to the terminal
// Inputs:      The string error message
// Returns:     None
//------------------------------------------------------------------------------

void errorAndTerminate(char message[], int failStatus);
// Description: Outputs the passed error to the terminal and terminates the
//              program
// Inputs:      The string error message,
//              the status relating to the perceived issue
// Returns:     None
//------------------------------------------------------------------------------

void status(char message[]);
// Description: Outputs the passed message if verbose output wanted
// Inputs:      The string text of the current status
// Returns:     None
//------------------------------------------------------------------------------

void setVerboseMode(int verboseOnOff);
// Description: Sets the verbose mode toggle
// Inputs:      A boolean representing whether the verbose output is shown
// Returns:     None
```

```
#endif
```

B10 ERROR.C

```
//-------------------------------------------------------------------------------
// error.c - implementation file for errors encountered
//-------------------------------------------------------------------------------

#include <stdio.h>
#include <stdlib.h>
#include <windows.h>
#include "error.h"

//-------------------------------------------------------------------------------
// Global Variables
//-------------------------------------------------------------------------------

static int verboseMode;
static HANDLE hOutputSemaphore;

//-------------------------------------------------------------------------------
void warning(char message[]) {
//-------------------------------------------------------------------------------
    // If output semaphore not previously used, attempt creation
    if (hOutputSemaphore == NULL) {
        hOutputSemaphore = CreateSemaphore(NULL, 1, 1, "output");
        if (hOutputSemaphore == NULL) {
            errorAndTerminate("Output semaphore creation failed.",
                SEMAPHORE_ERROR);
        }
    }
    // Wait for previous output to finish, then proceed to output
    if (WaitForSingleObject(hOutputSemaphore, INFINITE) == WAIT_OBJECT_0) {
        printf("%s\n", message);
        fflush(stdout);
        ReleaseSemaphore(hOutputSemaphore, 1, NULL);
    }
    else {
        errorAndTerminate("Unable to obtain semaphore access to task thread.",
            SEMAPHORE_ERROR);
    }
}

//------------------------------------------------------------------------------
void errorAndTerminate(char message[], int failStatus) {
//------------------------------------------------------------------------------
    char statusBuffer[512];

    sprintf(statusBuffer, "Error: %.511s", message);
    warning(statusBuffer);
    exit(failStatus);
}

//------------------------------------------------------------------------------
void status(char message[]) {
//------------------------------------------------------------------------------
    if (verboseMode) {
        warning(message);
    }
}

//------------------------------------------------------------------------------
void setVerboseMode(int verboseOnOff) {
//------------------------------------------------------------------------------
    verboseMode = verboseOnOff;
}
```

B11 FILEIO.H

```
//------------------------------------------------------------------------------
// fileio.h - header file for reading and writing image files
//------------------------------------------------------------------------------
#ifndef FILEIO_H
#define FILEIO_H
//------------------------------------------------------------------------------

#include "boolean.h"

//------------------------------------------------------------------------------
// Structures
```

```
//----------------------------------------------------------------------------

typedef struct {
    unsigned char red;
    unsigned char green;
    unsigned char blue;
    unsigned char reserved;
} GENERIC_RGB;

//----------------------------------------------------------------------------
// Functions
//----------------------------------------------------------------------------

BOOLEAN_TYPE setFilePath(char pathToFile[]);
// Description:    Assigns the path to a vacant position in the image input array
// Inputs:         A path to a valid file
// Returns:        Whether the operation completed successfully
//----------------------------------------------------------------------------

BOOLEAN_TYPE setFileOutput(char pathToFile[]);
// Description:    Outputs the passed message to the terminal
// Inputs:         The string error message
// Returns:        Whether the operation completed successfully
//----------------------------------------------------------------------------

void readImageFile(int indexOfFile);
// Description:    Reads the selected file from disk
// Inputs:         The index of the file in the image input array
// Returns:        None
//----------------------------------------------------------------------------

BOOLEAN_TYPE readAllImageFiles(void);
// Description:    Reads all of the files from disk
// Inputs:         None
// Returns:        Whether the operation completed successfully
//----------------------------------------------------------------------------

void writeImageFile(void);
// Description:    Write the output file to disk
// Inputs:         None
// Returns:        None
//----------------------------------------------------------------------------
```

```
GENERIC_RGB getPixel(int indexOfFile, int x, int y);
// Description:    Returns the pixel at the coordinate x,y
// Inputs:         The index of the image,
//                 the x,y coordinate of the point
// Returns:        The structure relating to the colour at the coordinate
//--------------------------------------------------------------------------------

void setPixel(int indexOfFile, int x, int y, GENERIC_RGB value);
// Description:    Sets the pixel at the coordinate x,y
// Inputs:         The index of the image, coordinates of the point and the value
//                 to save
// Returns:        None
//--------------------------------------------------------------------------------

unsigned int getNumberOfFiles(void);
// Description:    Returns the number of files presently read in
// Inputs:         None
// Returns:        Number of files
//--------------------------------------------------------------------------------

unsigned long getFileWidth(int indexOfFile);
// Description:    Returns the width of file
// Inputs:         The index of the image
// Returns:        Given file width
//--------------------------------------------------------------------------------

unsigned long getFileHeight(int indexOfFile);
// Description:    Returns the height of file
// Inputs:         The index of the image
// Returns:        Given file height
//--------------------------------------------------------------------------------

void setFileSize(int indexOfFile, unsigned long height, unsigned long width);
// Description:    Sets the width and height properties of the selected image
// Inputs:         The index of the image,
//                 the new file height,
//                 the new file width
// Returns:        None

#endif
```

B12 FILEIO.C

```
//----------------------------------------------------------------------------
// fileio.c - implementation file for reading and writing image files
//----------------------------------------------------------------------------

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "fileio.h"
#include "bmp.h"
#include "boolean.h"
#include "error.h"
#include "tiff.h"

#define MAX_FILES 50

//----------------------------------------------------------------------------
// Structures
//----------------------------------------------------------------------------

enum FILE_TYPE {NO_FORMAT, BITMAP_FORMAT, TIFF_FORMAT};

typedef struct {
    char* path;
    enum FILE_TYPE fileType;
    BITMAP bitmap;
    TIFF tiff;
} IMAGE_FILE;

//----------------------------------------------------------------------------
// Global Variables
//----------------------------------------------------------------------------

static IMAGE_FILE imageFiles[MAX_FILES + 1];
static char* outputImagePath;
static unsigned short numberOfFiles;
static char* numberOfFilesPtr;

//----------------------------------------------------------------------------
```

```
BOOLEAN_TYPE setFilePath(char pathToFile[]) {
//-------------------------------------------------------------------------------
    if (numberOfFilesPtr == NULL) {
        numberOfFilesPtr = (char*)&numberOfFiles;
        numberOfFiles = 0;
    }
    // Ensure that the passed path is not NULL
    if (pathToFile == NULL) {
        return FALSE;
    }
    // Allocate the appropriate amount of memory and store the path
    imageFiles[numberOfFiles].path = (char*)malloc(sizeof(char) *
        (strlen(pathToFile) + 1));
    if (imageFiles[numberOfFiles].path == NULL) {
        errorAndTerminate("Insufficient memory to allocate for file path.",
            INSUFFICIENT_MEMORY);
    }
    strcpy(imageFiles[numberOfFiles].path, pathToFile);
    numberOfFiles++;
    return TRUE;
}

//-------------------------------------------------------------------------------
BOOLEAN_TYPE setFileOutput(char pathToFile[]) {
//-------------------------------------------------------------------------------
    // Ensure that the passed path is not NULL
    if (pathToFile == NULL) {
        outputImagePath = NULL;
        return FALSE;
    }
    // Allocate the appropriate amount of memory and store the path
    outputImagePath = (char*)malloc(sizeof(char) * (strlen(pathToFile) + 1));
    if (outputImagePath == NULL) {
        errorAndTerminate("Insufficient memory to allocate for file path.",
            INSUFFICIENT_MEMORY);
    }
    strcpy(outputImagePath, pathToFile);
    return TRUE;
}

//-------------------------------------------------------------------------------
void readImageFile(int indexOfFile) {
```

```
//-------------------------------------------------------------------------------
    FILE* imageFile;
    unsigned char fileSignature[] = "\0\0\0\0\0";
    char statusBuffer[300];

    // Open the file for reading in binary mode
    imageFile = fopen(imageFiles[indexOfFile].path, "rb");
    if (!imageFile) {
        sprintf(statusBuffer, "Image file \"%.256s\" not available for "
            "reading.", imageFiles[indexOfFile].path);
        errorAndTerminate(statusBuffer, IMAGE_IO_ERROR);
    }
    sprintf(statusBuffer, "Attempting to read the file \"%.256s\".",
        imageFiles[indexOfFile].path);
    status(statusBuffer);
    // Read the file header and assign the appropriate loader for the image
    imageFiles[indexOfFile].fileType = NO_FORMAT;
    fread((char *)&fileSignature, sizeof(char), 2, imageFile);
    if (strcmp(fileSignature, "BM") == 0){
        imageFiles[indexOfFile].fileType = BITMAP_FORMAT;
        imageFiles[indexOfFile].bitmap = readBitmap(imageFile);
    }
    else if (strcmp(fileSignature, "MM") == 0){
        // Double check that the next two magic numbers match
        fread((char *)&fileSignature[2], sizeof(char), 2, imageFile);
        if (strcmp(fileSignature, "MM\0*") != 0) {
            errorAndTerminate("Unsupported file type.", UNSUPPORTED_TYPE);
        }
        imageFiles[indexOfFile].fileType = TIFF_FORMAT;
        imageFiles[indexOfFile].tiff = readTIFF(imageFile, TRUE);
    }
    else if (strcmp(fileSignature, "II") == 0) {
        // Double check that the next two magic numbers match
        fread((char *)&fileSignature[2], sizeof(char), 2, imageFile);
        if (strcmp(fileSignature, "II*\0") != 0) {
            errorAndTerminate("Unsupported file type.", UNSUPPORTED_TYPE);
        }
        imageFiles[indexOfFile].fileType = TIFF_FORMAT;
        imageFiles[indexOfFile].tiff = readTIFF(imageFile, FALSE);
    }
    else {
        errorAndTerminate("Unsupported file type.", UNSUPPORTED_TYPE);
    }
    sprintf(statusBuffer, "Reading of file \"%.256s\" done.",
        imageFiles[indexOfFile].path);
    status(statusBuffer);
    // Close the open file
    fclose(imageFile);
}

//-------------------------------------------------------------------------------
BOOLEAN_TYPE readAllImageFiles(void) {
//-------------------------------------------------------------------------------
    for (int fileNumber = 0; fileNumber < numberOfFiles; fileNumber++) {
        readImageFile(fileNumber);
    }
    return TRUE;
}

//-------------------------------------------------------------------------------
void writeImageFile(void) {
//-------------------------------------------------------------------------------
    FILE* imageFile;
    unsigned char fileSignature[] = "\0\0\0\0\0";
    char statusBuffer[300];

    // Open the file for writing in binary mode
    imageFile = fopen(outputImagePath, "wb");
    if (!imageFile) {
        sprintf(statusBuffer, "Image file \"%.256s\" not available for "
            "writing.", outputImagePath);
        errorAndTerminate(statusBuffer, IMAGE_IO_ERROR);
    }
    sprintf(statusBuffer, "Attempting to write the file \"%.256s\".",
        outputImagePath);
    status(statusBuffer);
    // Determine the appropriate saver for the image
    if (imageFiles[numberOfFiles - 1].fileType == BITMAP_FORMAT) {
        writeBitmap(imageFile, imageFiles[numberOfFiles - 1].bitmap);
    }
    else if (imageFiles[numberOfFiles - 1].fileType == TIFF_FORMAT) {
        writeTIFF(imageFile, imageFiles[numberOfFiles - 1].tiff);
    }
    // Close the open file
    fclose(imageFile);
}

//--------------------------------------------------------------------------------
GENERIC_RGB getPixel(int indexOfFile, int x, int y) {
//--------------------------------------------------------------------------------
    GENERIC_RGB colours;
    BITMAP_RGB bitmapRGB;
    TIFF_RGB tiffRGB;

    if (imageFiles[indexOfFile].fileType == BITMAP_FORMAT) {
        bitmapRGB = getBitmapPixel(&imageFiles[indexOfFile].bitmap, x, y);
        // Setup RGB colour parameters
        colours.red = bitmapRGB.red;
        colours.green = bitmapRGB.green;
        colours.blue = bitmapRGB.blue;
    }
    else if (imageFiles[indexOfFile].fileType == TIFF_FORMAT) {
        tiffRGB = getTIFFPixel(&imageFiles[indexOfFile].tiff, x, y);
        // Setup RGB colour parameters
        colours.red = tiffRGB.red;
        colours.green = tiffRGB.green;
        colours.blue = tiffRGB.blue;
    }
    else {
        // Setup RGB colour parameters
        colours.red = 0;
        colours.green = 0;
        colours.blue = 0;
    }
    return colours;
}

//--------------------------------------------------------------------------------
void setPixel(int indexOfFile, int x, int y, GENERIC_RGB value) {
//--------------------------------------------------------------------------------
    BITMAP_RGB bitmapRGB;
    TIFF_RGB tiffRGB;

    if (imageFiles[indexOfFile].fileType == BITMAP_FORMAT) {
        // Setup RGB colour parameters
        bitmapRGB.red = value.red;
        bitmapRGB.green = value.green;
        bitmapRGB.blue = value.blue;
        // Set the bitmap colours
        setBitmapPixel(&imageFiles[indexOfFile].bitmap, x, y, bitmapRGB);
    }
    else if (imageFiles[indexOfFile].fileType == TIFF_FORMAT) {
        // Setup RGB colour parameters
        tiffRGB.red = value.red;
        tiffRGB.green = value.green;
        tiffRGB.blue = value.blue;
        // Set the TIFF colours
        setTIFFPixel(&imageFiles[indexOfFile].tiff, x, y, tiffRGB);
    }
}

//--------------------------------------------------------------------------------
unsigned int getNumberOfFiles(void) {
//--------------------------------------------------------------------------------
    return numberOfFiles;
}

//--------------------------------------------------------------------------------
unsigned long getFileWidth(int indexOfFile) {
//--------------------------------------------------------------------------------
    // Check that index in range
    if (indexOfFile < 0) {
        indexOfFile = 0;
    }
    else if (indexOfFile > numberOfFiles) {
        indexOfFile = numberOfFiles - 1;
    }
    // Return the width
    if (imageFiles[indexOfFile].fileType == BITMAP_FORMAT) {
        return (unsigned long)getBitmapWidth(&imageFiles[indexOfFile].bitmap);
    }
    else if (imageFiles[indexOfFile].fileType == TIFF_FORMAT) {
        return (unsigned long)getTIFFWidth(&imageFiles[indexOfFile].tiff);
    }
    return 0;
}

//--------------------------------------------------------------------------------
```

```
unsigned long getFileHeight(int indexOfFile) {
//------------------------------------------------------------------------------
    // Check that index in range
    if (indexOfFile < 0) {
        indexOfFile = 0;
    }
    else if (indexOfFile > numberOfFiles) {
        indexOfFile = numberOfFiles - 1;
    }
    // Return the height
    if (imageFiles[indexOfFile].fileType == BITMAP_FORMAT) {
        return (unsigned long)getBitmapHeight(&imageFiles[indexOfFile].bitmap);
    }
    else if (imageFiles[indexOfFile].fileType == TIFF_FORMAT) {
        return (unsigned long)getTIFFLength(&imageFiles[indexOfFile].tiff);
    }
    return 0;
}

//------------------------------------------------------------------------------
void setFileSize(int indexOfFile, unsigned long height, unsigned long width) {
//------------------------------------------------------------------------------
    if (imageFiles[indexOfFile].fileType == BITMAP_FORMAT) {
        setBitmapSize(&imageFiles[indexOfFile].bitmap, width, height);
    }
    else if (imageFiles[indexOfFile].fileType == TIFF_FORMAT) {
        setTIFFSize(&imageFiles[indexOfFile].tiff, width, height);
    }
}
```

B13 NOISEREDUCTION.H

```
//------------------------------------------------------------------------------
// noisereduction.h - header file for reducing noise in image files
//------------------------------------------------------------------------------
#ifndef NOISEREDUCTION_H
#define NOISEREDUCTION_H
//------------------------------------------------------------------------------

void reduceNoise(int sizeOfFilter);
```

```
// Description:    Filters out noise in all of the input images
// Inputs:         Filter size relating to the aggressiveness of filtering to use
// Returns:        None

#endif
```

B14 NOISEREDUCTION.C

```
//--------------------------------------------------------------------------------
// noisereduction.c - implementation file for reducing noise in image files
//--------------------------------------------------------------------------------

#include <stdio.h>
#include <stdlib.h>
#include "noisereduction.h"
#include "axis.h"
#include "error.h"
#include "fileio.h"
#include "threads.h"

#define DEFAULT_FILTER_SIZE 3

//--------------------------------------------------------------------------------
// Global Variables
//--------------------------------------------------------------------------------

unsigned int filterSize;

//--------------------------------------------------------------------------------
int channelIsLessThan(const void* channelA, const void* channelB) {
//--------------------------------------------------------------------------------
    return (*(unsigned char*)channelA - *(unsigned char*)channelB);
}

//--------------------------------------------------------------------------------
void medianFilter(IMAGE_LIMITS noisyImage,
                  IMAGE_LIMITS reserve,
                  char* reserved) {
//--------------------------------------------------------------------------------
    COORDINATE min;
    COORDINATE max;
    unsigned long arrayOffset;
    GENERIC_RGB pixelAtPoint;
    unsigned long filterSizeSquared = filterSize * filterSize;
    unsigned int center = filterSize / 2;
    unsigned char viewportRedChannel[filterSizeSquared];
    unsigned char viewportGreenChannel[filterSizeSquared];
    unsigned char viewportBlueChannel[filterSizeSquared];

    // Establish the coordinate limits
    min.x = noisyImage.limits.min.x == 0 ? center : noisyImage.limits.min.x;
    min.y = noisyImage.limits.min.y == 0 ? center : noisyImage.limits.min.y;
    max.x = noisyImage.limits.max.x == getFileWidth(noisyImage.imageIndex) ?
        noisyImage.limits.max.x - center : noisyImage.limits.max.x;
    max.y = noisyImage.limits.max.y == getFileHeight(noisyImage.imageIndex) ?
        noisyImage.limits.max.y - center : noisyImage.limits.max.y;
    // Reduce the noise for every pixel excluding boundaries
    for (unsigned long x = min.x; x < max.x; x++) {
        for (unsigned long y = min.y; y < max.y; y++) {
            // Gather the viewport values
            arrayOffset = 0;
            for (long viewportY = 0; viewportY < filterSize; viewportY++) {
                for (long viewportX = 0; viewportX < filterSize; viewportX++) {
                    //arrayOffset = filterSize * viewportY + viewportX;
                    pixelAtPoint = getPixel(noisyImage.imageIndex,
                        x + viewportX - center, y + viewportY - center);
                    viewportRedChannel[arrayOffset] = pixelAtPoint.red;
                    viewportGreenChannel[arrayOffset] = pixelAtPoint.green;
                    viewportBlueChannel[arrayOffset] = pixelAtPoint.blue;
                    arrayOffset++;
                }
            }
            // Sort the cached values
            qsort(viewportRedChannel, filterSizeSquared,
                sizeof(char), channelIsLessThan);
            qsort(viewportGreenChannel, filterSizeSquared,
                sizeof(char), channelIsLessThan);
            qsort(viewportBlueChannel, filterSizeSquared,
                sizeof(char), channelIsLessThan);
            // Save the appropriate value
            arrayOffset = filterSize * center + center;
            pixelAtPoint.red = viewportRedChannel[arrayOffset];
```

```
            pixelAtPoint.green = viewportGreenChannel[arrayOffset];
            pixelAtPoint.blue = viewportBlueChannel[arrayOffset];
            setPixel(noisyImage.imageIndex, x, y, pixelAtPoint);
        }
    }
}

//--------------------------------------------------------------------------------
void reduceNoise(int sizeOfFilter) {
//--------------------------------------------------------------------------------
    COORDINATE baseIncrement;
    IMAGE_LIMITS base;
    char statusBuffer[100];

    // Define the agressiveness of the filter by the size
    filterSize = sizeOfFilter;
    if (sizeOfFilter < 3) {
        filterSize = DEFAULT_FILTER_SIZE;
    }
    else if (sizeOfFilter > 15) {
        filterSize = 15;
    }
    for (short fileIndex = 0; fileIndex < getNumberOfFiles(); fileIndex++) {
        // Split up the workload by coordinates
        base.imageIndex = fileIndex;
        resetLimit(&base,&baseIncrement);
        // Assign the threads to reduce the noise by median filter
        for (short thread = 0; thread < getNumberOfAvailableThreads();
                thread++) {
            assignThreadFunction(thread, &medianFilter, base, base, NULL);
            incrementLimit(&base,baseIncrement);
        }
        // Wait until all threads complete and update the status
        waitForAllCores();
        sprintf(statusBuffer, "Finished noise reduction on image %d.",
            fileIndex);
        status(statusBuffer);
    }
}
```

B15 THREADS.H

```
//------------------------------------------------------------------------------
// threads.h - header file for creation and modification of treads
//------------------------------------------------------------------------------
#ifndef THREADS_H
#define THREADS_H
//------------------------------------------------------------------------------

#include "axis.h"

void createMultipleThreads(int numberOfThreads, int processorOffset);
// Description:    Forms the number of threads corresponding to the number of CPU
//                 cores to utilize and appropriate
// Inputs:         Number of cores to use
// Returns:        None
//------------------------------------------------------------------------------

short getNumberOfAvailableThreads(void);
// Description:    Returns the number of available threads on each of the CPUs
// Inputs:         None
// Returns:        Number of threads
//------------------------------------------------------------------------------

void assignThreadFunction(int core,
                           void (*function)(IMAGE_LIMITS,IMAGE_LIMITS,char*),
                           IMAGE_LIMITS image1,
                           IMAGE_LIMITS image2,
                           char* output);
// Description:    Assigns the function to a thread to execute
// Inputs:         The pointer to the function to execute,
//                 the bounds of the first image structure to operate on,
//                 the bounds of the second image structure to operate on,
//                 the reference to where return values are to be stored
// Returns:        Status of whether the method succeeded
//------------------------------------------------------------------------------

void waitForAllCores(void);
// Description:    Waits until all created threads are all freed from tasks
// Inputs:         None
// Returns:        None
```

```
#endif
```

B16 THREADS.C

```
//------------------------------------------------------------------------------
// threads.c - implementation file for creation and modification of threads
//------------------------------------------------------------------------------

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <windows.h>
#include "threads.h"
#include "axis.h"
#include "error.h"

#define MAX_THREADS 32

//------------------------------------------------------------------------------
// Structures
//------------------------------------------------------------------------------

typedef struct {
    HANDLE hThread;
    HANDLE hIsProcessing;
    void (*functionPtr)(IMAGE_LIMITS,IMAGE_LIMITS,char*);
    IMAGE_LIMITS image1;
    IMAGE_LIMITS image2;
    char* output;
} THREAD_PROPERTIES;

//------------------------------------------------------------------------------
// Global Variables
//------------------------------------------------------------------------------

static THREAD_PROPERTIES threadTable[MAX_THREADS];
static HANDLE hFreeThreadSemaphore;
static int numberOfAvailableThreads;
```

```
//------------------------------------------------------------------------------
void processThreadTasks(int coreNumber) {
//------------------------------------------------------------------------------
    while (TRUE) {
        if (threadTable[coreNumber].functionPtr != NULL) {
            // Begin processing
            threadTable[coreNumber].functionPtr(
                threadTable[coreNumber].image1,
                threadTable[coreNumber].image2,
                threadTable[coreNumber].output);
            // Cleanup after function facilities ready for the next task
            threadTable[coreNumber].functionPtr = NULL;
            ReleaseSemaphore(threadTable[coreNumber].hIsProcessing, 1, NULL);
        }
        else {
            if (SuspendThread(threadTable[coreNumber].hThread) == -1) {
                errorAndTerminate("Thread refusing to suspend.", THREAD_ERROR);
            }
        }
    }
}

//------------------------------------------------------------------------------
short getNumberOfProcessorCores(void) {
//------------------------------------------------------------------------------
    SYSTEM_INFO systemInfo;

    GetSystemInfo(&systemInfo);
    return (int)systemInfo.dwNumberOfProcessors;
}

//------------------------------------------------------------------------------
void createThreadOnCore(int core) {
//------------------------------------------------------------------------------
    char statusBuffer[100];

    // Verify that no thread already exists on the core, then create a new one
    if (threadTable[core].hThread != NULL) return;
    threadTable[core].hThread = CreateThread(NULL, 0,
                                  (LPTHREAD_START_ROUTINE)processThreadTasks,
                                    (void*)core, 0, NULL);
    sprintf(statusBuffer, "isThreadAvailable%d", core);
    threadTable[core].hIsProcessing = CreateSemaphore(NULL, 1, 1,
                                    statusBuffer);
    if (threadTable[core].hThread == NULL) {
        errorAndTerminate("Thread creation failed.", THREAD_ERROR);
    }
    if (SetThreadAffinityMask(threadTable[core].hThread,
        (DWORD_PTR)pow(2, core)) == 0) {
        warning("Setting thread core affinity failed.");
    }
    // Update the status
    sprintf(statusBuffer, "Produced thread on core %d.", core);
    status(statusBuffer);
}

//--------------------------------------------------------------------------------
void createMultipleThreads(int numberOfThreads, int processorOffset) {
//--------------------------------------------------------------------------------
    short numberOfProcessors = getNumberOfProcessorCores();

    // Ensure that at least one core is selected
    if (numberOfThreads <= 0) {
        numberOfThreads = 1;
    }
    else if (numberOfThreads > numberOfProcessors) {
        numberOfThreads = numberOfProcessors;
    }
    if (numberOfThreads > MAX_THREADS) {
        numberOfThreads = MAX_THREADS;
    }
    // Setup a semaphore for free thread availability and assignment
    hFreeThreadSemaphore = CreateSemaphore(NULL, numberOfThreads,
                            numberOfThreads, "availableThread");
    if (hFreeThreadSemaphore == NULL) {
        errorAndTerminate("Semaphore creation failed.", SEMAPHORE_ERROR);
    }
    status("Free thread semaphore now available.");
    // Ensure that the processor offset is not greater than the number of CPUs
    if (numberOfThreads + processorOffset > numberOfProcessors) {
        processorOffset = numberOfProcessors - numberOfThreads;
    }
    for (int coreNumber = processorOffset;
        coreNumber < numberOfThreads + processorOffset; coreNumber++) {
```

```
        createThreadOnCore(coreNumber);
    }
    numberOfAvailableThreads = numberOfThreads;
}

//--------------------------------------------------------------------------------
short getNumberOfAvailableThreads(void) {
//--------------------------------------------------------------------------------
    return numberOfAvailableThreads;
}

//--------------------------------------------------------------------------------
void assignThreadFunction(int core,
                           void (*function)(IMAGE_LIMITS,IMAGE_LIMITS,char*),
                           IMAGE_LIMITS image1,
                           IMAGE_LIMITS image2,
                           char* output) {
//--------------------------------------------------------------------------------
    char statusBuffer[100];

    if (hFreeThreadSemaphore == NULL) {
        createMultipleThreads(1000, 0);
    }
    // Wait for a thread to become avialable or free its tasks
    if (WaitForSingleObject(threadTable[core].hIsProcessing, INFINITE) ==
            WAIT_OBJECT_0) {
        if (threadTable[core].hThread != NULL ){
            threadTable[core].functionPtr = function;
            threadTable[core].image1 = image1;
            threadTable[core].image2 = image2;
            threadTable[core].output = output;
        }
    }
    else {
        errorAndTerminate("Unable to obtain semaphore access to task thread.",
            SEMAPHORE_ERROR);
    }
    if (ResumeThread(threadTable[core].hThread) == -1) {
        sprintf(statusBuffer,
            "Thread on processor %d not awaking from suspend.", core);
        errorAndTerminate(statusBuffer, THREAD_ERROR);
    }
```

```
}

//------------------------------------------------------------------------------
void waitForAllCores(void) {
//------------------------------------------------------------------------------
    if (hFreeThreadSemaphore == NULL) {
        createMultipleThreads(1000, 0);
    }
    // Otain semaphores as cores are freed then release all when done
    for (int core = 0; core < numberOfAvailableThreads; core++) {
        if (WaitForSingleObject(threadTable[core].hIsProcessing,
                INFINITE) != WAIT_OBJECT_0) {
            errorAndTerminate("Unable to obtain semaphore access for threads.",
                SEMAPHORE_ERROR);
        }
    }
    for (int core = 0; core < numberOfAvailableThreads; core++) {
        ReleaseSemaphore(threadTable[core].hIsProcessing, 1, NULL);
    }
}
```

B17 TIFF.H

```
//------------------------------------------------------------------------------
// tiff.h - header file for reading and writing TIFF files
//------------------------------------------------------------------------------
#ifndef TIFF_H
#define TIFF_H
//------------------------------------------------------------------------------

//------------------------------------------------------------------------------
// Structures
//------------------------------------------------------------------------------

typedef struct {
    unsigned short numberOfTags;
    unsigned long newSubfileType;
    unsigned long imageWidth;
    unsigned long imageLength;
    unsigned short* bitsPerSample;
    unsigned short compression;
    unsigned short photometric;
    unsigned long* stripOffsets;
    unsigned long stripsPerImage;
    unsigned short samplesPerPixel;
    unsigned long rowsPerStrip;
    unsigned long* stripByteCounts;
    unsigned long xResolution[2];
    unsigned long yResolution[2];
     unsigned short resolutionUnit;
    unsigned long offsetToNextIFD;
} TIFF_IMAGE_FILE_DIRECTORY;

typedef struct {
    unsigned short tagID;
    unsigned short dataType;
    unsigned int numberOfValues;
} TIFF_BASIC_TAG;

typedef struct {
    unsigned char red;
    unsigned char green;
    unsigned char blue;
    unsigned char reserved;
} TIFF_RGB;

typedef struct {
    TIFF_IMAGE_FILE_DIRECTORY ifd;
    TIFF_RGB* imageData;
} TIFF;

//------------------------------------------------------------------------------
// Functions
//------------------------------------------------------------------------------

TIFF readTIFF(FILE* imageFile, int fileIsLittleEndian);
// Description:    Reads the TIFF structure from disk
// Inputs:         A FILE pointer to a previously opened file
// Returns:        The bitmap structure containing the file data
//------------------------------------------------------------------------------

void writeTIFF(FILE* imageFile, TIFF tiff);
```

```
// Description:    Writes the TIFF structure to disk
// Inputs:         An open FILE pointer and a TIFF image to write to disk
// Returns:        None
//------------------------------------------------------------------------------

TIFF_RGB getTIFFPixel(TIFF* tiff, int x, int y);
// Description:    Returns the pixel at the point x,y
// Inputs:         A pointer to the image data to be read, and coordinates of a
//                 point
// Returns:        The RGB structure correlating to the coordinate
//------------------------------------------------------------------------------

void setTIFFPixel(TIFF* tiff, int x, int y, TIFF_RGB value);
// Description:    Sets the pixel at the point x,y
// Inputs:         A pointer to the image data to be read, coordinates of a point
//                 and the value to save
// Returns:        None
//------------------------------------------------------------------------------

unsigned long getTIFFWidth(TIFF* tiff);
// Description:    Returns the width of the TIFF file
// Inputs:         A pointer to the image data to be assessed
// Returns:        Width of the image
//------------------------------------------------------------------------------

unsigned long getTIFFLength(TIFF* tiff);
// Description:    Returns the length of the TIFF file
// Inputs:         A pointer to the image data to be assessed
// Returns:        Height of the image
//------------------------------------------------------------------------------

void setTIFFSize(TIFF* tiff, unsigned long width, unsigned long length);
// Description:    Sets the size of the TIFF file
// Inputs:         A pointer to the image data to be read,
//                 the new width of the image,
//                 the new height of the image
// Returns:        None

#endif
```

B18 TIFF.C

```
//-------------------------------------------------------------------------------
// tiff.c - implementation file for reading and writing TIFF files
//-------------------------------------------------------------------------------

#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "tiff.h"
#include "boolean.h"
#include "error.h"

#define STRIP_SIZE 8192

//-------------------------------------------------------------------------------
// Structures
//-------------------------------------------------------------------------------

typedef enum {
    UNSIGNED_SHORT = 3,
    UNSIGNED_LONG = 4,
    UNSIGNED_RATIONAL = 5,
    SIGNED_SHORT = 8,
    SIGNED_LONG = 9,
    SIGNED_RATIONAL = 10
} DATA_TYPES;

typedef enum {
    SHORT_SIZE = 2,
    LONG_SIZE = 4,
    HEADER_SIZE = 8,
    TAG_SIZE = 12
} DATA_SIZES;

typedef enum {
    NEW_SUBFILE_TYPE = 254,
    IMAGE_WIDTH = 256,
    IMAGE_LENGTH = 257,
    BITS_PER_SAMPLE = 258,
    COMPRESSION = 259,
    PHOTOMETRIC = 262,
    STRIP_OFFSETS = 273,
    SAMPLES_PER_PIXEL = 277,
    ROWS_PER_STRIP = 278,
    STRIP_BYTE_COUNT = 279,
    X_RESOLUTION = 282,
    Y_RESOLUTION = 283,
     RESOLUTION_UNIT = 296,
    LAST_TAG
} TAG_IDS;

//------------------------------------------------------------------------------
// Global Variables
//------------------------------------------------------------------------------

static short convertEndianness;

//------------------------------------------------------------------------------
int machineIsLittleEndian(void) {
//------------------------------------------------------------------------------
    int endiannessInteger = 1;

    return *(char*)&endiannessInteger;
}

//------------------------------------------------------------------------------
void swapEndianness(unsigned char* dataPointer, unsigned int numberOfBytes) {
//------------------------------------------------------------------------------
    unsigned int lowerSection = 0;
    unsigned int upperSection = numberOfBytes - 1;
    unsigned int swapTemp;

    if (dataPointer != NULL && convertEndianness) {
        while (lowerSection < upperSection) {
            swapTemp = dataPointer[lowerSection];
            dataPointer[lowerSection] = dataPointer[upperSection];
            dataPointer[upperSection] = swapTemp;
            lowerSection++;
            upperSection--;
        }
    }
```

```
}

//-------------------------------------------------------------------------------
unsigned long readUnsignedLong(FILE* imageFile, unsigned int numberOfBytes) {
//-------------------------------------------------------------------------------
    unsigned long tagValue = 0;

    if (fread((unsigned long*)&tagValue, numberOfBytes, 1, imageFile) == 0) {
        errorAndTerminate("Tag value unreadable.", IMAGE_IO_ERROR);
    }
    swapEndianness(&tagValue, numberOfBytes);
    return tagValue;
}

//-------------------------------------------------------------------------------
unsigned char* readTagProperties(FILE* imageFile, TIFF_BASIC_TAG* tag) {
//-------------------------------------------------------------------------------
    unsigned char* dataPointer;
    unsigned long* longPointer;
    unsigned short* shortPointer;
    unsigned short readInSize = 0;

    if (imageFile == NULL || tag == NULL) {
        return NULL;
    }
    // Organise the data type and move to the position where the data is stored
    switch(tag->dataType) {
        case UNSIGNED_LONG:
        case SIGNED_LONG:
            readInSize = LONG_SIZE;
            if (tag->numberOfValues > 1) {
                fseek(imageFile, readUnsignedLong(imageFile, LONG_SIZE),
                    SEEK_SET);
            }
            break;
        case UNSIGNED_SHORT:
        case SIGNED_SHORT:
            readInSize = SHORT_SIZE;
            if (tag->numberOfValues > 2) {
                fseek(imageFile, readUnsignedLong(imageFile, LONG_SIZE),
                    SEEK_SET);
            }
            break;
        default:
            return NULL;
    }
    // Allocate the storage size
    dataPointer = (unsigned char*)malloc(tag->numberOfValues * readInSize);
    if (dataPointer == NULL) {
        errorAndTerminate("Insufficient memory to allocate for properties.",
            INSUFFICIENT_MEMORY);
    }
    longPointer = dataPointer;
    shortPointer = dataPointer;
    // Read the values
    for (int value = 0; value < tag->numberOfValues; value++) {
        switch(tag->dataType) {
            case UNSIGNED_LONG:
            case SIGNED_LONG:
                longPointer[value] = readUnsignedLong(imageFile, readInSize);
                break;
            case UNSIGNED_SHORT:
            case SIGNED_SHORT:
                shortPointer[value] = readUnsignedLong(imageFile, readInSize);
        }
    }
    return dataPointer;
}

//--------------------------------------------------------------------------------
void writeUnsignedLong(FILE* imageFile,
                       unsigned long tagValue,
                       unsigned int numberOfBytes) {
//--------------------------------------------------------------------------------
    if (fwrite((char *)&tagValue, numberOfBytes, 1, imageFile) == 0) {
        errorAndTerminate("Tag value not written.", IMAGE_IO_ERROR);
    }
}

//--------------------------------------------------------------------------------
void writeBasicTag(FILE* imageFile, TIFF_BASIC_TAG* tag) {
//--------------------------------------------------------------------------------
    if (fwrite((char *)tag, sizeof(TIFF_BASIC_TAG), 1, imageFile) == 0) {
        errorAndTerminate("Tag header properties not written.",IMAGE_IO_ERROR);
    }
}

//--------------------------------------------------------------------------------
TIFF readTIFF(FILE* imageFile, int fileIsLittleEndian) {
//--------------------------------------------------------------------------------
    TIFF tiff;
    TIFF_BASIC_TAG tag;
    unsigned long filePosition;
    unsigned long dataPosition = 0;
    TIFF_RGB* imageDataStrip;
    int offsetToFirstIFD;
    unsigned short readInSize;
    unsigned long* longData;
    unsigned short* shortData;
    char statusBuffer[100];

    // Assess the endianness of the current machine and modify bytes if needed
    convertEndianness = !(machineIsLittleEndian() ^ fileIsLittleEndian);
    // Read the file header and the first image file directory
    if (fread((char*)&offsetToFirstIFD, LONG_SIZE, 1, imageFile) == 0) {
        errorAndTerminate("Unable to read image file header.", IMAGE_IO_ERROR);
    }
    swapEndianness((char*)&offsetToFirstIFD, sizeof(int));
    fseek(imageFile, offsetToFirstIFD, SEEK_SET);
    if (fread((short*)&tiff.ifd.numberOfTags,
        sizeof(short), 1, imageFile) == 0) {
        errorAndTerminate("Unable to read image file directory properties.",
            IMAGE_IO_ERROR);
    }
    swapEndianness((char*)&tiff.ifd.numberOfTags,sizeof(tiff.ifd.numberOfTags));
    // Individually read each tag and sort the contents
    for (int tagNumber = 0; tagNumber < tiff.ifd.numberOfTags; tagNumber++) {
        if (fread((char *)&tag, sizeof(TIFF_BASIC_TAG), 1, imageFile) == 0) {
            errorAndTerminate("Unable to read tag properties.",IMAGE_IO_ERROR);
        }
        swapEndianness((char*)&tag.tagID, sizeof(tag.tagID));
        swapEndianness((char*)&tag.dataType, sizeof(tag.dataType));
        swapEndianness((char*)&tag.numberOfValues, sizeof(tag.numberOfValues));
        // Store the current position in the file
        filePosition = ftell(imageFile);
        // Read values based on data type
```

```
switch(tag.tagID) {
    case NEW_SUBFILE_TYPE:
        if (tag.dataType == UNSIGNED_LONG ||
            tag.dataType == SIGNED_LONG) {
            tiff.ifd.newSubfileType =
                readUnsignedLong(imageFile, LONG_SIZE);
        }
        break;
    case IMAGE_WIDTH:
        if (tag.dataType == UNSIGNED_LONG ||
            tag.dataType == SIGNED_LONG) {
            tiff.ifd.imageWidth =
                readUnsignedLong(imageFile, LONG_SIZE);
        }
        else if (tag.dataType == UNSIGNED_SHORT ||
            tag.dataType == SIGNED_SHORT) {
            tiff.ifd.imageWidth =
                readUnsignedLong(imageFile, SHORT_SIZE);
        }
        break;
    case IMAGE_LENGTH:
        if (tag.dataType == UNSIGNED_LONG ||
            tag.dataType == SIGNED_LONG) {
            tiff.ifd.imageLength =
                readUnsignedLong(imageFile, LONG_SIZE);
        }
        else if (tag.dataType == UNSIGNED_SHORT ||
            tag.dataType == SIGNED_SHORT) {
            tiff.ifd.imageLength =
                readUnsignedLong(imageFile, SHORT_SIZE);
        }
        break;
    case BITS_PER_SAMPLE:
        tiff.ifd.samplesPerPixel = tag.numberOfValues;
        tiff.ifd.bitsPerSample = readTagProperties(imageFile, &tag);
        break;
    case COMPRESSION:
        if (tag.dataType == UNSIGNED_SHORT ||
            tag.dataType == SIGNED_SHORT) {
            tiff.ifd.compression =
                readUnsignedLong(imageFile, SHORT_SIZE);
        }
```

```
                break;
            case PHOTOMETRIC:
                if (tag.dataType == UNSIGNED_SHORT ||
                    tag.dataType == SIGNED_SHORT) {
                    tiff.ifd.photometric =
                        readUnsignedLong(imageFile, SHORT_SIZE);
                }
                break;
            case STRIP_OFFSETS:
                tiff.ifd.stripsPerImage = tag.numberOfValues;
                tiff.ifd.stripOffsets = readTagProperties(imageFile, &tag);
                break;
            case SAMPLES_PER_PIXEL:
                if (tiff.ifd.samplesPerPixel != readUnsignedLong(imageFile,
                    SHORT_SIZE)) {
                    errorAndTerminate("Conflict in pixel sampling tags.",
                        UNSUPPORTED_TYPE);
                }
                break;
            case ROWS_PER_STRIP:
                if (tag.dataType == UNSIGNED_LONG ||
                    tag.dataType == SIGNED_LONG) {
                    tiff.ifd.rowsPerStrip =
                        readUnsignedLong(imageFile, LONG_SIZE);
                }
                else if (tag.dataType == UNSIGNED_SHORT ||
                    tag.dataType == SIGNED_SHORT) {
                    tiff.ifd.rowsPerStrip =
                        readUnsignedLong(imageFile, SHORT_SIZE);
                }
                break;
            case STRIP_BYTE_COUNT:
                if (tiff.ifd.stripsPerImage != tag.numberOfValues) {
                    errorAndTerminate("Incorrect tag offset value.",
                        IMAGE_IO_ERROR);
                }
                tiff.ifd.stripByteCounts = readTagProperties(imageFile, &tag);
                break;
            case X_RESOLUTION:
                if (tag.dataType == UNSIGNED_RATIONAL ||
                    tag.dataType == SIGNED_RATIONAL) {
                    // Move to the position where the data is stored and read
                    fseek(imageFile, readUnsignedLong(imageFile, LONG_SIZE),
                        SEEK_SET);
                    tiff.ifd.xResolution[0] =
                        readUnsignedLong(imageFile, LONG_SIZE);
                    tiff.ifd.xResolution[1] =
                        readUnsignedLong(imageFile, LONG_SIZE);
                }
                break;
            case Y_RESOLUTION:
                if (tag.dataType == UNSIGNED_RATIONAL ||
                    tag.dataType == SIGNED_RATIONAL) {
                    // Move to the position where the data is stored and read
                    fseek(imageFile, readUnsignedLong(imageFile, LONG_SIZE),
                        SEEK_SET);
                    tiff.ifd.yResolution[0] =
                        readUnsignedLong(imageFile, LONG_SIZE);
                    tiff.ifd.yResolution[1] =
                        readUnsignedLong(imageFile, LONG_SIZE);
                }
                break;
            case RESOLUTION_UNIT:
                if (tag.dataType == UNSIGNED_SHORT ||
                    tag.dataType == SIGNED_SHORT) {
                    tiff.ifd.resolutionUnit =
                        readUnsignedLong(imageFile, SHORT_SIZE);
                }
                break;
        }
        // Move file pointer to the position for the next tag
        fseek(imageFile, filePosition + 4, SEEK_SET);
    }
    if (fread((short *)&tiff.ifd.offsetToNextIFD, SHORT_SIZE, 1,
        imageFile) == 0) {
        errorAndTerminate("Unable to read image file directory properties.",
            IMAGE_IO_ERROR);
    }
    // Check the compression status and colour types
    status("Acquired the TIFF headers.");
    if (tiff.ifd.compression != 1) {
        errorAndTerminate("TIFF compression unsupported.", UNSUPPORTED_TYPE);
    }
    if (!(tiff.ifd.photometric == 1 || tiff.ifd.photometric == 2)) {
        errorAndTerminate("Unsupported TIFF colour space.", UNSUPPORTED_TYPE);
    }
    if (tiff.ifd.samplesPerPixel < 1){
        errorAndTerminate("Unsupported TIFF sampling per pixel.",
            UNSUPPORTED_TYPE);
    }

    if (tiff.ifd.bitsPerSample == NULL ||
        tiff.ifd.stripByteCounts == NULL ||
        tiff.ifd.stripOffsets == NULL){
            errorAndTerminate("Incomplete TIFF header properties.",
                UNSUPPORTED_TYPE);
    }
    for (short sample = 0; sample < tiff.ifd.samplesPerPixel; sample++) {
        if (tiff.ifd.bitsPerSample[sample] != 8){
            errorAndTerminate("TIFF pixel bitrate unsupported.",
                UNSUPPORTED_TYPE);
        }
    }
    // Notify of status and read bitmap image data
    sprintf(statusBuffer, "Beginning read of %d bytes.",
        tiff.ifd.imageWidth * tiff.ifd.imageLength * sizeof(TIFF_RGB));
    status(statusBuffer);
    tiff.imageData = (TIFF_RGB*)malloc(tiff.ifd.imageWidth *
        tiff.ifd.imageLength * sizeof(TIFF_RGB));
    if (tiff.imageData == NULL) {
        errorAndTerminate("Insufficient memory to allocate for image data.",
            INSUFFICIENT_MEMORY);
    }
    for (int strip = 0; strip < tiff.ifd.stripsPerImage; strip++) {
        // Move to the strip and read
        fseek(imageFile, tiff.ifd.stripOffsets[strip], SEEK_SET);
        while (dataPosition < tiff.ifd.stripByteCounts[strip] /
            tiff.ifd.samplesPerPixel) {
            tiff.imageData[dataPosition].red = getc(imageFile);
            if (tiff.ifd.samplesPerPixel >= 2) {
                tiff.imageData[dataPosition].green = getc(imageFile);
            }
            else {
                tiff.imageData[dataPosition].green =
                    tiff.imageData[dataPosition].red;
            }
            if (tiff.ifd.samplesPerPixel >= 3) {
                tiff.imageData[dataPosition].blue = getc(imageFile);
            }
            else {
                tiff.imageData[dataPosition].blue =
                    tiff.imageData[dataPosition].red;
            }
            // If more than three samples, skip the rest
            if (tiff.ifd.samplesPerPixel >= 4) {
                fseek(imageFile, ftell(imageFile) +
                    (tiff.ifd.samplesPerPixel - 3), SEEK_SET);
            }
            dataPosition++;
        }
        if (strip > 0) {
            tiff.ifd.stripByteCounts[0] += tiff.ifd.stripByteCounts[strip];
        }
    }
    tiff.ifd.samplesPerPixel = 3;
    // Merge into one strip
    if (tiff.ifd.stripsPerImage > 1) {
        tiff.ifd.stripsPerImage = 1;
        tiff.ifd.rowsPerStrip = tiff.ifd.imageLength;
    }
    status("Successfully buffered image data.");
    return tiff;
}

//--------------------------------------------------------------------------------
void writeTIFF(FILE* imageFile, TIFF tiff) {
//--------------------------------------------------------------------------------
    TIFF_BASIC_TAG tag;
    unsigned short numberOfTags = 13;
    unsigned long offsetAfterTags = (numberOfTags + 1) * TAG_SIZE + HEADER_SIZE;
    unsigned long filePosition;
    unsigned long stripTagOffset;
    short movePosition = FALSE;
    unsigned long dataPosition = 0;
    char fileSignature[5];

    // Write the preliminary file constructs
    strcpy(fileSignature, machineIsLittleEndian() ? "II*\0" : "MM\0*");
```

```
if (fwrite(fileSignature, 4, 1, imageFile) == 0) {
    errorAndTerminate("Image signature not written.", IMAGE_IO_ERROR);
}
filePosition = 8;
if (fwrite((char *)&filePosition, LONG_SIZE, 1, imageFile) == 0) {
    errorAndTerminate("First IFD pointer not written", IMAGE_IO_ERROR);
}
fseek(imageFile, filePosition, SEEK_SET);
if (fwrite((char *)&numberOfTags, SHORT_SIZE, 1, imageFile) == 0) {
    errorAndTerminate("Number of tags property not written.",
        IMAGE_IO_ERROR);
}
// Individually determine each tag and write the contents
for (int tagNumber = 0; tagNumber < LAST_TAG; tagNumber++) {
    movePosition = TRUE;
    filePosition = ftell(imageFile);
    // Write the tags based on the tag ID
    tag.tagID = tagNumber;
    tag.numberOfValues = 1;
    switch(tagNumber) {
        case NEW_SUBFILE_TYPE:
            tag.dataType = UNSIGNED_LONG;
            writeBasicTag(imageFile, &tag);
            writeUnsignedLong(imageFile, tiff.ifd.newSubfileType,LONG_SIZE);
            break;
        case IMAGE_WIDTH:
            tag.dataType = UNSIGNED_LONG;
            writeBasicTag(imageFile, &tag);
            writeUnsignedLong(imageFile, tiff.ifd.imageWidth, LONG_SIZE);
            break;
        case IMAGE_LENGTH:
            tag.dataType = UNSIGNED_LONG;
            writeBasicTag(imageFile, &tag);
            writeUnsignedLong(imageFile, tiff.ifd.imageLength, LONG_SIZE);
            break;
        case BITS_PER_SAMPLE:
            tag.dataType = UNSIGNED_SHORT;
            tag.numberOfValues = tiff.ifd.samplesPerPixel;
            writeBasicTag(imageFile, &tag);
            if (tag.numberOfValues > 2) {
                writeUnsignedLong(imageFile, offsetAfterTags, SHORT_SIZE);
                // Move to where the data is to be written
                fseek(imageFile, offsetAfterTags, SEEK_SET);
                // Increment the position after the tags for the image data
                offsetAfterTags += tag.numberOfValues * SHORT_SIZE;
            }
            for (int data = 0; data < tag.numberOfValues; data++) {
                writeUnsignedLong(imageFile, tiff.ifd.bitsPerSample[data],
                    SHORT_SIZE);
            }
            break;
        case COMPRESSION:
            tag.dataType = UNSIGNED_SHORT;
            writeBasicTag(imageFile, &tag);
            writeUnsignedLong(imageFile, tiff.ifd.compression, SHORT_SIZE);
            break;
        case PHOTOMETRIC:
            tag.dataType = UNSIGNED_SHORT;
            writeBasicTag(imageFile, &tag);
            writeUnsignedLong(imageFile, tiff.ifd.photometric, SHORT_SIZE);
            break;
        case STRIP_OFFSETS:
            tag.dataType = UNSIGNED_LONG;
            tag.numberOfValues = tiff.ifd.stripsPerImage;
            writeBasicTag(imageFile, &tag);
            stripTagOffset = ftell(imageFile);
            break;
        case SAMPLES_PER_PIXEL:
            tag.dataType = UNSIGNED_SHORT;
            writeBasicTag(imageFile, &tag);
            writeUnsignedLong(imageFile, tiff.ifd.samplesPerPixel,
                SHORT_SIZE);
            break;
        case ROWS_PER_STRIP:
            tag.dataType = UNSIGNED_LONG;
            writeBasicTag(imageFile, &tag);
            writeUnsignedLong(imageFile, tiff.ifd.rowsPerStrip, LONG_SIZE);
            break;
        case STRIP_BYTE_COUNT:
            tag.dataType = UNSIGNED_LONG;
            tag.numberOfValues = tiff.ifd.stripsPerImage;
            writeBasicTag(imageFile, &tag);
            if (tag.numberOfValues > 1) {
                writeUnsignedLong(imageFile, offsetAfterTags, LONG_SIZE);
                // Move to where the data is to be written
                fseek(imageFile, offsetAfterTags, SEEK_SET);
                // Increment the position after the tags for the image data
                offsetAfterTags += tag.numberOfValues * LONG_SIZE;
            }
            for (int data = 0; data < tag.numberOfValues; data++) {
                writeUnsignedLong(imageFile,
                    tiff.ifd.stripByteCounts[data], LONG_SIZE);
            }
            break;
        case X_RESOLUTION:
            tag.dataType = UNSIGNED_RATIONAL;
            writeBasicTag(imageFile, &tag);
            writeUnsignedLong(imageFile, offsetAfterTags, LONG_SIZE);
            // Move to where the data is to be written and write
            fseek(imageFile, offsetAfterTags, SEEK_SET);
            writeUnsignedLong(imageFile, tiff.ifd.xResolution[0],
                LONG_SIZE);
            writeUnsignedLong(imageFile, tiff.ifd.xResolution[1],
                LONG_SIZE);
            // Increment the position after the tags for the image data
            offsetAfterTags += 2 * LONG_SIZE;
            break;
        case Y_RESOLUTION:
            tag.dataType = UNSIGNED_RATIONAL;
            writeBasicTag(imageFile, &tag);
            writeUnsignedLong(imageFile, offsetAfterTags, LONG_SIZE);
            // Move to where the data is to be written and write
            fseek(imageFile, offsetAfterTags, SEEK_SET);
            writeUnsignedLong(imageFile, tiff.ifd.yResolution[0],
                LONG_SIZE);
            writeUnsignedLong(imageFile, tiff.ifd.yResolution[1],
                LONG_SIZE);
            // Increment the position after the tags for the image data
            offsetAfterTags += 2 * LONG_SIZE;
            break;
        case RESOLUTION_UNIT:
            tag.dataType = UNSIGNED_SHORT;
            writeBasicTag(imageFile, &tag);
            writeUnsignedLong(imageFile, tiff.ifd.resolutionUnit,
                SHORT_SIZE);
            break;
            default:
                movePosition = FALSE;
        }
        // Move in the file to the position for the next tag
        if (movePosition) {
            fseek(imageFile, filePosition + TAG_SIZE, SEEK_SET);
        }
    }
    writeUnsignedLong(imageFile, 0, LONG_SIZE);
    // Return to write the next available location for the image data strip
    fseek(imageFile, stripTagOffset, SEEK_SET);
    writeUnsignedLong(imageFile, offsetAfterTags, LONG_SIZE);
    status("Written image headers.");
    // Write the TIFF image data
    fseek(imageFile, offsetAfterTags, SEEK_SET);
    while (dataPosition < tiff.ifd.imageWidth * tiff.ifd.imageLength) {
        fputc(tiff.imageData[dataPosition].red, imageFile);
        fputc(tiff.imageData[dataPosition].green, imageFile);
        fputc(tiff.imageData[dataPosition].blue, imageFile);
        dataPosition++;
    }
    status("Written image data.");
}

//--------------------------------------------------------------------------------
TIFF_RGB getTIFFPixel(TIFF* tiff, int x, int y) {
//--------------------------------------------------------------------------------
    // Sanity check first
    if (tiff->imageData == NULL) {
        TIFF_RGB newTIFF;
        return newTIFF;
    }
    if (x > tiff->ifd.imageWidth) {
        x = tiff->ifd.imageWidth;
    }
    if (y > tiff->ifd.imageLength) {
        y = tiff->ifd.imageLength;
    }
    return tiff->imageData[y * tiff->ifd.imageWidth + x];
}

//--------------------------------------------------------------------------------
```

```
void setTIFFPixel(TIFF* tiff, int x, int y, TIFF_RGB value) {
//----------------------------------------------------------------------------
    // Sanity check first
    if (tiff->imageData == NULL) {
        return;
    }
    if (x > tiff->ifd.imageWidth) {
        setTIFFSize(tiff, x, tiff->ifd.imageLength);
    }
    if (y > tiff->ifd.imageLength) {
        setTIFFSize(tiff, tiff->ifd.imageWidth, y);
    }
    tiff->imageData[y * tiff->ifd.imageWidth + x] = value;
}

//----------------------------------------------------------------------------
unsigned long getTIFFWidth(TIFF* tiff) {
//----------------------------------------------------------------------------
    return tiff->ifd.imageWidth;
}

//----------------------------------------------------------------------------
unsigned long getTIFFLength(TIFF* tiff) {
//----------------------------------------------------------------------------
    return tiff->ifd.imageLength;
}

//----------------------------------------------------------------------------
void setTIFFSize(TIFF* tiff, unsigned long width, unsigned long length) {
//----------------------------------------------------------------------------
    // Change the general size of the image and data
    tiff->ifd.stripByteCounts[0] = (width * length * tiff->ifd.samplesPerPixel);
    tiff->ifd.imageWidth = width;
    tiff->ifd.imageLength = length;
    tiff->ifd.rowsPerStrip = tiff->ifd.imageLength;
    tiff->imageData = (TIFF_RGB*)realloc(tiff->imageData,
        width * length * sizeof(TIFF_RGB));
    if (tiff->imageData == NULL) {
        errorAndTerminate("Insufficient memory to allocate for TIFF image "
            "data.", INSUFFICIENT_MEMORY);
    }
}
```