Fast Probabilistic Inference and GPU Video Processing

A dissertation submitted to the University of Dublin
for the degree of Doctor of Philosophy

Francis Kelly
Trinity College Dublin, May 2006

Signal Processing and Media Applications
Department of Electronic and Electrical Engineering
Trinity College Dublin

To my family and friends.

Declaration

I hereby declare that this thesis has not been submitted as an exercise for a degree at this or any other University and that it is entirely my own work.

I agree that the Library may lend or copy this thesis upon request.

Signed,

Francis Kelly

May 30, 2006.

Summary

This thesis is concerned with two main topics: probabilistic methods for motion estimation and the use of the GPU for fast video processing. The motion estimation problem is presented using a Bayesian methodology. Within this framework a new maximum-a-posteriori (MAP) estimation algorithm based on Belief Propagation (BP) is developed. This new algorithm also exploits a technique known as candidate selection for motion estimation. This is shown to be a powerful method for motion refinement when given some initial estimate of the motion field. BP is used to overcome problems such as the Aperture Effect which can arise in maximum-likelihood motion estimators. The new BP based algorithm matches and in some cases exceeds the performance of another popular MAP estimation technique known as Iterated Conditional Modes (ICM). It also performs favourably against the 3DRS (3D Recursive Search) candidate selection based motion estimator.

A new algorithm for on-line global motion estimation is also presented. This is based on a Monte Carlo technique known as the Particle Filter, which is used to incorporate temporal information into the estimation process. The Particle Filter algorithm can solve ambiguous matching problems which can arise due to the image content. The Particle Filter is then extended to include an estimate for an Iterative Re-weighted Least Squares (IRLS) algorithm. This is used as a diagnostic tool for determining whether random or impulsive shake is present in a sequence.

The GPUs (Graphics Processing Unit) in modern graphics hardware are extremely powerful, with performance increasing rapidly year on year. The use of the GPU for general purpose computations (GPGPU) is a fast-growing area of research. An introduction to the GPU and some of the techniques of GPGPU, as well as a review of the state of the art in this area, are provided. Finally, it is shown how the GPU may be used to accelerate various video processing algorithms. In some cases this provides performance which is much faster than real-time, resulting is processing speed greater than expensive industrial hardware units.

Acknowledgments

At last I am at the stage where the work is nearly done and I can start thanking all those people who have helped to make this thesis possible. There really are too many to give them all a mention here but I would like to acknowledge a few people.

Firstly my deep gratitude to my supervisor Dr. Anil Kokaram for his constant help, advice, ideas, and above all patience over the last four years. And also for somehow managing to find the funding to support me along the way. Thanks boss.

I would also like to thank all the staff and postgraduates of the Electronic and Electrical Engineering Department of Trinity College who have helped me over the years, from my undergraduate degree right through to this thesis. In particular though the following people who, as well as going out of their way to help me on many occasions, have also become good friends: Claire Gallagher, Niall Rea, Hugh Denman, François Pitiè, Andy Crawford, and Louise Moriarty.

Finally I would like to thank all my friends and especially my wonderful family for always being there for me.

Go raibh míle maith agaibh go léir.

Contents

Abbreviations

2D Two Dimensional

3D Three Dimensional

BM Block Matching

BP Belief Propagation

CPU Central Processing Unit

DFD Displaced Frame Difference

DFT Discrete Fourier Transform

FBM Full Search Block Matching

FFT Fast Fourier Transform

FPS Frames Per Second

GMC Global Motion Compensation

GME Global Motion Estimation

GPU Graphics Processing Unit

GPGPU General-Purpose Computation on GPUs

ICM Iterated Conditional Modes

IRLS Iterative Re-weighted Least Squares

M2SE Modified Mean Square Error

MAD Mean Absolute Difference

MAP Maximum-a-Posteriori

MCMC Markov Chain Monte Carlo

MMSE Minimum Mean Squared Error

MRF Markov Random Field

MSE Mean Squared Error

SAD Sum Absolute Difference

PCIe PCI Express

Pel Pixel or Picture Element

WBME Wiener Based Motion Estimation

1 Introduction

For the first time in the history of the consumer desktop PC, real-time statistical video processing is possible. The performance growth and complexity of the central processing unit (CPU) of the PC has been incredible, keeping up with Moore's prediction [135]. However even this is still not enough to provide the processing power necessary for sophisticated video processing in real-time. In order to accomplish this feat another significant boost in computational resources was required. This has been provided by another more powerful processor sitting relatively un-utilised in a modern PC: the GPU (Graphics Processing Unit). In the last decade or so the performance growth of this processor has been much greater than that of CPUs. Unfortunately exploiting this processing power for tasks such as video processing is not yet a straightforward matter. To do this two things are required, (i) an in-depth knowledge of the architecture of a modern GPU and (ii) knowledge of the graphics-centric programming environment needed to interface with the GPU.

This thesis brings together two areas of research, statistical video processing and GPU acceleration. There is a nice symmetry in this as the first is quite a mature area of research whereas the second is still very young. This thesis may be divided into the following two parts:

1. Investigating probabilistic motion estimation algorithms.

2. Exploiting the GPU for real-time video processing.

1.1 Motion Estimation

Motion estimation is one of the most important aspects of many video processing and computer vision tasks. Because motion estimation is the enabling technology in video coding standards such as MPEG-2, upon which digital television broadcasting is based, it has been the focus of much research effort over the years. A distinction is usually made between the local motion estimation of objects in a scene and the global motion of the camera that captured the scene. There is a huge amount of work in the literature devoted to local and global motion estimation techniques. Suffice to say a comprehensive review of this work is not feasible here. Chapter 2 provides a review of some of the more established motion estimation methods developed during this time. Also, by considering a Bayesian approach, a common framework is provided to unify many of these algorithms. Chapters 3 and 4 then provide details of two new algorithms developed for local and global motion estimation respectively.

1.1.1 Candidate Selection Motion Estimation

The first contribution of this thesis is a novel local motion estimation algorithm. This algorithm is based on a technique known as candidate selection. In a candidate selection algorithm a limited number of candidate vectors are tested at each site where a motion estimate is required. These candidates are chosen to satisfy spatial and/or temporal smoothness constraints. This new algorithm is formulated under a Bayesian framework to yield a maximum-a-posteriori (MAP) estimation problem. Two deterministic methods are then used to solve this MAP problem, Iterated Conditional Modes and Belief Propagation. The application of Belief Propagation to this MAP motion estimation problem is an innovative approach.

1.1.2 On-line Global Motion Estimation

The second major contribution in this thesis is a new method for global motion estimation. This was developed using Sequential Monte Carlo methods, also known as Particle Filters. The motivation for this was to provide an elegant means of incorporating history into the global motion estimation process. Doing this was expected to solve some problems which can arise with other global motion estimation techniques, such as ambiguous matching. This can arise when certain properties within the image data, such as periodic structure yield multiple equally likely motion estimates. Also investigated is the problem of shake diagnosis in video sequences. Two cases are dealt with, random shake and impulsive shake.

1.2 GPU Accelerated Video Processing

Due to the computational effort needed, real-time video processing has traditionally been the domain of hardware manufactures. For example in video coding, motion estimation has long

been provided by dedicated chips to achieve real-time performance. In the television industry, hardware units that perform such tasks as image stabilisation and brightness flicker compensation have recently become available from companies such as Snell & Wilcox [169] and For-A [56]. In that industry real time processing is standard because of the demand on throughput. Real-time processing of this type has not been possible on a standard desktop PC. With the advancement of the GPU however such performance is now within reach.

The performance growth of GPUs over the last decade or so has been phenomenal, easily surpassing the growth curve for CPUs. Today's GPUs are extremely powerful vector machines capable of full 32-bit floating point processing. The performance of the latest high-end GPU from NVIDIA, the GeForce 7800 GTX, has been put at over 150 GFLOPS [153]. Compare this to the 14.8 GFLOPS theoretical peak for a 3.7 GHz Intel Pentium 4 SSE unit [22]. This GPU can be purchased for approximately 500 euros, the Intel processor is closer to 1000 euros[1].

However using all this computational muscle for applications other than graphics processing is not a trivial task. In order to achieve such impressive performance, GPUs have a much simpler pipeline than a modern CPU. They are designed for efficient processing of the four-element vectors associated with 3D graphics. Typically the operations required in graphics processing are highly parallelisable, and the GPU is designed to exploit this. However these architectural features limit the type of operations which can be performed efficiently on the GPU. Hence general purpose processing on the GPU is still not a widespread technique.

Using the GPU for general purpose computing is however a fast growing area of research. It is commonly known by the acronym GPGPU (General Purpose Computation on GPU) [66]. While part one of this thesis is concerned with motion estimation, the second part (chapters 5 & 6) looks at using the GPU for real-time video processing. Chapter 5 gives an introduction to the GPU pipeline and outlines the various aspects that make GPGPU possible. A brief review of the state of the art in GPGPU is also provided. The chapter concludes with a illustrative example of an image processing algorithm known as Texture Synthesis implemented on the GPU. Chapter 6 then details some video processing algorithms which were implemented using the GPU as an image co-processor to the CPU. These algorithms either run faster than real-time or provide substantial speed improvements over similar CPU only implementations. This is the third main contribution of this thesis.

1.3 Thesis Outline

This thesis is divided into two distinct but complimentary parts. Part one covers chapters 2-4, and part two, chapters 5 & 6. Chapters 2 and 5 provide an introduction and review of the state of the art in motion estimation and GPGPU respectively. Chapters 3, 4, and 6 present the main contributions of this research. Finally, chapter 7 gives some final comments and outlines some areas for future work.

[1] At the time of writing, Autumn 2005.

Chapter 2: Motion Estimation: A Review

A review of various motion estimation algorithms relevant to the work presented here is provided. This includes local and global motion estimation algorithms as well as a review of current motion smoothness techniques. It is also shown how these techniques may be formulated under a common Bayesian framework.

Chapter 3: Candidates in Motion

This chapter outlines a new method for performing motion estimation in a Bayesian framework. This is based on a candidate selection approach to motion estimation. Belief Propagation is used to find the maximum-a-posteriori (MAP) solution to the problem. This is compared against another MAP estimation technique known as Iterated Conditional Modes (ICM) and a recursive search motion estimation algorithm.

Chapter 4: On-line Global Motion Estimation

This chapter introduces a new approach to on-line global motion estimation using Sequential Monte Carlo methods. Experimental results show that this new method performs well in situations where previously picture data lead to ambiguous motion estimates. This method can also be used to distinguish between different types of shake which may be present in a sequence.

Chapter 5: GPGPU: An Introduction and Review

Here the GPU is introduced as a useful co-processor to the CPU for image and video processing. A review of GPU architecture and some of the GPGPU literature is provided. To illustrate the possibility of using the GPU for fast image processing a Texture Synthesis algorithm is presented and implemented on the GPU. The results of this algorithm are compared against those obtained in a similar CPU implementation.

Chapter 6: Real-Time Video Processing Algorithms

Having established the potential of the GPU for fast image processing, several video processing algorithms were implemented to take advantage of this extra processing power. These include flicker compensation, motion estimation, and global motion compensation. The implementation details and results obtained are outlined here.

Chapter 7: Final Comments

This final chapter of the thesis accesses the contribution of the research and outlines possible future work. It also discusses the impact of the GPU on video processing and possible changes to the GPU that would make it more attractive for this kind of processing.

Part I

Probabilistic Inference for Motion Estimation

2

Motion Estimation: A Review

EXPLOITING temporal redundancy in image sequences is necessary for most applications in digital video processing. This temporal redundancy manifests principally along motion trajectories within the image sequence. Hence estimating the motion present in an image sequence is a fundamental tool in digital video processing.

The motion estimation problem has attracted heavy research effort over the last thirty years or so. It would be impractical to provide a comprehensive review in these pages due to the sheer volume of work in the literature. There have however been many detailed review publications which cover a broader area than is possible here including [128, 164, 141, 174, 46]. For a good introduction to digital video processing and an overview of common motion estimation methods see Tekalp [183].

This chapter outlines the approaches to motion estimation which have the greatest influence on the work presented later in this thesis. The motion estimation problem will be formulated under a Bayesian framework. Specific motion estimation algorithms may then be shown to arise as a consequence of certain choices of the probability density functions employed.

2.1 Image Sequence Modelling

Image sequence modelling may be categorised into two distinct approaches: an image centric view and a world centric view. In a world centric approach the 3D position, lighting, and size etc. of the elements in the scene are first considered. The subsequent projection of that scene onto a 2D image plane is then used to model the behaviour of the pixels in the image sequence.

The image centric view disregards the source of the information and tries to model the behaviour of the pixel intensities by observing the sequence itself. Ultimately, both views try answer the question "What makes good pictures look good?". The image centric view tries to answer this question by investigating pixel intensities of the captured image data. A world centric view attempts to answer it by trying to understand the 3D world that the images were captured in.

Nagel [142] highlights how difficult it is to derive image sequence constraints based on scene modelling. One particular problem is that of depth ambiguity of the objects in the scene. Motion perpendicular to the image plane causes a change in size of the object. This implies that constraint equations should incorporate depth motion as an independent variable. However, the ambiguity arises in attributing image displacement to motion or a zooming effect; both could yield similar image transformations over small regions. Another problem that can arise is due to lighting effects within a scene, such as shadow, which may cause intensity changes in the sequence which are not due to motion. A world centric model may try to incorporate the actual lighting conditions as well as scene and object 3D structure to overcome this problem. However this can increase further the complexity of the problem. The huge amount of work in the computer vision community of course has most relevance for a world centric approach, e.g. Hartley and Zisserman [72], and Fitzgibbon [55].

If over small displacements in scene space, depth motion is indistinguishable from motion parallel to the image plane, then an image centric motion model will suffice. Almost all motion estimation techniques use an image centric approach and that is the type of model considered here. Most of these techniques to date can be related via the following image sequence model

$$I_n(\mathbf{x}) = I_{n-1}(F(\mathbf{x}, \Theta_n)) \tag{2.1}$$

I_n is the intensity function for the image at frame n and $\mathbf{x} = [x, y]^T$ is the spatial coordinate vector representing the position within a frame. The vector function $F(\mathbf{x}, \Theta_n)$ represents the transformation of the image coordinates caused by the motion between frames n and $n-1$, and Θ_n is the motion parameters vector. Intuitively this model seems correct since it implies that the picture in the current frame can be created by rearranging the position of the pixels in the previous frame. This is the approach taken here.

2.1.1 Local/Global Motion Estimation

The motion in a scene may be separated into two distinct types: local and global motion. Global motion typically refers to the dominant motion in the scene which is due to the motion of the device that captured the image sequence. It includes entire frame effects such as panning, zooming, camera rotation etc. Local motion may then be classified as any motion in the scene other than the global motion. Local motion typically refers to the motion of objects within a scene and global motion refers to the camera motion.

It would seem obvious that an understanding of both the local and global motion of an image sequence is necessary to successfully exploit any temporal redundancy. Traditionally however,

estimation of local and global motion have been treated as two separate processes. This may be attributed to various factors including the complexity and computational cost of estimating both types of motion, while only parameters for one or the other may be necessary, depending on the application. Much of the early work on motion estimation for video coding was based primarily on estimating the local motion of a scene. This was justified as early compression schemes, such as H.261 [67], MPEG–1,2 [138, 139], H.263 [68], were based on motion vectors which gave the lowest prediction error for a block of pixels. More recent video coding standards like MPEG–4 [140] and H.264 [69] exploit the fact that knowledge of the global motion parameters can lead to improved coding efficiency [45]. Indeed there are two-step methods where the global motion parameters may be estimated by examining the distribution of the local motion vectors [89, 31]. Also, if it can be assumed that large areas of the image undergo global motion, then an estimate for the global motion may be used as an initial guess to kick start local motion estimation.

Following the framework of (2.1) makes it easy to model any combination of local and global motion parameters by choosing a suitable transformation. When the motion in the 3D world represents completely unconstrained movement, involving for instance motion perpendicular to the image plane (e.g. zooming), the vector transformation $F(\mathbf{x}, \Theta_n)$ can become non-linear. A six parameter affine transformation is usually sufficient to represent motion such as zooming, rotation, and translation, which are the most common types of camera motion found in image sequences. Hence a suitable model for global motion would be

$$F(\mathbf{x}, \Theta_n) = \mathbf{A}^{\mathbf{g}}\mathbf{x} + \mathbf{d}^{\mathbf{g}} \tag{2.2}$$

where $\mathbf{A}^{\mathbf{g}}$ represents the global affine transformation matrix and $\mathbf{d}^{\mathbf{g}}$ the translational component of the global motion. A similar model may be employed for local motion where $\mathbf{A}(\mathbf{x})$ represent the local affine components and $\mathbf{d}_{n,n-1}(\mathbf{x})$ the local translational components mapping the region centred on position $\mathbf{x}$ in the current frame n into some region in the previous frame $n-1$

$$F(\mathbf{x}, \Theta_n) = \mathbf{A}(\mathbf{x})\mathbf{x} + \mathbf{d}_{n,n-1}(\mathbf{x}) \tag{2.3}$$

Indeed a more complex model might include affine transformations for both local and global motion parameters.

$$F(\mathbf{x}, \Theta_n) = \begin{bmatrix} \mathbf{A}^{\mathbf{g}} & 0 \\ 0 & \mathbf{A}(\mathbf{x}) \end{bmatrix} \mathbf{x} + \mathbf{d}_{n,n-1}(\mathbf{x}) + \mathbf{d}^{\mathbf{g}} \tag{2.4}$$

If the motion between frames is small enough, then zooming and rotation of small portions of an object can be approximated by translation. Hence a model suitable for local motion is a translational one, and is given by

$$F(\mathbf{x}, \Theta_n) = \mathbf{x} + \mathbf{d}_{n,n-1}(\mathbf{x}) \tag{2.5}$$

Unless stated otherwise, this translational only assumption is used for modelling local motion throughout this thesis. The model used throughout for global motion estimation is given by (2.2).

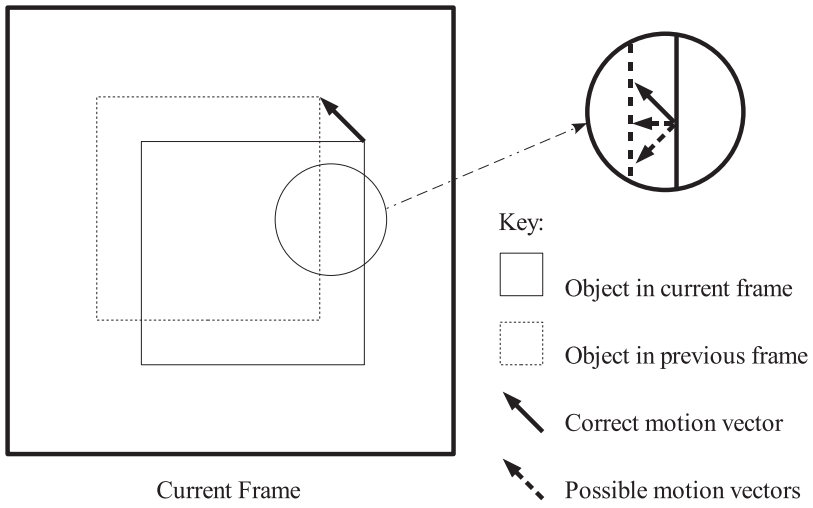


Figure 2.1: *The aperture effect in motion estimation. There can be ambiguity at motion edges when the block straddles the object boundary. The motion estimate is most accurate along the direction of maximum image gradient. Also in areas with low texture, such as the center of the moving object, typical motion estimation algorithms will yield a zero motion vector.*

2.2 Motion Discontinuities

A fundamental feature of the motion field in real sequences is that it will contain discontinuities. A model such as given in (2.1) does not take into account these discontinuities. It only encapsulates the notion that a good motion field allows a good prediction of the current frame n by rearranging elements in the previous frame $n-1$. Some properties of a real sequence should include smoothness of the motion field, spatial discontinuities and temporal discontinuities.

Since objects tend to be contiguous in the image space and all parts of rigid objects tend to undergo the same motion, then the motion field within an object can be expected to be homogenous. This *smoothness* constraint is important for generating realistic motion fields. In practice, without applying this constraint region based estimation can still fail because of the nature of the region itself. Some examples of motion discontinuities are discussed next.

2.2.1 Spatial Discontinuities

The *aperture effect* is a term with many interpretations but will be used here to describe the problems arising from the ill-posed[1] nature of motion estimation. It is not possible to find a unique solution for the motion of a pixel by considering it independently of the other pixels in the image. This is because in a translational motion model, there are two unknowns (the horizontal and vertical components) but only one equation at a pixel site. To overcome this

[1]According to Hadamard's definition a problem is well-posed if it has a unique solution that depends continuously on the data.

motion estimation is typically carried out by first considering the motion of a group of pixels together. This is based on the assumption that the motion within a small region is constant. This is known as the Smooth Local Flow or the Constant Local Flow assumption. If it is the case, that all the pixels in the region move with the same motion, then many equations can be set up with just two unknowns and a motion estimate for the region obtained.

However this Smooth Local Flow assumption can lead to other problems, as shown in Figure 2.1. The rectangular object of uniform intensity is moving diagonally and so has two non zero components of motion. Depending on the size and location of the region used for motion estimation, a completely wrong motion estimate can result. For example if a region, such as that highlighted in Figure 2.1 straddles an object boundary, then any of the motion vectors shown may be assigned as the motion for the region. The motion estimate is most accurate along the direction of maximum image gradient and this is known as the aperture effect. If the region has image gradients in two directions, such as at a corner, then the two components of the motion can be resolved. If the region was large enough it could incorporate the whole object and there would be no problem.

2.2.2 Texture

For motion to be estimated between two regions there must be something in the regions to be matched against; there needs to be some gradient or texture information present in both regions. Consider for example a region located in the center of the moving object in Figure 2.1. This region contains no gradient and will most likely be assigned a zero motion by typical region based motion estimators. This causes a motion discontinuity, in this case a spatial discontinuity in a contiguously moving region where it should not be. The estimate for the region is different from the global motion of the object.

2.2.3 Temporal Discontinuities

Occlusion is a fundamental feature of image sequences which contain motion. It is often ignored in traditional motion estimation techniques. Consider Figure 2.2. When objects in a scene move some areas in the frame n are uncovered, which are not present in frame $n-1$. Motion vectors are only defined for areas which can be found in both frames n and $n-1$. For these areas there can be no motion vector since these areas represent some newly formed region. This is an example of the discontinuous nature of the motion field of typical sequences. The occlusion problem is not considered in this thesis. However it should be noted that it is possible to extend the methods in chapter 3 to include an occlusion prior, similar to the work of Konrad and Dubois [108] and Stiller [172, 173].

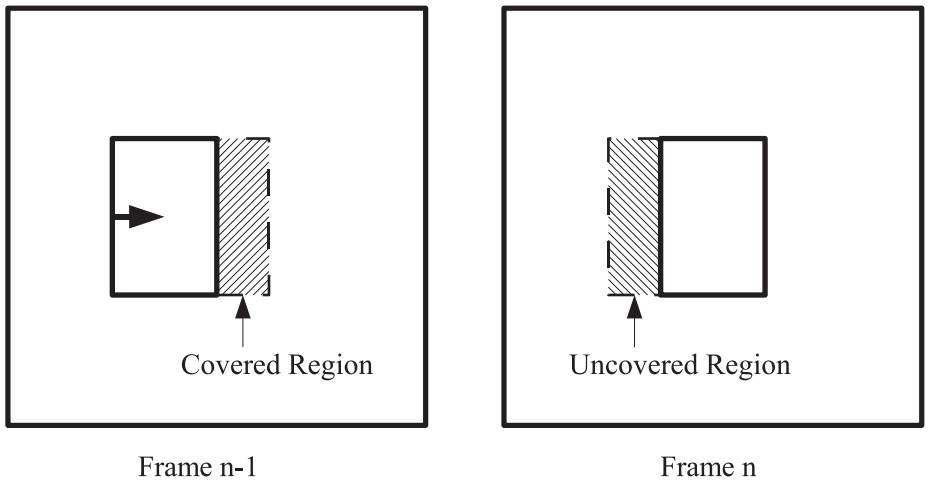


Figure 2.2: *The occlusion problem in motion estimation. Objects undergoing motion cover and uncover regions of the frame. These regions cannot be matched to areas in the previous frame. This is an example of a temporal discontinuity in the motion field.*

2.2.4 Examples

Figure 2.3 shows some examples of the problems that arise. These results were generated using a Wiener based pel-recursive algorithm outlined in section 2.4.1.2. The region highlighted at the top of the image is an example of the aperture effect. Blocks overlapping the calendar edge contain areas moving with two separate motions. Here the Smooth Local Flow is violated and hence some blocks are given the motion of the calendar and some the motion of the background. Only the motion component perpendicular to the edge is correct. The region at the bottom of the image illustrates the problem in motion estimation when there is no texture for matching. The blocks with low image gradient receive a zero motion vector even though they are part of the moving calendar. Also note how the shadow of the ball on the calendar causes incorrect motion vectors for these blocks. It is worth pointing out however that these problems only affect portions of the image. The work presented in chapter 3 aims to overcome some of these problems by exploiting the idea that most of the time the motion estimate is quite good.

2.3 Unifying Motion Estimation Algorithms

A Bayesian framework is used here to provide a common foundation upon which a whole range of motion estimation algorithms may be built, including the new approaches presented in chapters 3 and 4. The basic problem is trying to estimate the motion parameters between a pair of frames given the image data available, some or all of previously estimated motion parameters,

Figure 2.3: *Example Motion Discontinuities. (**A**) The top region shows an example of the aperture effect. Blocks overlapping the calender edge contain two motions; that of the calender and of the background. Blocks tend to be given motion perpendicular to the edge. (**B**) The bottom region in the top figure is an example of blocks that contain low gradient or little texture. The motion estimation gives zero motion for these blocks even though they are part of the moving calender.*

and possibly some other data as necessary. In probabilistic terms this may be expressed as

$$p(\text{Motion Parameters}|\text{Image Data}, \text{Previous Motion Parameters}, \text{Other Data})$$

Bayesian estimation is a technique for identifying model parameters from noisy observations of a given process, given certain prior information about those parameters [165]. The work which is presented later in this thesis is based on the following assumptions. Given a current estimate of the motion field, $D(\mathbf{x})$ which may be zero, the problem is to manipulate a probability distribution function similar to the following

$$p(\mathbf{d}_{n,n-1}(\mathbf{x}), \mathbf{d}^{\mathbf{g}}, \mathbf{A}^{\mathbf{g}}|I_n, I_{n-1}, D(\mathbf{x}), \mathbf{d}^{\mathbf{g}}_{n-1}, \mathbf{A}^{\mathbf{g}}_{n-1}) \tag{2.6}$$

That is to find the local translational motion, global translational motion, and global affine parameters given the image data, some neighbourhood (spatial and/or temporal) of motion vectors for the current position, and previous estimates of the global motion and affine parameters. Using a Bayesian framework similar to work by Stiller [171], Konrad and Dubois [108], and Kokaram et al. [101, 103], this may be written as

$$p(\mathbf{d}_{n,n-1}, G_n|I_n, I_{n-1}, D(\mathbf{x}), G_{n-1}) \propto \underbrace{p(I_n|I_{n-1}, \mathbf{d}_{n,n-1}, G_n)}_{likelihood} \underbrace{p(\mathbf{d}_{n,n-1}|D(\mathbf{x}))p(G_n|G_{n-1})}_{priors} \tag{2.7}$$

where the global motion parameters have been combined together in the variable G. The right hand side of this expression contains likelihood and prior distributions where:

- The *likelihood* ensures that the frames, I_n and I_{n-1}, correspond using the estimated motion parameters Θ_n. It connects the motion information to the observed image data through the image sequence model (2.1).
- The *priors* can be used to insert information about the expected organisation of the motion field and to provide constraints on the estimation process.

Often motion estimation is concerned with finding the parameters which maximise the posterior distribution (2.7). This is known as Maximum a-posteriori (MAP) estimation. If no prior information is incorporated, this leads to Maximum likelihood estimation. The following sections describe the likelihood and prior in more detail.

2.3.1 Likelihood

The likelihood should discourage motion estimates that are not supported by the observed data. It is possible to incorporate uncertainty into the image sequence model by writing (2.1) as

$$I_n(\mathbf{x}) = I_{n-1}(F(\mathbf{x}, \Theta_n)) + e_n(\mathbf{x}) \tag{2.8}$$

where e_n represents any error in the model and is a combination of noise in the image formation process and errors in the motion model of the underlying signal. If e_n is assumed to be Gaussian [108], $e_n \sim \mathcal{N}(0, \sigma_e^2)$, then it follows that the likelihood expression required in the assembly of the posterior may be expressed as

$$p(I_n(\mathbf{x})|I_{n-1}(\mathbf{x}), \Theta_n, \sigma_e^2) = \left(\frac{1}{\sqrt{2\pi\sigma_e^2}}\right)^N \exp\left(-\frac{\sum_{\mathbf{x}}(I_n(\mathbf{x}) - I_{n-1}(F(\mathbf{x}, \Theta_n)))^2}{2\sigma_e^2}\right) \tag{2.9}$$

2.3.2 Priors

This probability distribution can be used to inject constraints on the motion field such as motion smoothness. To encode the notion of local smoothness of the motion field as recognised by Horn [76], Nagel [143], Enkelmann [50], Konrad and Dubois [108] and others, a typical prior distribution for motion is as follows

$$p(\mathbf{d}_{n,n-1}(\mathbf{x})|S_{n,n-1}(\mathbf{x})) \propto \exp\left(-\sum_{\mathbf{v}\in S_{n,n-1}(\mathbf{x})} \lambda(\mathbf{v})[\mathbf{d}_{n,n-1}(\mathbf{x}) - \mathbf{v}]^2\right) \tag{2.10}$$

where $\mathbf{v}$ is each vector in some spatial neighbourhood represented by $S_{n,n-1}(\mathbf{x})$, and $\lambda(\mathbf{v})$ is the weight associated with each clique or pair of neighbourhood interactions. This energy function penalises vectors that deviate substantially from the surrounding neighbourhood. The prior term may also be used to model other motion discontinuities as in Kokaram [101], Konrad and Dubois [108] and Stiller [172, 173] for example.

2.4 Local Motion Estimation

As outlined in section 2.1.1, the model for local motion is purely translational. Following (2.5), the image sequence model considered here (2.8) may be written as

$$I_n(\mathbf{x}) = I_{n-1}(\mathbf{x} + \mathbf{d}_{n,n-1}(\mathbf{x})) + e_n(\mathbf{x}) \tag{2.11}$$

The only parameter that is needed for this model is the motion vector $\mathbf{d}_{n,n-1}(\mathbf{x})$, which is the displacement vector mapping the region at position $\mathbf{x}$ in the current frame n into some region in the previous frame $n-1$.

It is possible to classify motion estimation algorithms based on how they manipulate (2.6) to obtain $\mathbf{d}$. Methods may be further categorised those which include prior information on the motion field and those which do not. This becomes straightforward under a Bayesian framework as outlined earlier, leading to *maximum likelihood* and *maximum-a-posteriori* methods. Implementations based on these two general methodologies are considered next.

2.4.1 Maximum Likelihood Methods

In motion estimation algorithms where no prior knowledge about the motion field is incorporated into the solution, the prior distribution in the posterior is a constant. Hence the posterior

distribution is only influenced by the likelihood. Assuming that the observation noise is white, $e_n \sim \mathcal{N}(0, \sigma_e^2)$, the likelihood may be modelled as

$$p(I_n(\mathbf{x})|I_{n-1}(\mathbf{x}), \mathbf{d}_{n,n-1}(\mathbf{x}), \sigma_e^2) \propto \exp\left(-\frac{\sum_{\mathbf{x}}(I_n(\mathbf{x}) - I_{n-1}(\mathbf{x} + \mathbf{d}_{n,n-1}(\mathbf{x})))^2}{2\sigma_e^2}\right) \tag{2.12}$$

Typically the goal is to find the best vector at each site $\mathbf{x}$. This implies maximising (2.12) with respect to $\mathbf{d}_{n,n-1}$. There is no need to evaluate the normalised probability since maximising that expression is the same as minimising the log probability. Also no knowledge of σ_e^2 is required since it is a constant. It can be useful here to define the Displaced Frame Difference (DFD) as

$$DFD(\mathbf{x}, \mathbf{d}) = I_n(\mathbf{x}) - I_{n-1}(\mathbf{x} + \mathbf{d}) \tag{2.13}$$

In general, greater clarity is achieved by classifying this error measure as the Displaced Region Difference (DRD) for local motion estimation since the DFD is used with respect to small regions only, rather than the whole frame or a single site. However for convenience only the DFD notation is used, while noting that for local motion estimation the DFD is calculated only over the region under consideration, e.g. a block. For global motion estimation algorithms, where the region is the whole frame, then the DFD is calculated over the whole frame.

Substituting (2.13) into (2.12) and using negative log probabilities, the problem then becomes a matter of estimating at each site $\mathbf{x}$ the motion vector $\mathbf{d}$ which minimises some function of the DFD

$$\mathbf{d}_{n,n-1}(\mathbf{x}) = \arg\min_{\mathbf{d}}(\|DFD(\mathbf{x}, \mathbf{d})\|^2) \tag{2.14}$$

There are three main types of method which may be grouped under this maximum likelihood approach

- Correspondence Matching.
- Gradient Based Methods.
- Transform Domain Methods.

2.4.1.1 Correspondence Matching

One approach to solving (2.14) is an exhaustive search method. This implies trying all possible motion vectors, within some defined limits, and choosing the best one at each site. Because estimating motion for a single pixel site is an ill-posed problem, the image is usually divided into distinct regions and motion vectors are estimated for each region [85]. A typical approach is to the divide the image into non-overlapping rectangular blocks and assume that the motion within a block is constant. Two correspondence matching techniques, Block matching and integral projections, are discussed next.

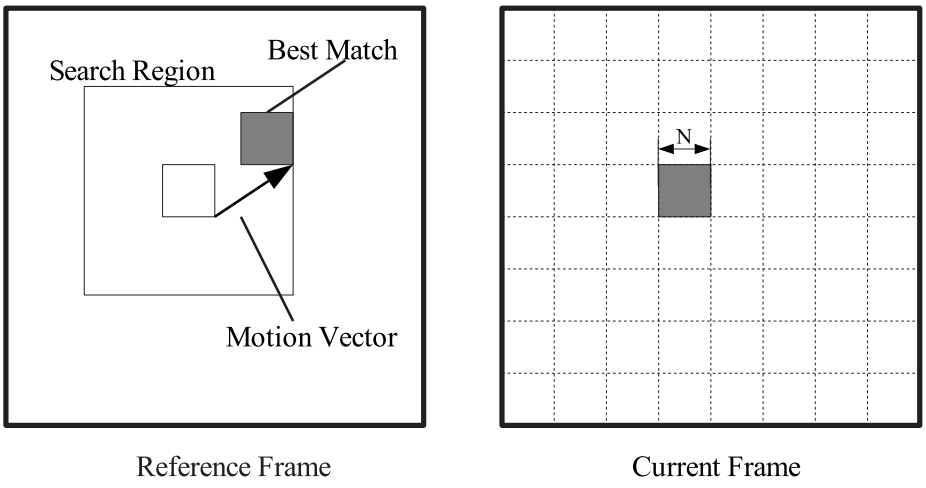


Figure 2.4: *Full Search Block Matching*

Block Matching

Block Matching (BM) is a widely used and robust technique for motion estimation. The image in frame n, is divided into blocks of size, $M \times N$ pixels[2]. Motion vectors are then assigned to each block in turn. The motion vectors are found by matching the block in frame n with a set of blocks, the same size as the current block, in some search space in frame $n-1$. The best matching block is usually determined by considering one of Mean Squared Error (MSE), Mean Absolute Difference (MAD), or Sum Absolute Difference (SAD) of the pixel intensities between the two blocks. The best match is the one which gives the smallest error.

$$MSE(\mathbf{x},\mathbf{d}) = \frac{1}{MN}\sum_{\mathbf{x}\in Block}(DFD(\mathbf{x},\mathbf{d}))^2 \tag{2.15}$$

$$MAD(\mathbf{x},\mathbf{d}) = \frac{1}{MN}\sum_{\mathbf{x}\in Block}|DFD(\mathbf{x},\mathbf{d})| \tag{2.16}$$

$$SAD(\mathbf{x},\mathbf{d}) = \sum_{\mathbf{x}\in Block}|DFD(\mathbf{x},\mathbf{d})| \tag{2.17}$$

If the assumptions about constant intensity over a block hold, the correct motion vector will give a zero DFD. The maximum displacement that (BM) can estimate is limited only by the size of the search space used. This search space is usually defined as $\pm w$ pixels for both the horizontal and vertical components. This leads to a search space of area $(2w+M)\times(2w+N)$ pixels, see Figure 2.4.

In a Full Motion Search there are $(2w+1)^2$ searched locations, for an integer accurate motion estimate. However, this is computationally demanding given that it requires on the order of $N^2(2w+1)^2$ operations per block. There are other block matching techniques which

[2]Typically square block sizes are used, $M == N$, which is the case assumed here unless noted otherwise

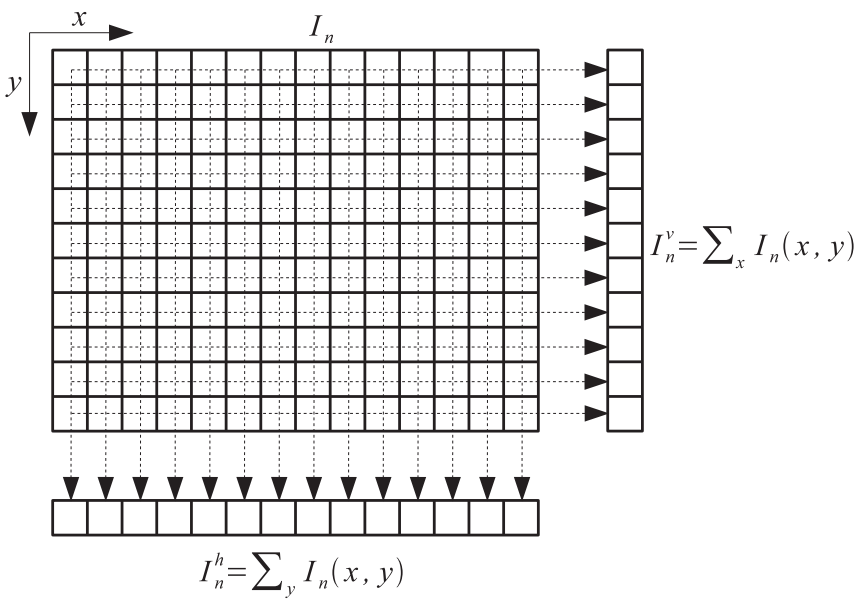


Figure 2.5: *Computing Integral Projections*

reduce the burden of the Full Motion Search such as the Three Step Search [100] or The Cross Search [64]. These algorithms usually reduce the number or locations searched, by limiting the search space to some fixed pattern, or reduce the number of pixels sampled in each block [203]. The Successive Elimination Algorithm [118, 21] uses the SAD and excludes many search positions from calculating the matching criterion by evaluating lower bounds for this error measure. While these reduced search techniques can offer large savings in computation, often accuracy is decreased as they may find local minima in the DFD function and yield wrong motion vectors.

In general the BM algorithm is quite a robust motion estimator since any noise tends to be averaged out over the block operations. It can also estimate large displacements since its search space is only limited by the user and can in fact cover the whole frame if needed. It has a simple control structure and hence is a simple algorithm to implement. Thus it was the first to be included in hardware for real-time video coding applications such as video compression for digital TV.

There are disadvantages to the BM algorithm, namely the heavy computation involved and the motion averaging effect of the blocks. If the blocks chosen are too big then there may be smaller objects moving within the block allocated the same motion, which is unlikely to match the motion of any of the objects. Block matching also suffers the problems due to the aperture effect.

Integral Projections

An alternative correspondence matching technique based on integral projections was first proposed by Lee and Park [113] and since expanded upon by many, including [99, 133, 114, 32]. Integral projections are defined to be the summation of the grey levels of the pixels along any fixed direction in the image. Vertical integral projections, I_n^v, are obtained by summing the grey levels in the vertical direction to produce a one dimensional vector; similarly for the horizontal projections, I_n^h. This is shown in Figure 2.5.

$$I_n^v(y) = \sum_x I_n(x, y) \tag{2.18}$$

$$I_n^h(x) = \sum_y I_n(x, y) \tag{2.19}$$

Instead of matching images or blocks of pixels, their respective horizontal and vertical integral projections are matched instead. Interestingly, operations involving just the integral projections yield similar results to operating on the full blocks. Furthermore the two components of the motion may be decoupled from each other and estimated independently. This leads to reduced computation as the matching cost now takes an order $O(M + N)$ operations instead of the $O(M \times N)$ needed for block matching. An alternative form of motion estimation known as gradient based methods are covered next.

2.4.1.2 Gradient Based Methods

Gradient based motion estimation arises from a different look at the image sequence model in (2.11). There are many different techniques using a gradient based approach but they all rearrange the model equation to have direct access to the motion parameter $\mathbf{d}$. Among the first to do this were Cafforio and Rocca [25]. By taking a Taylor series expansion of (2.11), it may be linearized about $\mathbf{d}$ to give

$$I_n(\mathbf{x}) = I_{n-1}(\mathbf{x}) + \mathbf{d}_{n,n-1}^T \nabla I_{n-1}(\mathbf{x}) + e_n(\mathbf{x}) \tag{2.20}$$

where ∇ is the multidimensional gradient operator[3] and $e_n(\mathbf{x})$ represents the higher order terms of the expansion and errors in the model lumped together. This equation can be rearranged to give an expression involving the Displaced Frame Difference with zero displacement.

$$\begin{aligned} DFD(\mathbf{x}, 0) &= I_n(\mathbf{x}) - I_{n-1}(\mathbf{x}) \\ &= \mathbf{d}_{n,n-1}^T \nabla I_{n-1}(\mathbf{x}) + e_n(\mathbf{x}) \end{aligned} \tag{2.21}$$

Equation (2.21) has 2 unknowns, so in order to solve it at least one more equation is necessary. Following the Smooth Local Flow assumption, where it is assumed that within a small region all pixels behave the same with respect to motion, several additional equations can be set up.

[3] $\nabla I_{n-1}(\mathbf{x}) = [\frac{\partial I_{n-1}(\mathbf{x})}{\partial x} \ \frac{\partial I_{n-1}(\mathbf{x})}{\partial y}]$, hence the name gradient based motion estimator.

Neglecting the higher order terms allows equations to be set up at each pixel site in some defined region, thus yielding the vector equation

$$\mathbf{z} = \mathbf{G}\mathbf{d}_{n,n-1} \tag{2.22}$$

where a solution for $\mathbf{d}$ can be generated using a pseudo inverse approach.

$$\mathbf{d}_{n,n-1} = [\mathbf{G}^T\mathbf{G}]^{-1}\mathbf{G}^T\mathbf{z} \tag{2.23}$$

This solution yields $\mathbf{d}$ in one step by using gradient measurements in the defined region. This region is typically a rectangular block of pixels. $\mathbf{G}$ and $\mathbf{z}$ are matrices, consisting of the gradient measurements and residuals respectively, for the region.

Recursive Methods

One important drawback with this gradient based approach is that the Taylor series expansion is only valid over very small distances. To overcome this problem an iterative solution known as the pel-recursive approach was proposed by Netravali and Robbins [145]. Linearising the image model about a current guess for $\mathbf{d}$, say $\mathbf{d}_i$, an update $\mathbf{u}_i$ is generated for this current estimate that will eventually force the estimation process to converge on the correct motion vector. This update is intended to be a small one to maintain the validity of the Taylor series expansion. Equation 2.20 now becomes

$$\begin{aligned} \mathbf{u}_i &= \mathbf{d}_{n,n-1} - \mathbf{d}_i \\ I_n(\mathbf{x}) &= I_{n-1}(\mathbf{x} + \mathbf{d}_i) + \mathbf{u}_i^T \nabla I_{n-1}(\mathbf{x} + \mathbf{d}_i) + e_n(\mathbf{x} + \mathbf{d}_i) \end{aligned} \tag{2.24}$$

Work by Walker and Rao [192] and Biemond et al. [10] improved upon the technique introduced by Netravali and Robbins. In [10], Biemond et al. presented a Wiener solution for the displacement which is more robust to noise. It considers the effect of the higher order terms on the solution as being the same as Gaussian white noise. This estimator is called the Wiener based motion estimator (WBME). The new vector equation becomes

$$\mathbf{z}_i = \mathbf{G}\mathbf{u}_i + \mathbf{e} \tag{2.25}$$

and the Wiener solution for the update $\mathbf{u}_i$ of the current estimate $\mathbf{d}_i$ is

$$\begin{aligned} \mathbf{u}_i &= [\mathbf{G}^T\mathbf{G} + \mu\mathbf{I}]^{-1}\mathbf{G}^T\mathbf{z} \\ \mu &= \frac{\sigma_{ee}^2}{\sigma_{uu}^2} \end{aligned} \tag{2.26}$$

Here, σ_{uu}^2 is the variance of the estimate for $\mathbf{u}_i$ and σ_{ee}^2 is the variance of the Gaussian white noise representing the higher order terms in (2.24). After the update is estimated, $\mathbf{d}_i$ is refined to yield the displacement that is used in the next iteration, $\mathbf{d}_{i+1} = \mathbf{d}_i + \mathbf{u}_i$.

Note that the WBME operates on a block basis. Hence the vector field generated has a block resolution. Typical block sizes are 8×8 and 16×16.

Adaptive Recursion

The μ in (2.26) acts as a damper on the system, preventing the inverse of the matrix $M_g = \mathbf{G}^T\mathbf{G}$ from becoming unstable. Errors in the WBME can be linked directly to the extent of ill-conditioning in the matrix M_g, which in turn is linked to the aperture effect. This ill-conditioning can be measured by the ratio of the eigenvalues of the matrix. Martinez [128] used an SVD decomposition of M_g in a motion estimator which was not pel-recursive. Many authors have concentrated on adapting μ in the WBME to the ill-conditioning of M_g during the pel-recursive process, including Efstratiadis and Katsagellos [48], Driessen et al. [43, 44], and Böröczky et al. [17].

The algorithm of Kokaram [107, 101], which is used for comparison in later chapters, combines the approaches of Driessen et al. and Martinez.

$$\begin{aligned} \mathbf{u}_i &= \begin{cases} \alpha_{max}\mathbf{e}_{max} & \text{if } \frac{\lambda_{max}}{\lambda_{min}} > \alpha \\ [\mathbf{G}^T\mathbf{G} + \mu\mathbf{I}]^{-1}\mathbf{G}^T\mathbf{z} & \text{otherwise} \end{cases} \\ \mu &= |\mathbf{z}|\frac{\lambda_{max}}{\lambda_{min}} \\ \alpha_{max} &= \frac{\mathbf{e}_{max}^T}{\lambda_{max}}\mathbf{G}^T\mathbf{z} \end{aligned} \tag{2.27}$$

Here, λ and $\mathbf{e}$ refer to the eigenvalues and eigenvectors of $\mathbf{G}^T\mathbf{G}$, and α_{max} is a scalar variable used to simplify the final expression. If the ratio of eigenvalues is greater than some threshold α, indicating that matrix M_g is ill-conditioned, the solution of Martinez is used. Otherwise the solution continues as normal with the damping parameter μ following the approach of Driessen et al.

2.4.1.3 Transform Domain Methods

The final class of motion estimation algorithms examined here are often know as Transform Domain methods. These methods aim to estimate the displacement directly, as opposed to indirect methods such as block matching and pel-recursive methods. Transform Domain methods refer here to methods which transform the images into another domain in which $\mathbf{d}$ is more directly accessibly. This alternative domain is often the frequency or Fourier domain. One example of this, called the phase-correlation method, is a spatial frequency domain technique that employs the Fourier transform. Other transforms such as the complex wavelet transform (CWT) have also been proposed [125].

Phase-Correlation

The phase-correlation method estimates the relative shift between two image blocks by means of a normalized cross-correlation function computed in the 2D spatial Fourier domain. The

cross-correlation can be efficiently estimated using the discrete Fourier transform (DFT) [152]. Phase-correlation is based on the evaluation of the phase of the cross-power spectrum between two frames. It exploits the fact that a relative shift in the spatial domain results in a linear phase term in the Fourier domain [183, pp. 99–100]. It was first introduced as a method for astronomical image registration by Kuglin and Hines [110]. Jain and Jain [84] divided radar images into blocks and used fast correlation to estimate the motion of each block. It has also been successfully used for television studio field rate conversion, Thomas [184, 185], and in industrial video restoration hardware [168]. The basis of the phase-correlation method is derived as follows.

Let I_n and I_{n-1} be two frames that differ only by a displacement $\mathbf{d} = [d_x, d_y]$ such that

$$I_n(x, y) = I_{n-1}(x + d_x, y + d_y) \tag{2.28}$$

Their corresponding Fourier transforms, F_n and F_{n-1}, will be related by

$$F_n(u, v) = e^{j2\pi(ud_x+vd_y)} F_{n-1}(u, v) \tag{2.29}$$

The cross correlation function between the two frames is defined as

$$c_{n,n-1}(x, y) = I_n(x, y) \circledast I_{n-1}(x, y) \tag{2.30}$$

where $\circledast$ denotes the 2-D convolution operation. Taking the Fourier transform of both sides yields the complex-valued cross-power spectrum expression

$$C_{n,n-1}(u, v) = F_n(u, v)F^*_{n-1}(u, v) \tag{2.31}$$

where F^* is the complex conjugate of F. Normalising $C_{n,n-1}(u, v)$ by its magnitude gives the phase of the CPS

$$\begin{aligned} \tilde{C}_{n,n-1}(u, v) &= \frac{F_n(u, v)F^*_{n-1}(u, v)}{|F_n(u, v)F^*_{n-1}(u, v)|} \\ &= e^{-j2\pi(ud_x+vd_y)} \end{aligned} \tag{2.32}$$

The inverse Fourier transform of $\tilde{C}_{n,n-1}(u, v)$ yields the phase-correlation function

$$\tilde{c}_{n,n-1}(x, y) = \delta(x - d_x, y - d_y) \tag{2.33}$$

which consists of an impulse centered on the location $[d_x, d_y]$, the required displacement. Hence the phase correlation method involves evaluating (2.32) and taking its inverse Fourier transform to yield a phase correlation surface which is used to find the motion estimate. An example phase correlation surface is shown in Figure 2.7 (f).

2.4.2 Maximum-a-Posteriori Methods

Moving on from maximum likelihood methods, maximum-a-posteriori (MAP) methods insert one or more priors on the estimation process. The MAP formulation requires at least two

probability distribution models: the conditional probability of the observed image intensity given the motion field, called the likelihood model or the observation model, and the a-priori probability of the motion vectors, called the motion field model. Using Bayes theorem, the a-posteriori probability of the motion field given the two frames may be expressed as

$$p(\mathbf{d}|I_n, I_{n-1}) = \frac{p(I_n|I_{n-1}, \mathbf{d})p(\mathbf{d})}{p(I_n|I_{n-1})} \tag{2.34}$$

Since the denominator is not a function of $\mathbf{d}$, it is a constant for the purposes of motion estimation and can be ignored. As the name suggests, MAP methods aim to find the motion field which maximises the posterior distribution. The MAP estimate $\hat{\mathbf{d}}$, is then be given by

$$\hat{\mathbf{d}} = \arg\max_{\mathbf{d}}(p(I_n|I_{n-1}, \mathbf{d})p(\mathbf{d})) \tag{2.35}$$

The posterior consists of the product of a likelihood and prior. These may, using negative log probabilities, be expressed as energy functions L and V respectively to yield

$$E(\mathbf{d}) = L(\mathbf{d}) + V(\mathbf{d}) \tag{2.36}$$

The problem then becomes one of energy minimisation

$$\hat{\mathbf{d}} = \arg\min_{\mathbf{d}} E(\mathbf{d}) \tag{2.37}$$

In this way it can be seen that including prior information in the estimate merely involves adding an extra term to the energy function, E. The form of these energy functions depends on the particular probability distribution model. For example, taking the likelihood model in (2.9), L would be expressed as

$$L(\mathbf{d}) = \frac{\sum_{\mathbf{x}}(I_n(\mathbf{x}) - I_{n-1}(\mathbf{x}+\mathbf{d}))^2}{2\sigma_e^2} \tag{2.38}$$

The prior term can be used to put constraints on the motion field. A typical prior used to encourage a smooth vector field was given in (2.10). This leads to an energy function of the following form

$$V(\mathbf{d}) = -\sum_{\mathbf{d}_s \in S_{n,n-1}(\mathbf{x})} \lambda(\mathbf{d}_s)[\mathbf{d}_{n,n-1}(\mathbf{x}) - \mathbf{d}_s]^2 \tag{2.39}$$

where $\mathbf{d}_s$ is each vector in some spatial neighbourhood represented by $S_{n,n-1}(\mathbf{x})$, and $\lambda(\mathbf{d}_s)$ is some weight associated with each pair of neighbourhood interactions. Examples of MAP estimation methods are discussed next.

2.4.2.1 Optical Flow

Optical flow methods are based on the constant-intensity assumption. This states that the image intensity remains constant along a motion trajectory. This assumption implies, that

among others, any intensity change is due to motion, and that scene illumination is constant. The constant-intensity assumption translates into the following constraint equation

$$\frac{dI(\mathbf{x})}{dt} = 0 \tag{2.40}$$

By applying the chain rule for differentiation, (2.40) can be re-expressed as

$$\frac{\partial I(\mathbf{x})}{\partial x}\frac{\partial x}{dt} + \frac{\partial I(\mathbf{x})}{\partial y}\frac{\partial y}{dt} + \frac{\partial I(\mathbf{x})}{dt} = 0 \tag{2.41}$$

where the partial derivatives $\frac{\partial x}{dt}, \frac{\partial y}{dt}$ are the horizontal and vertical velocity components at site $\mathbf{x} = (x, y)^T$ respectively. This is the well known Optical Flow equation as proposed by Horn and Schnuck [76], and can be expressed as

$$(\nabla I(\mathbf{x}))^T \mathbf{d}(\mathbf{x}) + \frac{\partial I(\mathbf{x})}{dt} = 0 \tag{2.42}$$

where ∇ denotes the spatial gradient operator, $\mathbf{d}(\mathbf{x})$ is the motion vector, and the temporal derivative $\frac{\partial I(\mathbf{x})}{dt}$ is the instantaneous DFD at pixel site $\mathbf{x}$. Note that this equation can also be derived from the Taylor Series expansion in (2.21) by assuming the error in the expansion is zero. Hence the Optic Flow equation is valid only for small $\mathbf{d}$. Also, since at a single site equation (2.42) is ill-posed due to the aperture effect, estimating motion on a pixel basis requires additional constraints on the motion field.

This can be done by putting a weak constraint on the estimate itself, reflecting the empirical observation that typical motion fields are spatially smooth [174]. This smoothness constraint is in effect a prior on the estimation turning it into a MAP estimation problem. Horn and Schunck [76] penalized the squared error resulting from the Optic Flow constraint (2.42) by adding a smoothness term. They proposed that the gradient of the motion field can be used as a measure of smoothness. This smoothness term states that in a local region the gradient of the motion field should be low. Using the 2D derivative operator, and letting $\mathbf{d}(\mathbf{x}) = [d_1(\mathbf{x}), d_2(\mathbf{x})]^T$, the motion gradient is given by

$$\begin{aligned} \|\nabla \mathbf{d}(\mathbf{x})\| &= \frac{\partial \mathbf{d}}{\partial x} + \frac{\partial \mathbf{d}}{\partial y} \\ &= \left(\frac{\partial d_1}{\partial x}\right)^2 + \left(\frac{\partial d_1}{\partial y}\right)^2 + \left(\frac{\partial d_2}{\partial x}\right)^2 + \left(\frac{\partial d_2}{\partial y}\right)^2 \end{aligned} \tag{2.43}$$

This results in the following energy function at a site $\mathbf{x}$

$$E(\mathbf{d}(\mathbf{x})) = \left((\nabla I(\mathbf{x}))^T \mathbf{d}(\mathbf{x}) + \frac{\partial I(\mathbf{x})}{dt}\right)^2 + \lambda \left(\|\nabla \mathbf{d}(\mathbf{x})\|\right) \tag{2.44}$$

which is in the same form as (2.36) for MAP estimation with

$$L(\mathbf{d}(\mathbf{x})) = \left((\nabla I(\mathbf{x}))^T \mathbf{d}(\mathbf{x}) + \frac{\partial I(\mathbf{x})}{dt}\right)^2 \tag{2.45}$$

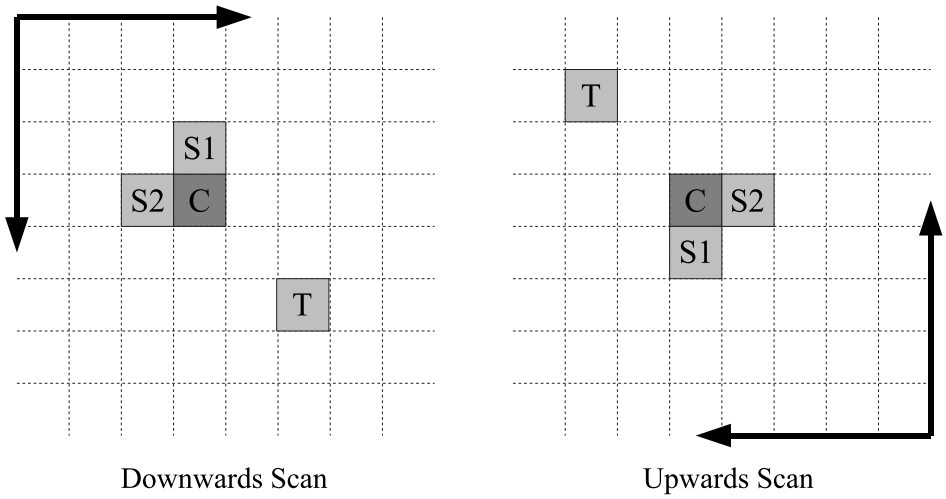


Figure 2.6: *3DRS candidate set configuration. C is the current block. S1, S2, and T indicate the relative position of the two spatial and one temporal candidates respectively. Update candidates are generated using S1 and S2. The arrows indicate the scanning direction.*

and

$$V(\mathbf{d}(\mathbf{x})) = \lambda \left(\|\nabla \mathbf{d}(\mathbf{x})\| \right) \tag{2.46}$$

The variable λ represents the degree of smoothness in the motion field. Increasing λ implies very smooth motion, whereas decreasing it implies very discontinuous motion. It is difficult to estimate λ from the sequence itself, a process called *hyperparameter* estimation. Typically, it is a user defined parameter used to encourage a certain level of smoothness in the estimated motion field.

2.4.2.2 3D Recursive Search

In [36] de Haan et al. presented a motion estimator they termed 3D Recursive Search Block Matching (3DRS). It has been used commercially for real-time scan-rate conversion in consumer televisions. This method employs an implicit temporal and spatial prior. The 3DRS algorithm limits its search to a small set of candidate vectors based on spatio-temporal predictions. It is similar to block matching with the exception that instead of testing all vectors in a certain search range, it limits the number of vectors to those already estimated, reducing the risk of reaching a local minimum in the error function. The algorithm used for comparison in later chapters is described next, and is as outlined by Braspenning and de Haan in [20].

The candidate set, $\mathbf{c}_i \in CS$, consists of spatial and temporal prediction candidates. Spatial candidates are chosen from neighbouring blocks that have been estimated in the current picture, $\mathbf{c}_i = \mathbf{d}_{n,n-1}(\mathbf{x} + \mathbf{b}_i)$, where $\mathbf{b}_i$ indicates the relative position with respect to the current block $\mathbf{x}$. The temporal candidates are given by $\mathbf{c}_i = \mathbf{d}_{n-1,n-2}(\mathbf{x} + \mathbf{b}_i)$. The relative position of the

prediction vectors is shown in Figure 2.6.

To ensure convergence of the algorithm and to correctly track variable object motion, update candidates are added to the candidate set by adding small random vectors ($\mathbf{u}$) to the spatial candidates, i.e. $\mathbf{c}_i = \mathbf{c}_j + \mathbf{u}, j \neq i$. In practice the update vectors are drawn cyclically from a limited update set, US. This update set can include fractional motion vectors to provide sub pixel accurate motion estimation [37].

$$\begin{aligned} US = \Bigg\{ &\begin{bmatrix} 1/4 \\ 0 \end{bmatrix}, \begin{bmatrix} 1/2 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 2 \\ 0 \end{bmatrix}, -\begin{bmatrix} 0 \\ 1/4 \end{bmatrix}, -\begin{bmatrix} 0 \\ 1/2 \end{bmatrix}, -\begin{bmatrix} 0 \\ 1 \end{bmatrix}, -\begin{bmatrix} 0 \\ 2 \end{bmatrix}, \\ &-\begin{bmatrix} 1/4 \\ 0 \end{bmatrix}, -\begin{bmatrix} 1/2 \\ 0 \end{bmatrix}, -\begin{bmatrix} 1 \\ 0 \end{bmatrix}, -\begin{bmatrix} 2 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1/4 \end{bmatrix}, \begin{bmatrix} 0 \\ 1/2 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 0 \\ 2 \end{bmatrix} \Bigg\} \end{aligned} \tag{2.47}$$

The candidate vectors $\mathbf{c}_i$ of the candidate set CS are then constructed as follows

$$\mathbf{c}_i = \begin{cases} \mathbf{d}_{n,n-1}(\mathbf{x} + \mathbf{b}_i) & \text{if } \mathbf{c}_i \text{ is a spatial candidate} \\ \mathbf{d}_{n-1,n-2}(\mathbf{x} + \mathbf{b}_i) & \text{if } \mathbf{c}_i \text{ is a temporal candidate} \\ \mathbf{c}_j + \mathbf{u},\ j \neq i, \mathbf{u} \in US & \text{if } \mathbf{c}_i \text{ is an update candidate} \end{cases} \tag{2.48}$$

Similar to block matching each candidate is tested in turn, and the one which minimises some matching criteria (SAD here) is chosen as the vector for the block. However a small penalty is added to the SAD for a block depending on the type of candidate vector [37]. The reason being that candidate vectors that are less likely to be the true-motion vector for the current block must be less likely to be chosen in case all vectors have a similar matching error. Typical values for the penalty p_i for a block size $M \times N$ are

$$p_i(\mathbf{c}_i) = \begin{cases} 0 & \text{if } \mathbf{c}_i \text{ is a spatial candidate} \\ 1/2 \cdot M \cdot N & \text{if } \mathbf{c}_i \text{ is a temporal candidate} \\ 2 \cdot M \cdot N & \text{if } \mathbf{c}_i \text{ is an update candidate} \end{cases} \tag{2.49}$$

This penalty mechanism encourages a smooth vector field by preferring spatial candidates when the matching error for candidates is similar. The motion vector $\mathbf{d}$ assigned to the block of pixels at location $\mathbf{x}$ is found by

$$\mathbf{d}_{n,n-1}(\mathbf{x}) = \arg\min_{\mathbf{c}_i}(SAD(\mathbf{x}, \mathbf{c}_i) + p_i(\mathbf{c}_i)),\ \mathbf{c}_i \in CS. \tag{2.50}$$

Although de Haan developed this algorithm from a practical viewpoint, it can be unified into the motion estimation paradigm using the Bayesian framework as follows. Equation (2.50) is in the form of (2.36), where the minimisation is over a combination of two energies with

$$L(\mathbf{d}(\mathbf{x})) = SAD(\mathbf{x}, \mathbf{c}_i) \tag{2.51}$$

and

$$V(\mathbf{d}(\mathbf{x})) = p_i(\mathbf{c}_i) \tag{2.52}$$

In this case the posterior distribution is

$$p(\mathbf{d}_{n,n-1}|I_n, I_{n-1}, \mathbf{d}_{n-1,n-2}) \propto p(I_n|I_{n-1}, \mathbf{c}_i)p(\mathbf{c}_i) \tag{2.53}$$

and the likelihood and prior distributions may be expressed as

$$p(I_n|I_{n-1}, \mathbf{c}_i) \propto \exp(-SAD(\mathbf{x}, \mathbf{c}_i)) \tag{2.54}$$

$$p(\mathbf{c}_i) \propto \exp(-p_i(\mathbf{c}_i)) \tag{2.55}$$

The candidate set used here contains five candidate vectors per block; two spatial candidates, two update candidates, and one temporal candidate. The update set used is given by (2.47). Furthermore estimation is performed with a downwards scan and an upwards scan. In the downwards scan the blocks are processed from left to right and top to bottom, and vice-versa for the upwards scan, see Figure 2.6.

This concludes the discussion of local motion estimation algorithms. The following section outlines the global motion estimation problem and some of the common algorithms associated with it.

2.4.2.3 Other Methods

Konrad and Dubois [108] take a similar approach to that outlined here of using Bayesian methods to provide a general framework for motion estimation. They use stochastic relaxation techniques such as the Metropolis algorithm [181] and the Gibbs sampler [62] to solve the MAP estimation problem. Chapter 3 outlines two deterministic approaches to this problem using the ICM [9] and Belief Propagation [155] algorithms.

Stiller [171] modelled smoothness constraints using Gibbs/Markov random fields. Stiller also introduced the idea of using a limited candidate set of motion vectors to reduce the computational complexity of solving the MAP estimation problem. These ideas are incorporated in the work outlined in chapter 3. A robust estimation technique by Black and Anandan [14] similar to the Horn and Schnuck method, formulated the problem to allow violations of the brightness constancy and spatial coherence assumptions. This allowed for smooth motion fields with discontinuities at object boundaries.

2.5 Global Motion Estimation

A similar classification to that used for local motion estimation algorithms may be applied while reviewing global motion estimation methods. All the techniques outlined for local motion estimation may be employed with slight modification for estimating global motion parameters. The main difference is the size of the region being matched. Whereas for local motion the regions being matched are small and typically rectangular blocks, in global motion estimation the region is the whole frame. Another key difference is the need to discount the effect of local motion on

the global motion estimate. This involves identifying those areas of the picture undergoing local motion.

Recall that the vector function $F(\mathbf{x}, \Theta_n)$ represents the transformation of image coordinates caused by the motion between frames I_n and I_{n-1}. To represent motion such as zooming, rotation, and translation, a six parameter affine transformation, as proposed in section 2.1.1, is used here where $\mathbf{A}$ is a 2×2 matrix for affine transformation and $\mathbf{d}$ is the displacement vector.

$$I_n(\mathbf{x}) = I_{n-1}(F(\mathbf{x}, \Theta_n)) + \epsilon(\mathbf{x}) \tag{2.56}$$

$$\begin{aligned} F(\mathbf{x}, \Theta_n) &= \mathbf{A}\mathbf{x} + \mathbf{d} \\ &= \begin{bmatrix} a_1 & a_2 \\ a_3 & a_4 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} d_x \\ d_y \end{bmatrix} \\ &= \begin{bmatrix} a_1 x + a_2 y + d_x \\ a_3 x + a_4 y + d_y \end{bmatrix} \\ &= B(\mathbf{x})\Theta_n \end{aligned} \tag{2.57}$$

where

$$B(\mathbf{x}) = \begin{bmatrix} x & y & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x & y & 1 \end{bmatrix}. \tag{2.58}$$

Θ_n is the vector formed by the motion parameters

$$\Theta_n = [a_1, a_2, d_x, a_3, a_4, d_y]^T \tag{2.59}$$

This model is a good tradeoff between complexity and representativeness [147]. In the case of a translational only motion model, the parameter vector is given by $\Theta_n = [d_x, d_y]^T = \mathbf{d}$, and $F(\mathbf{x}, \Theta) = \mathbf{x} + \mathbf{d}$.

Following a Bayesian formulation, the posterior distribution may be written as

$$p(\Theta_n | I_n, I_{n-1}, \Theta_{n-1}) \propto p(I_n | I_{n-1}, \Theta_n) p(\Theta_n | \Theta_{n-1}) \tag{2.60}$$

As before, algorithms may be classified as those which include prior information on the motion estimate and those which do not, leading to maximum likelihood and maximum-a-posteriori methods.

2.5.1 Maximum Likelihood Methods

Maximum likelihood methods do not include any prior information on the motion model. Hence the posterior distribution is

$$p(\Theta_n | I_n, I_{n-1}, \Theta_{n-1}) \propto p(I_n | I_{n-1}, \Theta_n) \tag{2.61}$$

and the estimation is concerned only with the global motion between two consecutive frames. Assuming ϵ is Gaussian, $\epsilon \sim \mathcal{N}(0, \sigma_\epsilon^2)$, a typical expression for the likelihood is

$$p(I_n | I_{n-1}, \Theta_n) \propto \exp\left(-\frac{\sum_{\mathbf{x}} (I_n(\mathbf{x}) - I_{n-1}(F(\mathbf{x}, \Theta)))^2}{2\sigma_\epsilon^2} \right) \tag{2.62}$$

Recall that the DFD between two frames, where here the DFD is over all pixels in the image, is given by

$$DFD(\mathbf{x}, \Theta) = I_n(\mathbf{x}) - I_{n-1}(F(\mathbf{x}, \Theta)) \tag{2.63}$$

Maximum likelihood estimation algorithms involve finding the parameters Θ_n which minimise some function of the DFD. For instance a least squares solution attempts to solve

$$\hat{\Theta} = \arg\min_{\Theta}(\|DFD(\mathbf{x}, \Theta)\|^2) \tag{2.64}$$

Since the motion model (2.56) used is a global one and applies to the whole image it cannot account for local motion in the scene. Errors due to local motion can therefore be considered as outliers which may bias the estimate of the global motion parameters. These outliers can be removed from the estimation by using weights to implicitly remove the areas undergoing local motion [147, 45]. A weighted DFD can be introduced which considers only those pixels undergoing global motion

$$WDFD(\mathbf{x}, \Theta) = w(\mathbf{x})(I_n(\mathbf{x}) - I_{n-1}(F(\mathbf{x}, \Theta))) \tag{2.65}$$

$w(\mathbf{x})$ can be chosen such that it is 1 at sites undergoing global motion and 0 otherwise. Θ is then chosen to minimise this weighted DFD. Two methods for estimating the weights are discussed next.

2.5.1.1 Robust Methods

Odobez and Bouthemy [147] use a robust estimation technique to deal with outliers in the estimation process. Weights are used to remove those areas undergoing local motion from the estimation process. In [147], these weights are calculated according to

$$w(\mathbf{x}) = \frac{\rho(\epsilon(\mathbf{x}))}{\epsilon(\mathbf{x})} \tag{2.66}$$

The function ρ is called an M-estimator. In this case Tukey's biweight function for ρ is used [77]. Iteratively Re-weighted Least Squares (IRLS) is a well known method to solve the M-estimation problem [74]. IRLS applied to global motion parameter estimation is outlined next.

Dufaux and Konrad [45] take a similar approach except they parameterise the camera motion using an eight parameter perspective model. To deal with outliers a truncated quadratic function is used instead of the standard quadratic function. The truncation threshold is found using a robust histogram technique. A histogram of the $|DFD|$ across all pixels is computed and a threshold t is chosen so as to exclude the samples representing the top $T\%$ of the distribution. In practice this involves using weights to remove those pixels for which the absolute value of the error term is greater than t. These binary weights are given by

$$w(\mathbf{x}) = \begin{cases} 0, & \text{if } \epsilon(\mathbf{x}) \geq t \\ 1, & \text{otherwise} \end{cases} \tag{2.67}$$

They then minimise the error by using a gradient descent algorithm [159] applied over a multiresolution pyramid of input images.

Iteratively Re-weighted Least Squares Estimation

A Taylor series expansion around an initial guess for the motion parameters, Θ_0 say, such that $\Theta = \Theta_0 + \mathbf{u}$, results in

$$\begin{aligned} WDFD(\mathbf{x}, \Theta) =& w(\mathbf{x})(I_n(\mathbf{x}) - I_{n-1}(B(\mathbf{x})\Theta_0) \\ & -\nabla I_{n-1}(B(\mathbf{x})\Theta_0) \cdot B(\mathbf{x}) \cdot \mathbf{u} + \acute{\epsilon}(\mathbf{x})) \end{aligned} \tag{2.68}$$

The ∇ operator is the multidimensional gradient operator. $\acute{\epsilon}(\mathbf{x})$ represents the higher order terms of the expansion and $\epsilon(\mathbf{x})$ lumped together as a Gaussian random variable with variance σ_e^2. The update term $\mathbf{u}$ is intended to be small for the Taylor expansion to be valid. A multiresolution pyramid is used to estimate the motion using coarse to fine refinement. At each resolution level in the pyramid Θ is estimated iteratively until some convergence criteria have been met. Multiresolution schemes are discussed in more detail in section 2.6.

At each iteration i, Least Squares estimation is performed such that

$$\begin{aligned} \hat{\mathbf{u}} &= [G^T W^i G]^{-1} G^T W^i \mathbf{z} \\ \Theta^{i+1} &= \Theta^i + \hat{\mathbf{u}} \end{aligned} \tag{2.69}$$

with $\mathbf{z}$ the vector collecting the residual values $\{I_n(\mathbf{x}) - I_{n-1}(B(\mathbf{x})\Theta^i)\}_{\mathbf{x}}$ and G the matrix collecting the gradient values $\{\nabla I_{n-1}(B(\mathbf{x})\Theta^i) \cdot B(\mathbf{x})\}_{\mathbf{x}}$. W^i is the matrix collecting the weights $w(\mathbf{x})$ over the entire image.

Implicit in this scheme is a segmentation of the image into areas undergoing global motion and those undergoing local motion. The problem is now to estimate this global/local segmentation *and* to estimate Θ where $w(\mathbf{x}) = 1$. This problem is solved iteratively by alternating between estimates of W and Θ. At each step Θ^{i+1} is calculated given the previous estimates (Θ^i, W^i) according to (2.69). Then W^{i+1} is estimated given Θ^{i+1}.

2.5.1.2 Wiener Based Estimation

A Wiener based solution for (2.68) was introduced by Kokaram and Delacourt [105]. It modifies the update step to be

$$\begin{aligned} \hat{\mathbf{u}} &= [G^T W^i G + \mu_w I]^{-1} G^T W^i \mathbf{z} \\ \text{with} \quad \mu_w &= \left(\frac{\sigma_{ee}^2}{\sigma_{uu}^2} \right) \end{aligned} \tag{2.70}$$

where σ_{uu}^2 is the variance of the estimate for the $\hat{\mathbf{u}}$ and σ_{ee}^2 is the variance of the residuals. In practice $\mu_w = \|\mathbf{z}_W\| \frac{\lambda_{MAX}}{\lambda_{MIN}}$ where $\frac{\lambda_{MAX}}{\lambda_{MIN}}$ is the condition number of $[G^T W G]$ and $\mathbf{z}_W$ the weighted DFD. The weighted Wiener estimator is similar to the IRLS solution given earlier with the exception that an additional regulariser term μ is introduced. This acts like a damper

for the iterative estimation system, limiting the update if the matrix is ill-conditioned or the DFD is too large. Parameter estimation follows a similar multiresolution scheme to before with estimates of Θ^i and W^i updated alternately.

The weights may be estimated using either method outlined earlier. In a way, the only real difference between the two weight functions is that in the first case, the weights are binary while in the second, the weights are a continuous function of $\acute{\epsilon}$. A comparison of robust estimation techniques for global motion estimation is given in [40, 34].

In this thesis this Wiener based solution is the method of choice for generating the Least Squares global motion estimates. The robust histogram technique is used for estimating weights with $T = 10\%$ used throughout. Subsequent references to the IRLS method refer to this robust histogram Wiener based variant. The next section introduces transform domain techniques for GME.

2.5.1.3 Transform Domain Methods

Similar to local motion estimation, transform domain methods may also be used for global motion estimation. Phase-correlation was actually first introduced as a method for image registration by Kuglin and Hines [110]. This however was for translational displacement only. The affine theorem for the 2D Fourier transform by Bracewell et al. [19] led to FFT-based image registration techniques with affine transformations [162, 121]. This has naturally been extended to global motion estimation [73, 111]. The affine theorem for the 2D Fourier transform is defined as follows [19]:
If $I_{n-1}(x, y)$ has 2D Fourier transform $F_{n-1}(u, v)$, then $I_n(x, y) = I_{n-1}(ax + by + c, dx + ey + f)$ has 2D Fourier transform

$$F_n(u, v) = \frac{1}{|\Delta|} exp\left\{\frac{i2\pi}{\Delta}[(ec - bf)u + (af - cd)v]\right\} \times F_{n-1}\left(\frac{eu - dv}{\Delta}, \frac{-bu + av}{\Delta}\right) \quad (2.71)$$

where the determinant is given by $\Delta = ae - bd$.

An interesting feature of this theorem is it allows the decoupling of the estimation of the translational displacement from the estimation of other motion components such as rotation, scaling, and shear. Since the phase term depends only on the translational component it may be estimated separately from the other components using standard phase correlation. Techniques for estimating the other parameters are outlined in Appendix A. Other advantages of this method are its robustness to noise and computational convenience thanks to efficient FFT implementations.

Figure 2.7 illustrates an example of the phase correlation method used for global motion estimation on the garden sequence. The global motion between the two frames is approximately 2 pels to the right. This is reflected in the phase correlation surface which has a peak centered on position $[2, 0]$. This surface can be interpolated to yield a sub-pel accurate estimate if necessary [184].

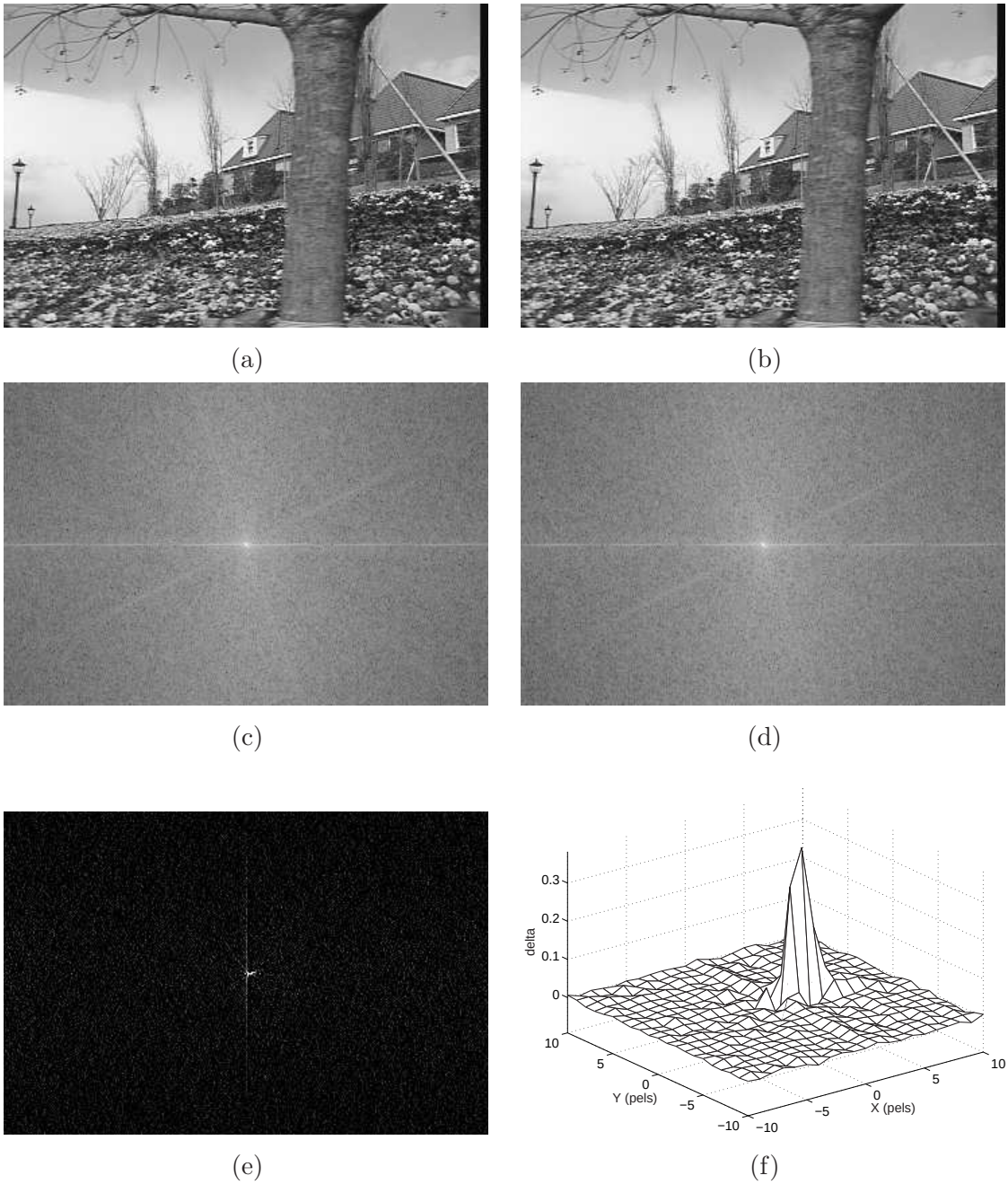


Figure 2.7: *Phase Correlation Motion Estimation. (a) and (b) are two consecutive frames from the garden sequence with (c) and (d) their corresponding 2D Fourier spectra. 2D FFT for first two frames of garden sequence. (e) and (f) show the Phase Correlation function,* $\tilde{c}_{n,n-1}$*, with peak at displacement* $[2, 0]$*, as an image and surface respectively.*

2.5.1.4 Integral Projections

Integral projections may also be used to estimate the global motion between frames. The use of integral projections was quantitatively justified in [133] by using the properties of the Radon Transform. Here the projections are over the whole frame. Crawford et al. [32] proposed an integral projection technique for video stabilisation which used a gradient based estimation scheme to match the 1D projections. This algorithm allows for real-time processing of PAL resolution video (720×576 @25Hz) on a standard desktop PC.

2.5.2 Note on Image Sequence Visualisation

Integral projections offer an interesting additional feature: they may be used as a means for visualising global trends on video sequences. If the projections for each frame are placed side by side and viewed as an image, then it is easy to observe something like the camera motion in a scene over time. Figure 2.8 shows some examples of what shall be denoted here as Temporal Integral Projection (TIP) images. Figure 2.8 (a) shows the horizontal and vertical integral projection images for the caltrain sequence. This sequence contains a pan to the left and no vertical motion. This is clearly seen in the two projection images. The horizontal projection image shows horizontal lines indicating no vertical motion. The vertical projection image shows vertical lines sloping from left to right indicating the left to right horizontal motion of the camera. Figure 2.8 (b) shows the projection images of a sequence shot with a hand-held camera. In this instance the shake of the hand-held camera can be clearly seen as the 'wavy' lines in both the horizontal and vertical directions.

2.5.3 Maximum-a-Posteriori Methods

Recall that MAP methods incorporate one or more priors on the estimation process. The global motion estimation posterior is as follows

$$p(\Theta_n|I_n, I_{n-1}, \Theta_{n-1}) \propto p(I_n|I_{n-1}, \Theta_n)p(\Theta_n|\Theta_{n-1}) \tag{2.72}$$

Much of the work done on MAP global motion estimation methods has focused on video stabilisation. A review of some video stabilisation methods is given in [137]. In this application, the frame to frame motion parameters are estimated and then filtered to provide a smooth camera motion. Hence, video stabilisation or de-shake algorithms have implicit priors if they are to work; they need to know something about the underlying motion in order to correct it.

Often a simple FIR or IIR filter is sufficient. For example Crawford et al. [32] use the following IIR low-pass filter

$$H(z) = \frac{0.0201 + 0.0402z^{-1} + 0.2017z^{-2}}{1 + 1.561z^{-1} - 0.6414z^{-2}} \tag{2.73}$$

This is usually acceptable for random shake, such as illustrated in Figure 2.8 (b). Consider however the example in Figure 2.9. In this case the shake is impulsive, only a few frames around

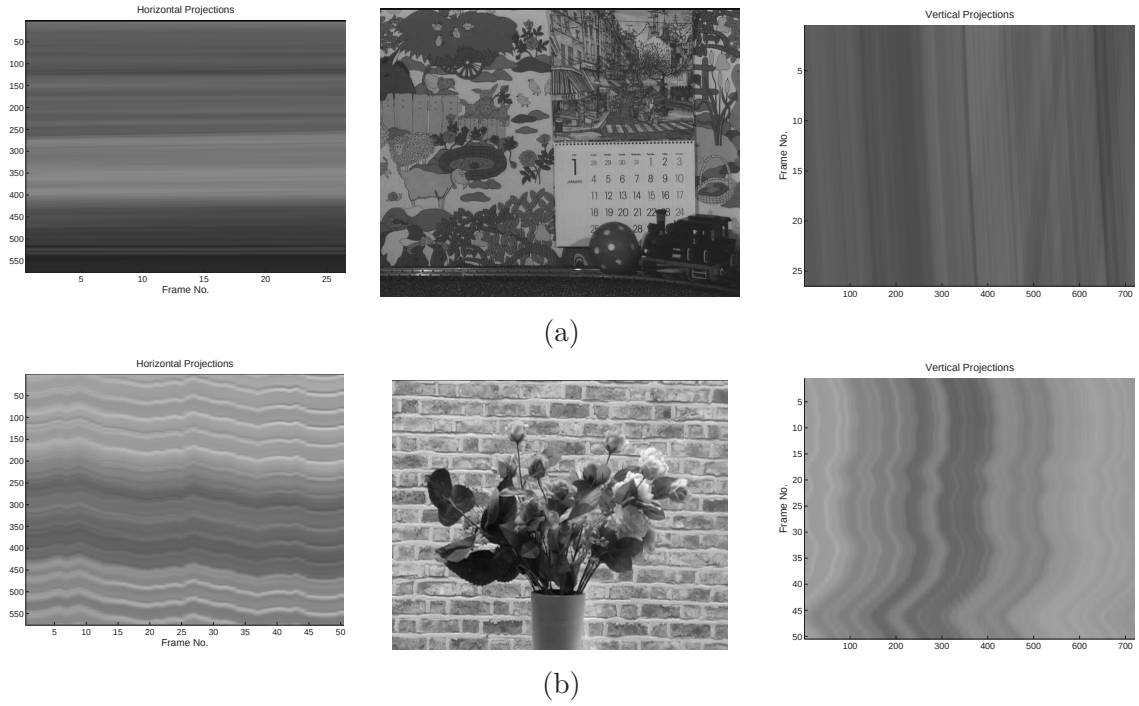


Figure 2.8: *Temporal Integral Projection Images. (a) Caltrain sequence which has a pan to the left and no vertical motion. (b) Flowers sequence which shows hand-held camera shake.*

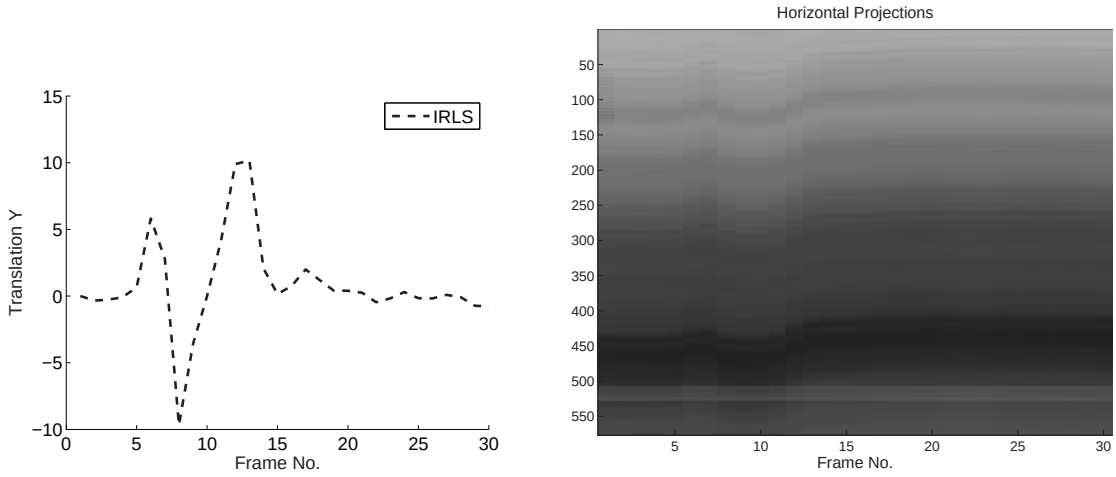


Figure 2.9: *Example of impulsive shake on a video sequence due to poor film splice at scene cut.*

frame 10 need to be compensated. Here the IIR filter will tend to compensate frames outside of the region of shake and leave a noticeable effect on the otherwise stable sequence. Compensation of this type of shake should hence use a different method to that of random shake. Incorporating a history of the previous motion estimates should help a diagnostic tool to distinguish between the two types of shake.

The quality of any video stabilisation method is very much dependent on the global motion estimate between frames. Certain properties within the image data, such as periodic structure, can lead to ambiguous matching in motion estimation. Also problems such as fast camera motion can lead to incorrect motion estimates.

Chapter 4 presents an on-line global motion estimation algorithm based on the Particle Filter technique. The Particle Filter is an efficient tool for incorporating prior information in a non-linear framework. By using the Particle Filter to efficiently incorporate the temporal smoothness of the global motion it is expected to overcome some of the problems outlined here.

2.5.3.1 Viterbi Method

Pilu [157] proposed a method for video stabilisation using a variational formulation with a numerical solution based on the Viterbi algorithm [57, 161]. In that approach the trajectory of the motion estimates in time is assumed to follow a curve $f(t)$. Given a bound on the maximum displacement of each frame w_f, a family of curves may be set up. The problem is to find the smoothest possible path between the curves $f(t) \pm w_f$.

This family of curves $\mathcal{F}$ can be parameterised by $g(t) = f(t) - w_f + 2w_f\lambda(t)$ where $\lambda(t) : \Re \rightarrow [0, 1]$. $\lambda(t)$ is quantized into a number of states. The viterbi algorithm is then used to find the optimum path through the trellis defined by all the possible states at each time step t. In the optimal case the filtering is done off-line where the motion parameters for all frames are available. The length of the trellis is then the same as the number of frames in the sequence. In practice however this may not be practical due to memory limitations. A fixed length sliding trellis is used instead to give a fixed lag estimate of the smoothest path.

2.6 Hierarchical Motion Estimation

A key problem in many motion estimation schemes is dealing with large displacements. Pel-recursive algorithms can only effectively estimate small displacements, since the updates at each iteration are intended to be small in order to keep the Taylor series expansion valid. In a correspondence technique such as block matching, the maximum displacement which can be estimated is only limited by the size of the search space. However, for large displacements increasing the search space significantly increases the computational requirement.

Multiresolution schemes provide a practical way of dealing with this problem. Figure 2.10 depicts a typical hierarchical pyramid representation used, with $N = 3$ levels. The base of the pyramid, level 0, is the original image. Each subsequent level in the pyramid is obtained by

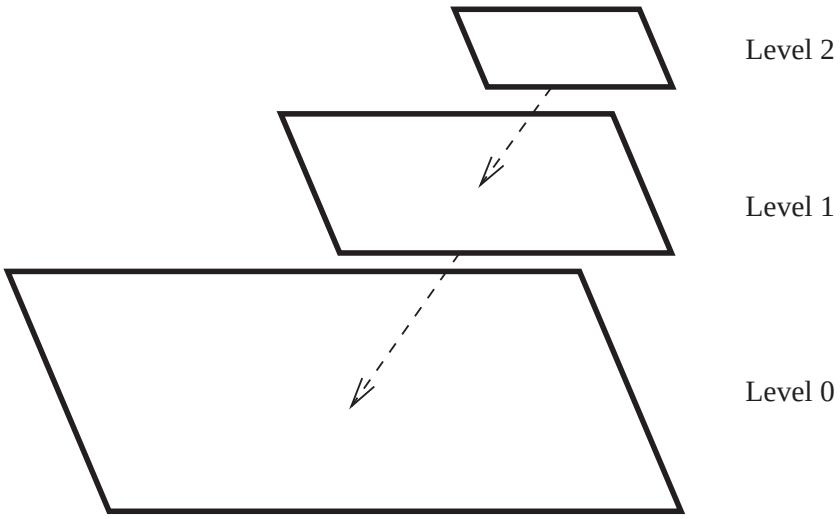


Figure 2.10: *Three level pyramid for multiresolution scheme. Each level is low-pass filtered and sub-sampled by a factor of two from the level below.*

low-pass filtering and subsampling the previous level. Typically each level is downsampled by a factor of two (*dyadic* downsampling). Because the resolution of the images is halved at each level, the displacement at each level is also halved. So, motion of magnitude k pixels at level 0 (the original resolution level), is then reduced to $k \times 2^{-(N-1)}$ at level $N-1$. This has the effect of greatly increasing the effective search range. Motion estimation proceeds in what is known as a *coarse–to–fine* manner. Starting at the coarsest level, $N-1$, motion estimation is performed at each level with the resulting motion vectors projected onto the next higher resolution level as an initial estimate. The idea being that the projected motion vectors will be close enough to the actual motion vectors at each level such that a reduced search space for block matching, or a few iterations for pel-recursive methods, is sufficient to estimate the motion at that level.

The earliest pyramidal representation of images was proposed by Burt and Adelson [24] for use in image coding. In their Laplacian pyramid, each level is a bandpass representation of the original image. In motion estimation algorithms a Gaussian shaped low-pass filter is typically used, which slightly blurs the image at each level. As noted by Kearney et al. [91] this provides a further benefit to pel-recursive techniques. It artificially increases the importance of the first order term in the Taylor series expansion, thus helping to stabilize the iterative process.

Multiresolution techniques, which have been employed for the motion estimation algorithms outlined here, include Bierling [11] for block matching, Erkam et al. [51] for phase correlation, and Enkelmann [50] for pel-recursive methods. Hierarchical motion estimation methods based on the discrete wavelet transform (DWT) [126] have also been developed [204], as well as the complex wavelet transform (CWT) [125].

2.7 Motion Estimator Performance Evaluation

One consideration while developing any new technique is its ability to improve upon existing techniques. To evaluate a motion estimator performance a subjective comparison could be conducted between different algorithms by viewing the resulting motion fields or motion compensated difference images. However, depending on the required application, there can be ambiguity regarding the quality of the motion field. In video coding for instance, the goal is finding motion vectors which will minimise the prediction error. Smooth motion fields are also important as a smooth motion field is easier to compress than an inconsistent one. On the other hand for something like scan-rate conversion estimating the true motion of the scene is more important. These criteria are often not equivalent and can lead to different motion fields. It is therefore necessary to somehow try and quantify motion estimation performance. The following common measures are used in this thesis.

2.7.1 MSE & PSNR

To indicate the performance of a motion estimator, often a mean squared error (MSE) is used, or the entropy of the prediction error. This directly relates to the prediction error in video coding and is a useful guide to the likely performance of the motion estimator for video coding. The MSE for frame n given the vector field $\mathbf{d}_{n,n-1}$ and frame $n-1$ is

$$MSE(n) = \frac{1}{|W|} \sum_{\mathbf{x} \in W} (I_n(\mathbf{x}) - I_{n-1}(\mathbf{x} + \mathbf{d}_{n,n-1}(\mathbf{x})))^2 \tag{2.74}$$

The measurement window W, is a window over the entire image, excluding some border region. The peak signal–to–noise ratio (PSNR) is a common metric used and is defined as

$$PSNR = 10log_{10}\left(\frac{255^2}{MSE}\right) \tag{2.75}$$

2.7.2 M2SE

A modification of the MSE, denoted the Modified Mean Square Error (M2SE), was introduced by de Haan et al. [36] as a performance indicator for how well the estimated motion vectors represent the true-motion of the sequence. The essence of the modification is that the validity of the vectors is extrapolated outside the temporal interval on which they are estimated, since such extrapolation is only plausible in case the vector describes the true-motion. A motion compensated frame, I_{mc}, is reconstructed from frames $n-1$ and $n+1$ using vector field $\mathbf{d}_{n,n-1}$,

$$I_{mc}(\mathbf{x}) = \frac{1}{2}(I_{n-1}(\mathbf{x} + \mathbf{d}_{n,n-1}(\mathbf{x})) + I_{n+1}(\mathbf{x} - \mathbf{d}_{n,n-1}(\mathbf{x}))) \tag{2.76}$$

The M2SE is then the MSE of the original frame I_n and the reconstructed one,

$$M2SE(n) = \frac{1}{|W|} \sum_{\mathbf{x} \in W} (I_n(\mathbf{x}) - I_{mc}(\mathbf{x}))^2 \tag{2.77}$$

2.7.3 Smoothness

A measure for the smoothness of the estimated motion field can be derived from the prior used for motion. For each site $\mathbf{x}$, the logarithm of the prior probability (2.10) yields this measure. It is given as the sum of the squared difference of the vectors in the eight connected neighbourhood around the site.

$$E_s(\mathbf{x}) = \sum_{k}^{8} [\mathbf{d}_{n,n-1}(\mathbf{x}) - \mathbf{d}_{n,n-1}(\mathbf{x} + \mathbf{o}_k)]^2 \tag{2.78}$$

where $\mathbf{o}_k = \{[-1,-1],[-1,0],\ldots,[1,1]\}$. A large value for E_s indicates poor spatial consistency and hence poor motion smoothness. A small value indicates good spatial consistency and hence strong motion smoothness. The smoothness measure used for the entire motion field is the mean of the E_s over all the sites. This overall measure is defined as $\bar{E}_s$.

2.8 Video Segmentation

Video object segmentation and motion estimation are two overlapping problems. If each frame of the video sequence can be accurately segmented into its component objects, motion estimation then becomes the problem of tracking each object. Likewise an accurate estimate of the motion of objects in a video sequence can be used as a basis for segmentation. Some notable work in video segmentation has been done by Wang and Adelson [194], Weiss and Adelson [197], Stiller [172, 173], and Torr et al. [188]. It is widely accepted that the video segmentation problem is a highly complex task and that motion estimation is a key part of this problem. This thesis focuses on the extraction of motion information from video sequences.

2.9 Final Comments

This chapter has given a review of some of the more common approaches to local and global motion estimation. It was also shown how these algorithms may be unified using a Bayesian framework. Under this framework, different motion estimation methods arise from different choices of the likelihood and prior models.

As stated previously a prior term may be included in the motion estimator to deal specifically with motion discontinuities. It was also noted that in order to use global motion estimates for shake removal or camera tracking, a temporally smooth estimate of the parameters is necessary. The next two chapters are specifically concerned with

- **Chapter 3:** Efficient exploitation of priors for propagation and/or estimation of local motion parameters to deal with smoothness and spatial discontinuities.
- **Chapter 4:** Incorporating temporal smoothness into global motion estimation for on-line processing.

3

Candidates in Motion [1]

The previous chapter provided a review of classical motion estimation techniques and introduced the idea of using prior information on the motion field to improve the estimate. Another way of thinking about motion estimation is the following: given a set of candidate vectors for the motion in a sequence, choose the candidate at each site in the frame which minimises some cost function for that site. In block matching for example, the candidate set is defined by the size of the search space and the required accuracy of the motion vectors. If the candidate set contains the correct motion vector for each site, and some prior information about the motion field is known, then an accurate motion field should be obtained. Furthermore, if by using prior information the number of candidates at each site can be kept low, then the motion estimator can also be very efficient. The 3DRS motion estimator by de Haan et al. [36] is a good example of this way of thinking about motion estimation.

As already seen incorporating prior information in the motion estimation solution merely involves adding an extra term to the cost function and leads to a MAP formulation. This chapter focuses on different strategies for solving the MAP estimation problem. Two algorithms are presented; one based on the ICM algorithm and the other based on Belief Propagation. The implementations of both methods are based on the idea of candidate selection for motion estimation, where a limited number of candidate motion vectors are tested at each site. By using candidate selection, these methods aim to address problems arising from the aperture effect, areas with low image gradient, etc. The choice of candidate vectors is key to the performance

[1] Results from this chapter have been submitted as: Francis Kelly and Anil Kokaram. Candidates in Motion. *Submitted to* 9^{th} *European Conference on Computer Vision (ECCV'06),* May 2006. [97]

of these algorithms.

3.1 MAP Optimization

As outlined in chapter 2 the motion estimation/smoothing problem may be formulated as a maximum a posteriori (MAP) markov random field (MRF) problem. There are several methods which can be used to solve the MAP-MRF problem. At the heart of all these methods is a step that locally perturbs the solution at a pixel or subset of pixels, hoping to improve the estimate at each iteration. The methods can be classified as either stochastic or deterministic. Deterministic methods such as Newton-Raphson and variational approaches are generally of lower computational cost than stochastic techniques. One of main class of stochastic techniques used are Markov Chain Monte Carlo (MCMC) methods, of which simulated annealing (SA) is one example. Deterministic relaxation methods like mean field annealing (MFA) [12], Highest Confidence First (HCF) [27], and Iterated Conditional Modes (ICM) [9] were developed to reduce the load of MCMC methods. In recent years two algorithms, Bayesian Belief Propagation [155] and Graph Cuts [18], have revolutionised the solution of MAP-MRF problems.

For the purposes of this work it is helpful to consider some aspects of an MCMC solution to the problem. Consider the following posterior distribution for the motion field $\mathbf{D}$ between frames n and $n-1$,

$$\mathbf{D}_{n,n-1} \sim p(\mathbf{D}_{n,n-1}|I_n, I_{n-1}) \tag{3.1}$$

$\mathbf{D}$ represents the entire motion field and it contains $M \times N \times 2$ unknown elements requiring estimation. The aim is to manipulate this probability distribution to yield the best motion field between the two frames. The essence of a stochastic solution is to draw samples of $\mathbf{D}$ from this distribution, hence numerically evaluating the pdf. The optimal $\mathbf{D}$ is then the one which is most probable. However drawing samples from this distribution is often quite difficult. One approach involves directly evaluating $p(\mathbf{D}_{n,n-1}|I_n, I_{n-1})$ at every possible value of $\mathbf{D}_{n,n-1}$ and hence obtain the cumulative density function (cdf). A numerical method may then be used to draw a sample from $\mathbf{D}_{n,n-1}$ [159]. Typically there are too many sites in an image and too many possible values for $\mathbf{D}_{n,n-1}$ for this to be feasible. The Gibbs sampler as presented by Geman and Geman [62] can be used to break the problem down into a draw from a number of conditional distributions which are easier to sample from. This is discussed next.

3.1.1 Gibbs Sampler

The Gibbs sampler [62] may be thought of as a special case of the MCMC Metropolis-Hastings algorithm [181]. It is a method for drawing samples from complicated distributions. The Gibbs sampler decomposes a draw from a joint multidimensional distribution into a draw from a number of conditional distributions of smaller dimension.

In this case the multidimensional draw can be decomposed into draws from local conditional

distributions at every site $\mathbf{x}_i$. Starting with an initial estimate for $\mathbf{D}_{n,n-1}$, defined as $\mathbf{D}^0_{n,n-1}$ which may be a zero estimate, samples for each site $\mathbf{x}_i$ are drawn as follows

$$\begin{aligned}
\mathbf{d}^1_{n,n-1}(\mathbf{x_1}) &\sim p(\mathbf{d}_{n,n-1}(\mathbf{x_1})|I_n, I_{n-1}, \mathbf{D}^0_{n,n-1}(-\mathbf{x_1})) \\
\mathbf{d}^1_{n,n-1}(\mathbf{x_2}) &\sim p(\mathbf{d}_{n,n-1}(\mathbf{x_2})|I_n, I_{n-1}, \mathbf{D}^0_{n,n-1}(-\mathbf{x_1}, -\mathbf{x_2}), \mathbf{d}^1_{n,n-1}(\mathbf{x_1})) \\
&\vdots \\
\mathbf{d}^2_{n,n-1}(\mathbf{x_1}) &\sim p(\mathbf{d}_{n,n-1}(\mathbf{x_1})|I_n, I_{n-1}, \mathbf{D}^1_{n,n-1}(-\mathbf{x_1})) \\
&\vdots
\end{aligned} \tag{3.2}$$

$\mathbf{D}^0_{n,n-1}(-\mathbf{x_1})$ denotes all the motion vectors in the motion field at iteration 0 which are *not* at site $\mathbf{x_1}$ and $\mathbf{d}^i_{n,n-1}(\mathbf{x}_j)$ denotes the ith sample of $\mathbf{d}_{n,n-1}$ at site $\mathbf{x}_j$. The method proceeds by drawing samples for $\mathbf{d}_{n,n-1}$ at each site given the current state of samples for the motion field. The sample drawn at a site $\mathbf{x}_j$ then replaces the previous sample at that site. The process continues at the next site where a motion sample is drawn based on the current state of the motion field which now includes the sample just drawn at the previous site. This is repeated iteratively until convergence when the samples being generated are samples from the multidimensional distribution $p(\mathbf{D}_{n,n-1}|I_n, I_{n-1})$ as required.

Of course it now becomes necessary to be able to draw samples from the conditional distribution $p(\mathbf{d}_{n,n-1}(\mathbf{x})|\cdot)$ at each iteration. Again a direct method of evaluating all possible values of $\mathbf{d}_{n,n-1}$ to generate the cdf may be used. Alternatively the Griddy sampler [181] can be evoked to obtain an approximation to the cdf by evaluating $p(\mathbf{d}_{n,n-1}(\mathbf{x})|\cdot)$ on a grid of points. A sample from the distribution may then be obtained by transforming a uniformly distributed random variable using the numerically evaluated cdf. The location of the grid points depends on the implementation. However after the step of random sample generation, there is still the next step of manipulating the samples to yield a MAP estimate. Rather than use MCMC, the best estimate at the current site could be chosen. Doing this iteratively at each site leads to the Iterated Conditional Modes (ICM) algorithm of Besag [9].

3.1.2 Iterated Conditional Modes

ICM is a deterministic procedure which aims to reduce the computational load of stochastic relaxation methods. ICM maximises the local conditional probabilities at each site and converges to some local minimum over all sites in the image [9]. Its implementation is similar to that of the Gibbs sampler, but in the case of ICM the value chosen at each site is the one which maximises the local conditional probability, rather than drawing a value based on the conditional probability distribution.

The MAP estimation problem is to find the motion field which gives the maximum posterior probability, i.e.,

$$\widehat{\mathbf{D}}_{n,n-1} = \max_{\mathbf{D}} p(\mathbf{D}_{n,n-1}|I_{n-1}, I_n) \tag{3.3}$$

This is done by choosing the values which maximise the conditional distributions at each site and proceeding in a similar manner to that outlined for the Gibbs sampler.

$$\begin{aligned}\hat{\mathbf{d}}^1_{n,n-1}(\mathbf{x_1}) &= \max_{\mathbf{d}(\mathbf{x_1})} p(\mathbf{d}_{n,n-1}(\mathbf{x_1})|I_n, I_{n-1}, \mathbf{D}^0_{n,n-1}(-\mathbf{x_1})) \\ \hat{\mathbf{d}}^1_{n,n-1}(\mathbf{x_2}) &= \max_{\mathbf{d}(\mathbf{x_2})} p(\mathbf{d}_{n,n-1}(\mathbf{x_2})|I_n, I_{n-1}, \mathbf{D}^0_{n,n-1}(-\mathbf{x_1}, -\mathbf{x_2}), \mathbf{d}^1_{n,n-1}(\mathbf{x_1}) \\ &\vdots\end{aligned} \tag{3.4}$$

ICM converges much faster than stochastic simulated annealing algorithms. However it is likely to get trapped in local minima and hence it is critical to initialize ICM with a reasonably good initial estimate. This is the function of the candidate selection strategy.

3.1.2.1 Implementation

The conditional distribution may be expressed using Bayes theorem as

$$p(\mathbf{d}_{n,n-1}(\mathbf{x})|I_n, I_{n-1}, \mathbf{D}(-\mathbf{x})) \propto \underbrace{p(I_n|I_{n-1}, \mathbf{D}(-\mathbf{x}), \mathbf{d}_{n,n-1}(\mathbf{x}))}_{likelihood} \underbrace{p(\mathbf{d}_{n,n-1}(\mathbf{x})|\mathbf{D}(-\mathbf{x}))}_{prior} \tag{3.5}$$

The *likelihood* is directly related to the image data and should penalise motion vectors which give a high DFD

$$\begin{aligned} p(I_n, I_{n-1}|\mathbf{D}(-\mathbf{x})), \mathbf{d}(\mathbf{x})) &\propto \exp\left(-\frac{\sum_{\mathbf{x}}(I_n(\mathbf{x}) - I_{n-1}(\mathbf{x}+\mathbf{d}(\mathbf{x})))^2}{2\sigma_e^2}\right) \\ &\propto \exp\left(-\frac{\|DFD(\mathbf{x}, \mathbf{d}(\mathbf{x}))\|^2}{2\sigma_e^2}\right) \end{aligned} \tag{3.6}$$

The *prior* distribution may be used to incorporate constraints on the motion field. Following on from work by Konrad and Dubois [108], Stiller [171] and Kokaram [101], the motion field is modelled as an MRF. The MRF used here is a four-connected MRF shown in Figure 3.1. This leads to the following prior distribution

$$p(\mathbf{d}(\mathbf{x})|\mathbf{D}(-\mathbf{x})) = \frac{1}{Z}\exp\left(-\Lambda\sum_{k=1}^{N}|\mathbf{d}(\mathbf{x}) - \mathbf{d}(\mathbf{x}+\mathbf{q}_k)|\right) \tag{3.7}$$

where Z is a normalisation factor and $\mathbf{q}_k$ is an offset corresponding to the N nearest neighbours as defined by the MRF. Λ is a user defined parameter to control the influence of the smoothness term; varying Λ encourages the motion field to be more or less smooth.

As noted in the previous chapter, the MAP problem may be expressed as an energy minimisation problem. This minimisation is over the sum of two energies; a likelihood or local data cost, and a prior or a spatial smoothness energy.

$$E(\mathbf{d}(\mathbf{x})) = L(\mathbf{d}(\mathbf{x})) + V(\mathbf{d}(\mathbf{x})) \tag{3.8}$$

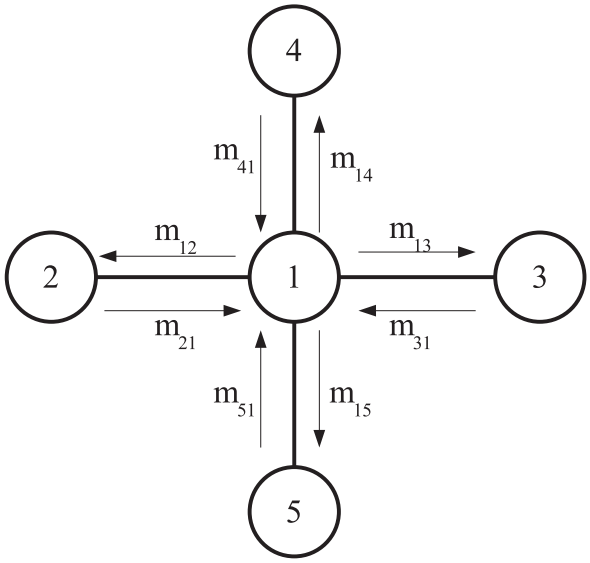


Figure 3.1: *Four-connected MRF used in ICM and Belief Propagation. A node can only communicate with those directly connected to it on the image grid. Also shown are the messages passed between the nodes in Belief Propagation.*

Here

$$L(\mathbf{d}(\mathbf{x})) = \sum_{\mathbf{x} \in Block} (I_n(\mathbf{x}) - I_{n-1}(\mathbf{x} + \mathbf{d}(\mathbf{x})))^2 \tag{3.9}$$

and

$$V(\mathbf{d}(\mathbf{x})) = \frac{1}{Z} \left(\Lambda \sum_{k}^{N_n} |\mathbf{d}(\mathbf{x}) - \mathbf{d}(\mathbf{x} + \mathbf{q}_k)| \right) \tag{3.10}$$

To generate a solution for the best vector at a local site, a measurement of E must be made for a number of possible motion vectors at that site. This set of possible vectors may be an exhaustive list of motion vectors e.g. in FBM for instance all motion vectors yielding ± 5 pels motion with an accuracy say of 0.25. Alternatively some more useful *candidate* set of vectors could be evaluated at that site. Discussion about what that candidate set should be is postponed till later in this chapter, but it suffices to say that the best candidate set would somehow include the *correct motion vector* at that site.

Hence, at each site the candidate vector which minimises E is chosen to be the vector for that site

$$\hat{\mathbf{d}}(\mathbf{x}) = \arg\min_{\mathbf{c}_i} E(\mathbf{d}(\mathbf{x}) = \mathbf{c}_i) \tag{3.11}$$

3.2 Belief Propagation

The method of Belief Propagation (BP) was first presented by Pearl [155]. It arises directly from the need to manipulate $p(\mathbf{D}|I_n, I_{n-1})$, the probability of a particular solution $\mathbf{D}$ given

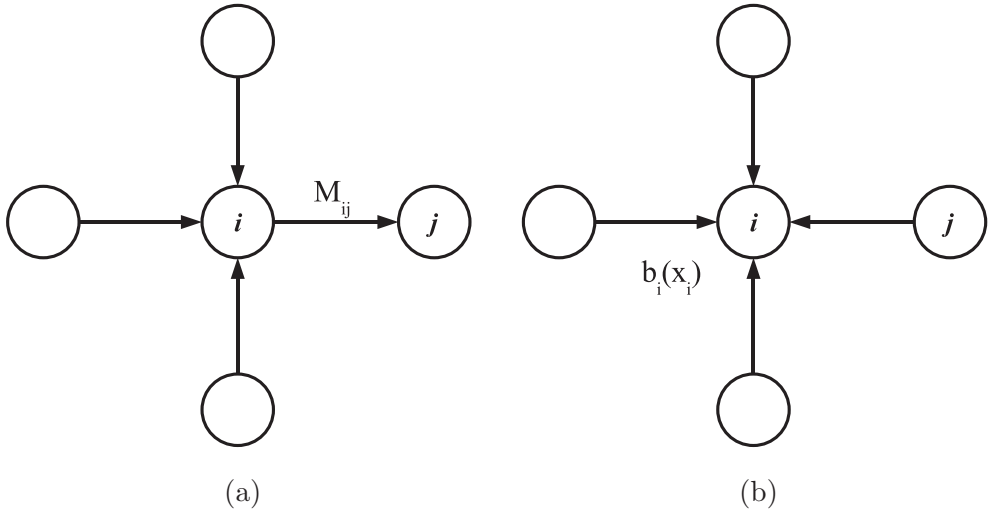


Figure 3.2: *Graphical representation of BP in action. (a) Generating message to pass from node i to node j. (b) Calculating the belief at a node.*

the data only. It is a form of marginal inference since the algorithm requires the manipulation of $p(\mathbf{d}(\mathbf{x})|I_n, I_{n-1})$ at each site $\mathbf{x}$. Compare this with the Gibbs sampler which manipulates $p(\mathbf{d}(\mathbf{x})|I_n, I_{n-1}, \mathbf{D}(-\mathbf{x}))$ at each site. It transpires that this is a very subtle difference since at the heart of both processes is the need to take account of nearby estimates of $\mathbf{D}$. Appendix B presents some more detailed background on BP. However, for the purposes of this discussion it is sufficient to note that the BP algorithm is reliant on the factorisation of $p(\mathbf{d}(\mathbf{x})|I_n, I_{n-1})$ into a number of probability distributions known as "messages". Each node sends each of its neighbours a message containing information about what state it believes that node should be in. In addition, after a certain number of iterations of message passing, each node calculates a "belief", a probability distribution over its own state. This belief is based on its local data and the messages received from neighbouring nodes. The state which maximises the belief at each node is chosen as the state for that node.

Figure 3.2 gives a graphical representation of BP in action. In this case the network is a four connected MRF. Node i wishes to pass a message to node j. This message, $M_{i,j}$, is generated using all the messages coming into node i *except* the message coming from node j, Figure 3.2 (a). The four incoming messages are used to calculate the belief at a node, Figure 3.2 (b).

3.2.1 Loopy Belief Propagation

BP is an exact inference method if the network contains no loops, i.e. is singly connected. However, as noted by Pearl [155, pp.195], if the network contains loops, then BP is not guaranteed to converge. Despite this, BP has been applied successfully in some vision and communications problems where the underlying network contains loops. Loopy Belief Propagation is belief propagation that ignores the existence of loops in the network [179]. A good example is the near

Shannon-limit performance of "Turbo codes", whose decoding algorithm is equivalent to BP on a loopy network [7, 130]. Also, good results have been reported for computer vision problems where the underlying MRF is full of loops [59].

3.2.2 Implementation

BP can be thought of as a parallel message passing algorithm. Every node sends a message to its neighbours representing what state it believes that node should be in. Here the nodes represent a pixel or a group of pixels in the image, and the states are the possible motion vectors at each node. A message to a neighbour depends on the messages received from all other neighbours. There are three key decisions to be made while implementing the BP algorithm: the message update schedule, the message update rule, and the belief update schedule.

Message Update Schedule: The update schedule determines when a message sent to a node will be used by that node to compute new messages for its neighbours. In a synchronous update schedule, each node first computes the messages for each neighbour. Once every node has computed its messages, the messages are delivered to each neighbour and used to compute the next round of messages. An alternative schedule is to propagate messages in one direction only, for example to the right, and update each node immediately to send a new message to its right. Once this has been completed for every row, the same procedure occurs in the up, down, and left direction. This is known as "accelerated" updating [182]. In this work a synchronous update schedule was used as this is more amenable to parallel implementation.
Message Update Rule: The message update rule determines how to compute new messages based on the messages received from neighbouring nodes. There are two kinds of BP algorithm with different message update rules: "max-product" and "sum-product", which maximise the joint posterior of the network, and the marginal distribution of each node respectively[2]. The max-product algorithm is sometimes known as belief revision and is equivalent to the Viterbi algorithm [191, 57, 161]. The sum-product algorithm computes the marginal distribution at each node and hence may be used for generating a MMSE estimate. The max-product version of BP computes a MAP estimate at each node and is used here to compare with the MAP estimate of ICM.

The max-product BP algorithm works by passing messages around the graph defined by the four-connected image grid, see Figure 3.1. Normally this algorithm is defined in terms of probability distributions. However using negative log probabilities leads to an equivalent computation where the max-product becomes a min-sum [53].

As already seen, using negative log probabilities allows the MAP problem to be expressed as an energy minimisation problem. This minimisation is over the sum of two energies; a likelihood or local data cost, and a prior or a spatial smoothness energy.

$$E(\mathbf{d}(\mathbf{x})) = L(\mathbf{d}(\mathbf{x})) + V(\mathbf{d}(\mathbf{x})) \tag{3.12}$$

[2]See Appendix B for more details.

The likelihood cost function is given by (3.6). The prior term incorporates the smoothness constraint on the motion field. It penalises vectors which differ from those in some local neighbourhood. The form of the prior term in this case however is slightly different. In BP each node $\mathbf{p}$ sends each of its neighbours $\mathbf{q}$ a message containing information about what it believes the motion vectors at the receiving node should be. Hence the smoothness term is given by

$$V(\mathbf{d}(\mathbf{p}), \mathbf{d}(\mathbf{q})) = \|\mathbf{d}(\mathbf{p}) - \mathbf{d}(\mathbf{q})\| \tag{3.13}$$

Let m_{pq}^t be the message that node $\mathbf{p}$ sends to a neighbouring node $\mathbf{q}$ at time t. Each message is in the form of a vector whose dimension is given by the number of possible motion vectors at the receiving node. When using negative log probabilities all entries in m_{pq}^0 are initialised to zero. At each iteration new messages are computed as follows,

$$m_{pq}^t(\mathbf{d}(\mathbf{q})) = \min_{\mathbf{d}(\mathbf{p})} \left(V(\mathbf{d}(\mathbf{p}), \mathbf{d}(\mathbf{q})) + L(\mathbf{d}(\mathbf{p})) + \sum_{s \in \mathcal{N}(\mathbf{p}) \backslash \mathbf{q}} m_{sp}^{t-1}(\mathbf{d}(\mathbf{p})) \right) \tag{3.14}$$

where $\mathcal{N}(\mathbf{p})\backslash\mathbf{q}$ denotes the neighbours of $\mathbf{p}$ other than $\mathbf{q}$. Typically messages are passed around for T iterations and then a belief is calculated for each node. The belief represents the probability distribution over the possible motion vectors at a node.

$$b(\mathbf{d}(\mathbf{q})) = L(\mathbf{d}(\mathbf{q})) + \sum_{\mathbf{p} \in \mathcal{N}(\mathbf{q})} m_{pq}^T(\mathbf{d}(\mathbf{q})) \tag{3.15}$$

The motion vector $\hat{\mathbf{d}}(\mathbf{q})$ that minimises the belief $b(\mathbf{d}(\mathbf{q}))$ individually at each node is selected as the vector for that node

$$\hat{\mathbf{d}}(\mathbf{q}) = \arg\min_{\mathbf{d}(\mathbf{q})} b(\mathbf{d}(\mathbf{q})) \tag{3.16}$$

Belief Update Rule The belief update rule determines how often the belief at a node will be computed. The number of iterations, T, bounds the distance information can propagate across the image. In a standard belief propagation algorithm T needs to grow by $n^{1/2}$ where n is the number of pixels in the image [53]. This is necessary to allow information from one part of the image to propagate everywhere else. For motion estimation however this much propagation is not necessary. Four different values for T were tested here, $T = 1, 3, 5, 10$, with a maximum of 10 iterations per frame. After the belief for a node has been computed the new motion vector for that node is obtained.

In the literature, where BP has been applied to image processing, the unknown states of the system typically come from some discrete label set. These labels are applicable for each node in the image and the messages are vectors whose length are the same size as the number of possible labels. Hence each node sends messages about what it considers that neighbours label should be.

The task here is to select the motion vectors at each site over which information about belief is to be gathered. The simplest approach is to assume (as in FBM) that at each site motion is

allowed to be $\pm w$ pixels with some fractional accuracy. Hence for ± 5 pixel motion at 0.25 pel accuracy, there are 1936 motion vectors to be evaluated at each site in the image. This implies that the messages passed around the grid would be all 1936 elements long. This is a massive computational undertaking and it is sensible instead to consider some mechanism to select a useful subset of the motion space at each site. This is discussed next.

3.3 Exploiting Candidates for Motion

Having introduced two optimisation strategies for the motion estimation problem, it is clear that the outstanding issue is generating candidate motion vectors for evaluation at each site in the image. Experience with motion estimators of any type shows that results from any motion estimator may be poor over the image as a whole, but tend to be very appropriate in some parts of the image. For instance, almost any simple likelihood-only motion estimator (gradient, phase or BM) would yield a good motion estimate at corners and in textured regions of the image. Where they fail are in areas where there is simply not enough *evidence from the data alone* to make a judgement about motion e.g. in flat areas or at edges. Given that objects tend to have a scale that is much larger than a pixel in the image, this implies that the motion of some corners and other high data-confidence areas are most likely the motion of pixels in those low data confidence areas. Hence what is needed is a process that can *propagate* motion information from high confidence to low confidence areas. The processes described previously can achieve exactly this goal by employing candidates derived from likelihood-only motion estimators, or indeed candidates already selected in other areas of the picture. It is highly likely that such candidate sets would contain the best motion estimate for a particular site, if not in the current iteration, then in the next.

Based on this observation, it is plausible to test only a limited number of candidates at each site. It is assumed that there is already an initial estimate for the motion field available from some motion estimation algorithm. Here the Wiener Based Motion Estimator (WBME) introduced in chapter 2 is used to generate these raw, or initial estimates. The ICM and BP methods work by carefully selecting candidates at each site from these initial estimates and applying the relevant rules outlined earlier.

The candidate selection strategy is crucial to the quality of the resulting motion field in these algorithms. If none of the candidates for a site contain the correct motion vector, then there is little that the algorithm can do except chose the one which gives the lowest energy. Hence the candidates should be chosen in a way to give the most probable motion vectors at a given site. Two simple techniques for candidate selection were tested here and are outlined as follows.

- **Nearest Neighbours (NN):** The simplest candidate selection strategy is to choose motion vectors from the immediate neighbourhood of the current site. This encourages strong local smoothness in the vector field. Here the eight nearest neighbours to the current block

are chosen as candidate vectors.

- **Histograms and Nearest Neighbours (HNN):** A histogram of the entire motion field is calculated and the five highest frequency vectors chosen as candidates. The four neighbours to the left, right, top, and bottom are also used as candidates to yield nine candidate vectors in total at each site.

3.3.1 General Candidate Selection Motion Estimation Algorithm

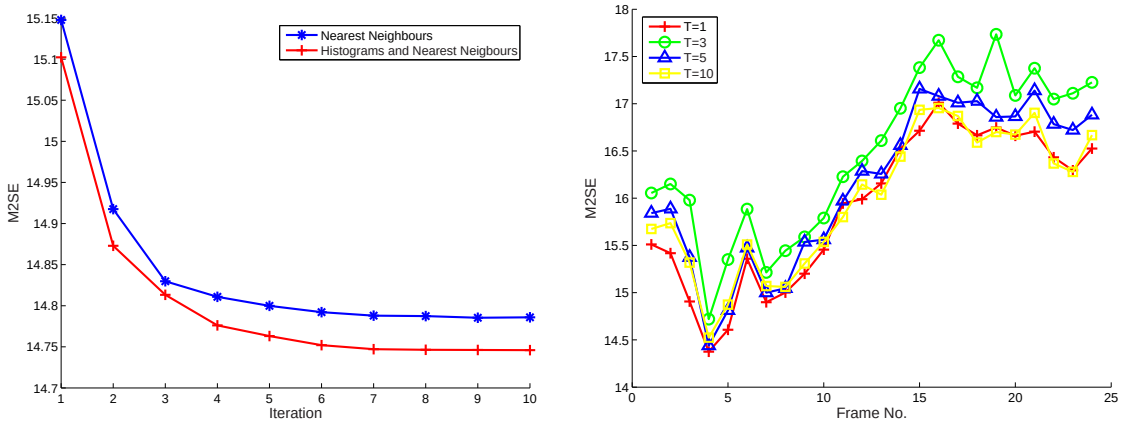
Both candidate selection strategies outlined here lead to similar implementations. The following general algorithm may be applied to both ICM and BP.

1. For each site in the current image:
   (a) Obtain a set of candidate vectors, $\mathbf{c}_i$, for current site $\mathbf{x}$.
   (b) Perform one iteration of either ICM or BP at that site.
      - ICM: Calculate energy $E()$ associated with each vector according to (3.8), (3.9), and (3.10).
      - BP: Perform message passing scheme for that site using (3.14).
   (c) Goto the next site.
2. Choose new motion vector for each site.
   - ICM: Choose motion vector corresponding to the minimum energy at each site.
   - BP: Calculate the belief for each site depending on the belief update rule.

This is performed iteratively until the motion field converges or for some maximum number of iterations. In the results that follow a maximum of 10 iterations was used.

3.4 Results

In this section the results for three test sequences are discussed. These are the caltrain sequence, the yosemite sequence, and the garden sequence and frames from each with overlayed motion fields are shown in Figures 3.8, 3.7, and 3.9 respectively. For each test sequence there are four sets of results. A multi-resolution WBME is used to generate the initial or raw vector field. The ICM and BP algorithms are then used to refine these raw vectors. To give a comparison against another candidate selection algorithm, results from the 3DRS motion estimator outlined in the previous chapter are also provided. Two performance metrics are used here to give an indication of the quality of the vector field arising from each method. These are the M2SE and smoothness measures outlined in chapter 2. Results for more test sequences are provided in Appendix B.



(a) (b)

Figure 3.3: *(a) Convergence of ICM for two candidate selection strategies, NN and HNN. (b) Comparison of four Belief update iteration values,* $T = 1, 3, 5, 10$.

3.4.1 Candidate Selection Strategy

The first thing to consider with a candidate selection motion estimation algorithm is the candidate selection strategy to use. Figure 3.3 (a) illustrates the convergence speed of the two strategies outlined earlier, NN and HNN, for the ICM algorithm on the caltrain sequence. These results indicate that HNN gives faster convergence and lower M2SE than the NN method. Hence this was the strategy chosen while generating both the ICM and BP results presented here. The 3DRS candidates were chosen as outlined in chapter 2 with two spatial, two update, and one temporal candidate for each site.

3.4.2 Belief Update Period

Figure 3.3 (b) shows the results of varying the period at which the belief for each node is calculated. These results indicate that performing the belief update at every iteration is optimum. This is in contrast to the literature on BP where typically many iterations of message passing are performed before calculating the belief. This may be due to the nature of the problem here where the candidates at each node change with every update, unlike other BP schemes where the labels at each node are fixed. Refreshing the candidate motion vectors frequently seems to yield better results.

3.4.3 Issues in Evaluation

The essence of the process being investigated here is that of candidate selection for Motion Estimation. Three algorithms are to be compared, the two that have been introduced in this work, and the 3DRS which is a well established algorithm with work from Philips Research by G. de Haan. All the algorithms employ a candidate selection strategy. It is clear that all of the

algorithms discussed can be unified under the Bayesian framework. As far as the manifestation of their operation is concerned therefore, the different strategies arise from different choices for the Likelihood, Prior, and very importantly the optimisation algorithm chosen. All the methods use a Displaced Frame Difference for Likelihood. The prior for the 3DRS technique is different from the algorithms introduced here. However, the optimisation strategy for 3DRS is indeed ICM, a strategy adopted by the ICM algorithm presented here.

The candidate selection strategy plays a large role in the performance of these algorithms. The underlying idea is that the *correct* answer should lie in the candidate set. Increasing the number of candidates would help reinforce this assumption. It is clear that 3DRS can be adapted to use the same candidate set as the algorithms introduced here and vice versa. It is further clear that 3DRS as it is used in Philips hardware will be different from the description outlined here. The comparison therefore will be flawed in the sense that it is not entirely possible to compare like with like to the satisfaction of all observers.

Some compromises must be made to make any progress. Therefore, the 3DRS description adopted is as described in the most up to date literature when this work was started i.e. circa Jan 2004 [20]. It is accepted and understood that much work on the 3DRS algorithm has been published [38, 35, 39, 6] which reflect many of the issues discussed here. None of the results shown can conclude that the precise implementations discussed here are better than the best of 3DRS, but it can be concluded that candidate selection is an effective and efficient mechanism for motion estimation, and that the BP optimisation strategy does not bring orders of magnitude improvement in this particular framework.

3.4.4 M2SE and Smoothness Performance

This section illustrates the M2SE and smoothness measures obtained for the three test sequences. Recall that the M2SE metric is designed to give an indication of how representative of the true motion field the results are. Reliable motion fields are expected to yield good motion compensation and hence low M2SE. In addition reliable motion fields tend to be spatially smooth and hence yield low $\bar{E}_s$. Hence for both measures small values indicate good performance.

Figure 3.4 shows the results for the caltrain sequence. In this case the ICM method gives both lower M2SE and smoother motion fields than the other three methods. The BP performance is between that of ICM and 3DRS. Since the three candidate selection algorithms incorporate explicit smoothness priors the resulting motion fields are much smoother than the maximum-likelihood WBME vectors. This is desirable behaviour as the motion fields should be smooth except at motion boundaries.

Figure 3.5 illustrates the results for the yosemite sequence. Here the 3DRS algorithm provides the lowest M2SE, although the ICM result is very similar. The ICM method gives the smoothest motion field with BP being slightly better than 3DRS. Again all methods give improved performance over the WBME results. Note that problems along the left image boundary

of the BP result are due to implementation issues and not the BP algorithm used.

Finally the results for the garden sequence are shown in Figure 3.6. In terms of M2SE there is little to choose between the three MAP methods. However in smoothness terms, BP and ICM give the best results with the BP method being the slightly better of the two. Further results are provided in Appendix B.

3.4.5 Visual Evaluation of Vector Fields

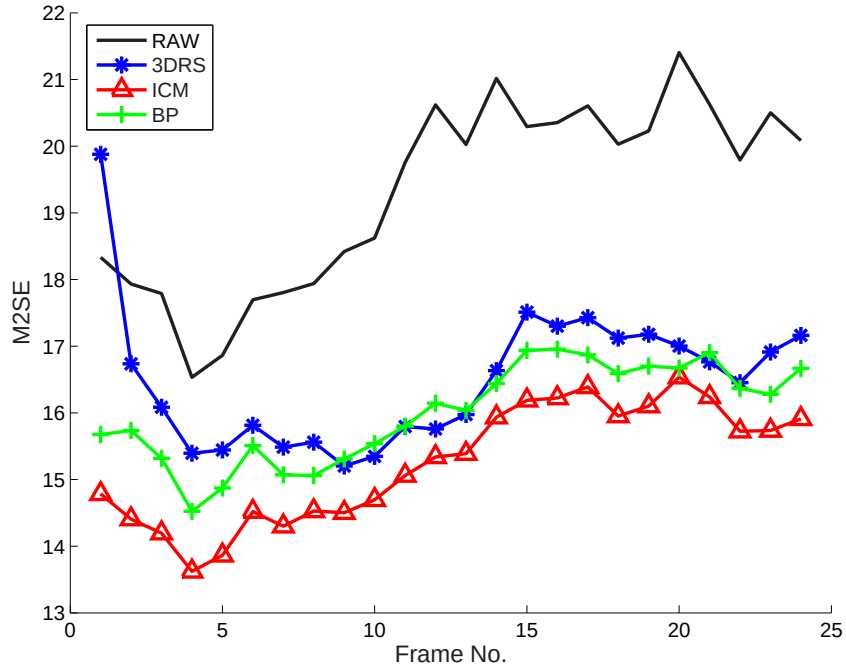
While the M2SE and smoothness results indicate that the ICM algorithm gives the best results, a qualitative analysis of the resulting vector fields gives a different picture. In some cases BP can provide better motion vectors, such as in large areas with low texture. Consider for example the lower left corner of the calender highlighted in Figure 3.7. Although this area is part of the moving calender, due to the low texture in this region it is assigned a zero motion. BP is the only method which successfully estimates motion vectors for this area. It seems that BP is able to propagate vectors over a larger distance than ICM. Over smaller distances, such as the region around the train or at the top left of the frame, ICM is able to successfully estimate the motion. The ball in this sequence illustrates some limitations of the 3DRS algorithm. Since the 3DRS algorithm has a high spatial smoothness constraint, it cannot estimate the motion of the ball correctly and instead assigns it the motion of the train.

In the yosemite sequence, Figure 3.8, the three methods again give better results than the WBME. The region highlighted in the foreground is estimated correctly by the three methods. The ability of the BP algorithm to propagate motion vectors further than ICM is also shown here. In the region highlighted at the top of Figure 3.8 there is no motion. Since there is no texture in this region there is no error due to the likelihood and so the smoothness term dominates the energy minimisation. Since propagating vectors into this region provides a smoother motion field, and hence a lower energy, this is what all three methods try to do. However the BP and 3DRS tend to push vectors further into this region than ICM can.

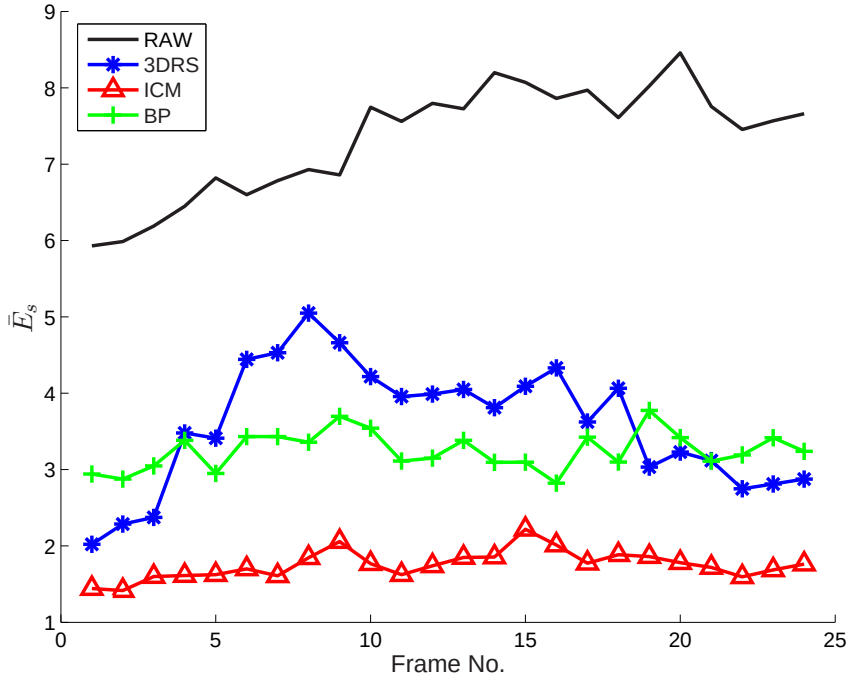
Similar results occur in the background of the garden sequence, particularly in the sky regions to the left and right of the tree, Figure 3.9. Again the BP and 3DRS methods seem to be better able to estimate the motion in these low texture regions than ICM.

3.4.5.1 A note about smoothness

In regions with no texture, the Likelihood in the Bayesian paradigm will yield no information to prefer one vector over another. One of the tenets of the processes described here is that smooth motion fields are to be preferred over discontinuous motion fields. This is encoded into all the priors used. In regions with no texture such as the bottom left hand edge of the calendar both no motion and smooth motion explain that region well over 3 frames. Over more frames however, the ‘no motion’ assumption would fail as background is revealed or obscured. Thus it is sensible that an algorithm yielding a smooth motion field i.e. no motion hole, would be preferable over

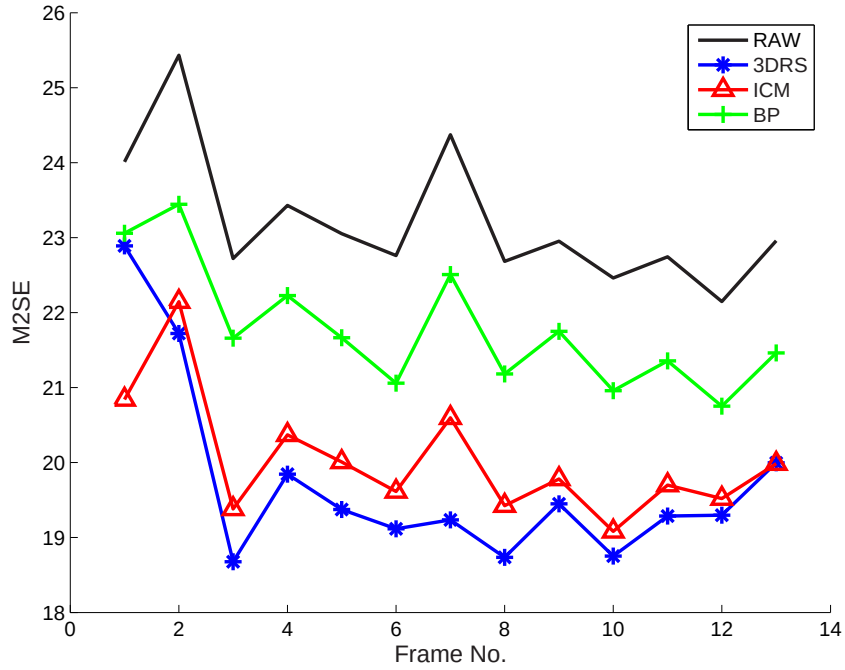


(a)

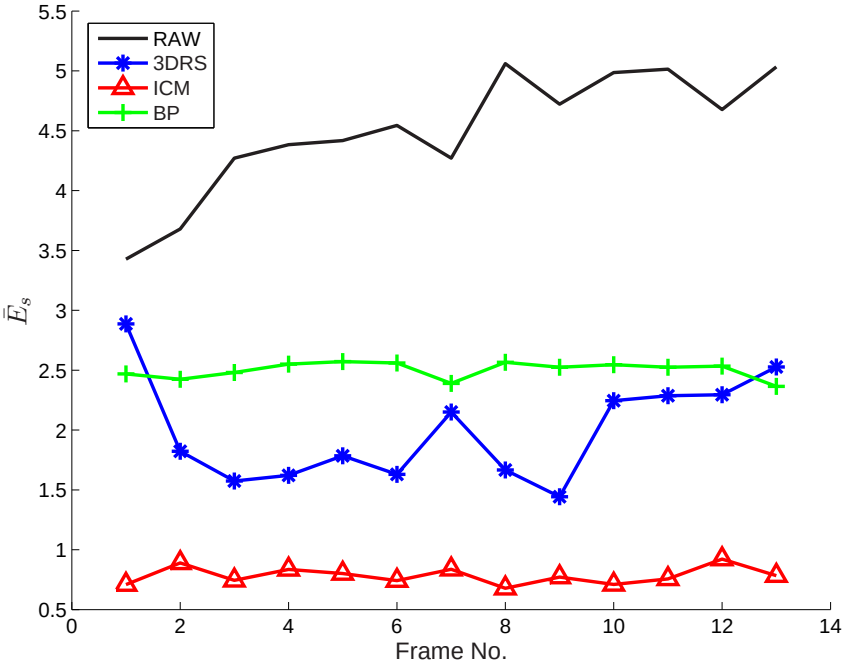


(b)

Figure 3.4: *Caltrain sequence (8x8 Blocks). (a) M2SE for 25 frames of the caltrain sequence. The mean value for the zero estimate is 93. (b) Measure of the Smoothness ($\bar{E}_s$) of the motion field.*

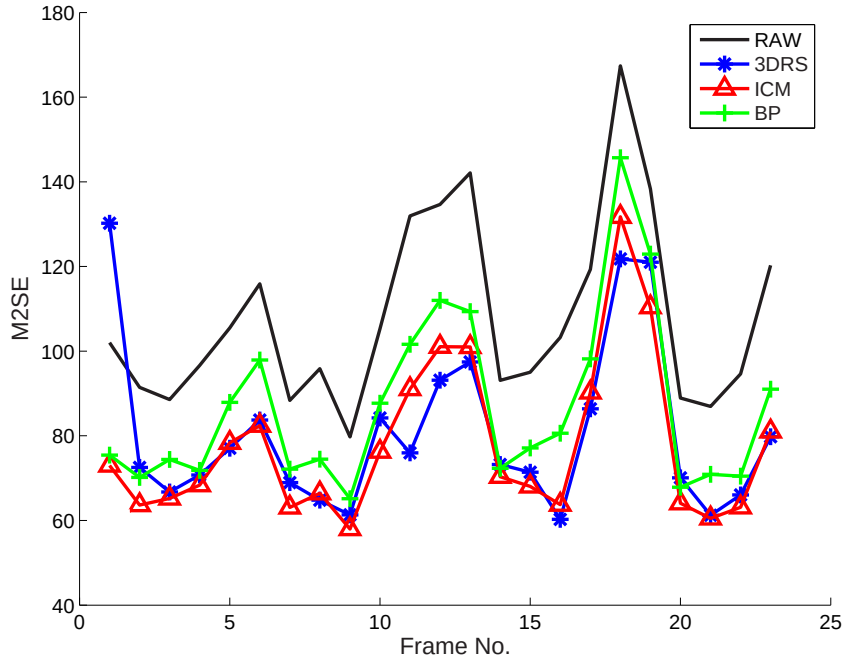


(a)

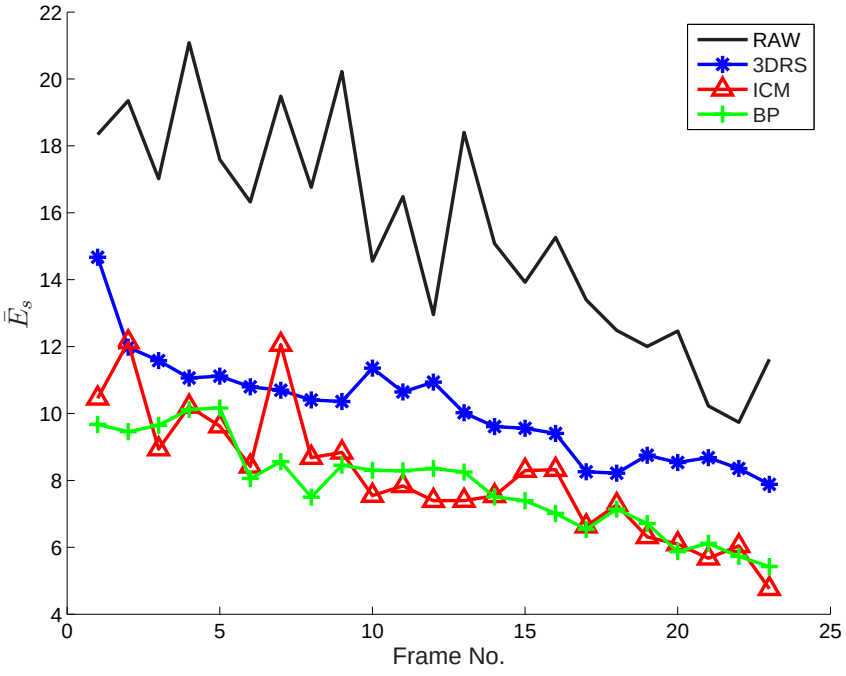


(b)

Figure 3.5: *Yosemite sequence (8x8 Blocks). (a) M2SE for 15 frames of the yosemite sequence. The mean value for the zero estimate is 185. (b) Measure of the Smoothness ($\bar{E}_s$) of the motion field.*



(a)



(b)

Figure 3.6: *Garden sequence (8x8 Blocks). (a) M2SE for 25 frames of the garden sequence. The mean value for the zero estimate is 956. (b) Measure of the Smoothness ($\bar{E}_s$) of the motion field.*

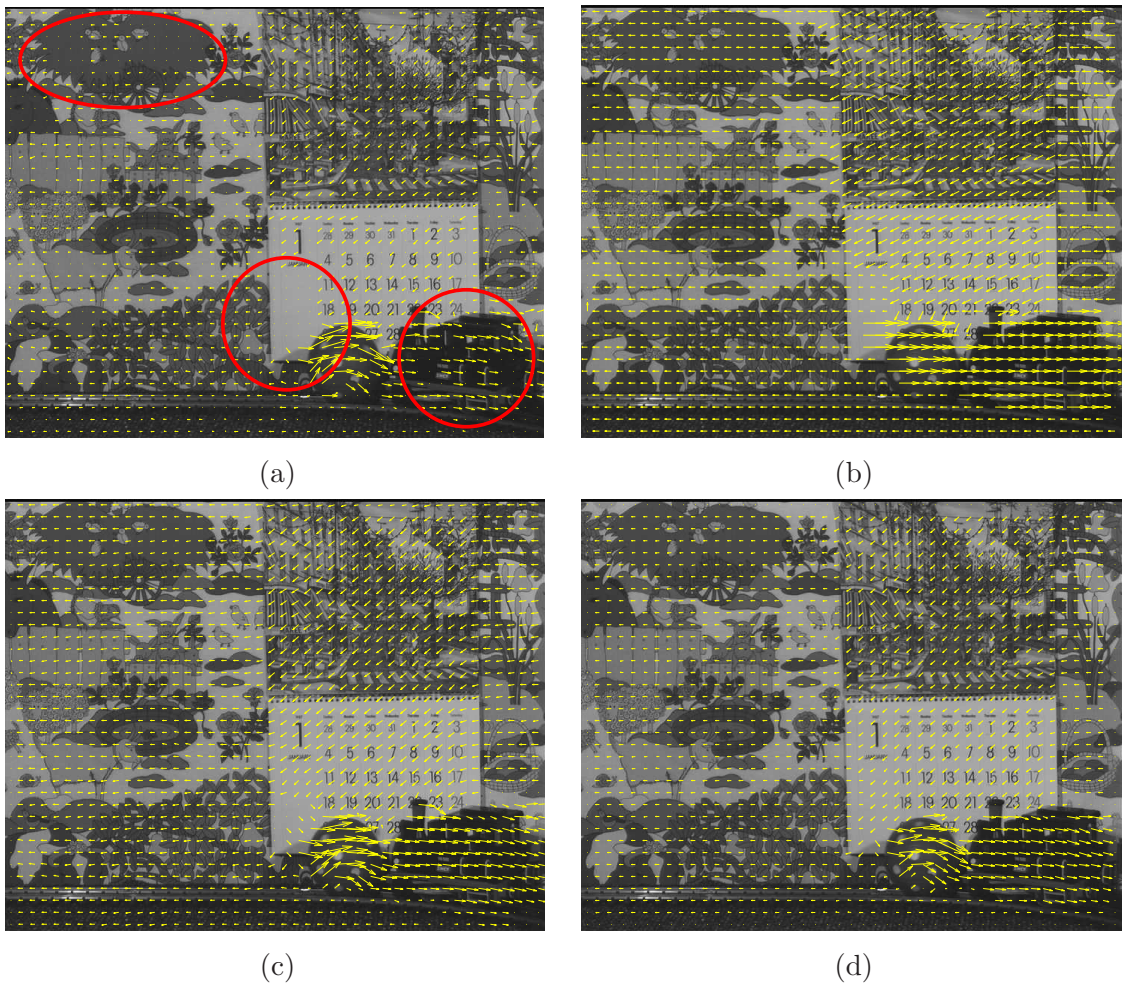
one that leaves the hole. In a sense this is the point about a prior. It may not make correct decisions all the time, but on average it biases the solution to *what is preferred*. The preference is based on experience, and is an acceptable notion that is well established within the Bayesian paradigm. It is accepted that another user might prefer *the hole*. We do not. This position is well in line with academic research in this area beginning with Horn and Schunck, continuing with Black and Anandan and more recently Boykov et al.

3.5 Conclusion

This chapter presented the idea of using candidate selection for motion estimation. While this idea is not new, an attempt was made to unify the method using Bayesian inference and MAP estimation. Two MAP estimation algorithms were investigated: ICM and Belief Propagation. The BP motion estimation algorithm is new and compares quite favourably with the other methods investigated. In fact in some cases BP provides better results than ICM or 3DRS, judging by the resulting motion fields. Also all methods give an improvement in both M2SE and smoothness over the original raw WBME motion vectors.

3DRS is a very efficient algorithm but can have some problems with local motion, such as seen in the caltrain sequence. A good hybrid algorithm might use 3DRS as a good initial guess for either the BP or ICM methods. This is an area for further research. The good performance of the 3DRS algorithm is in part due to its use of temporal candidates. Hence future work should also include extending the ICM and BP algorithms to include temporal candidates in their candidate sets.

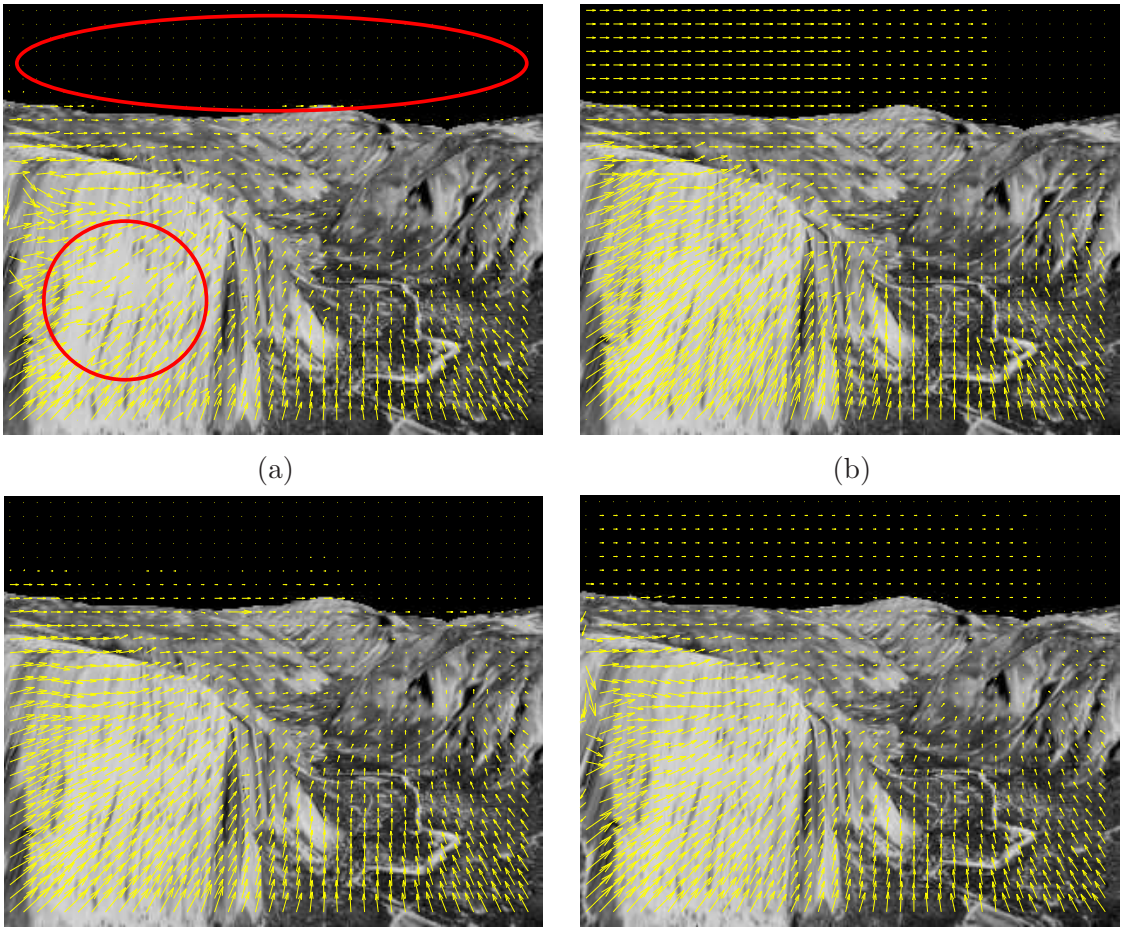
Since motion estimation is essentially a labelling problem, the method of candidate selection deserves further investigation for other labelling problems. It has been successfully used here to reduce the number of motion vectors tested at each site while still providing good MAP estimation performance. It was also shown that this provided much better results than a maximum-likelihood method.



(a) (b)

(c) (d)

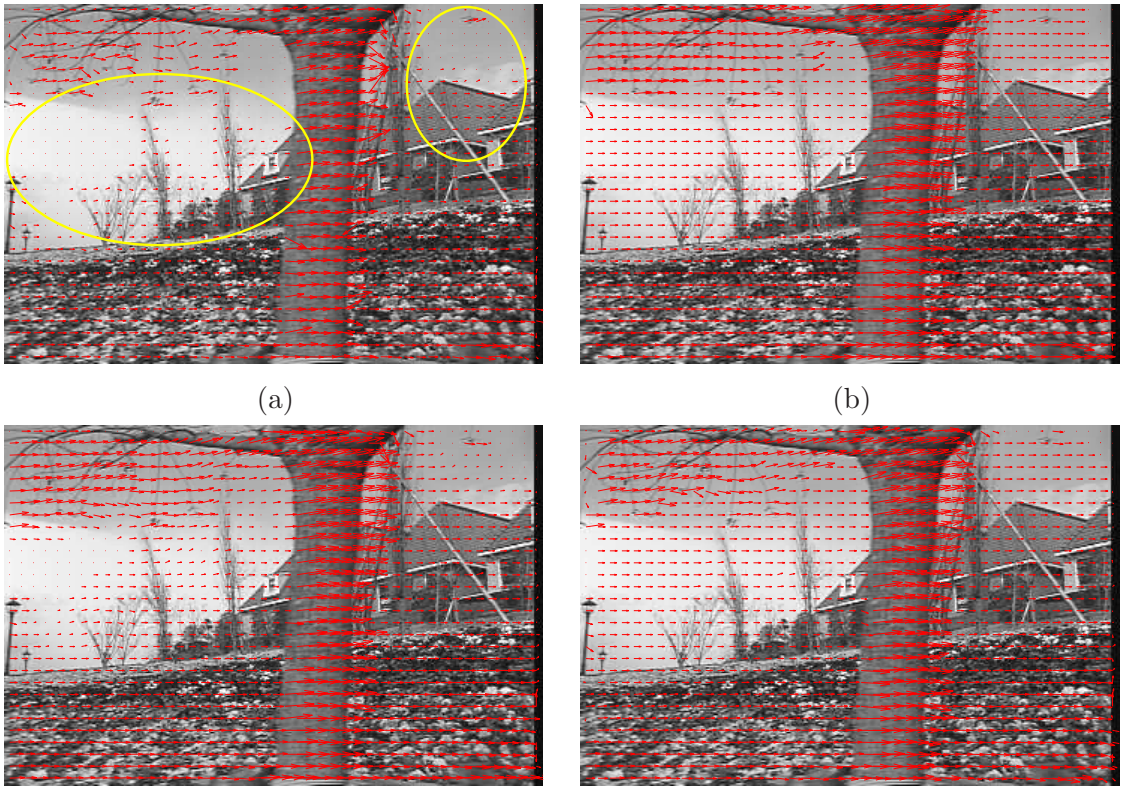
Figure 3.7: *Caltrain Sequence (16x16 Blocks). (a) WBME result (original vectors). (b) 3DRS result. (c) ICM smoothing. (d) BP smoothing. The two algorithms ICM and BP give an significant improvement over the original raw motion vectors. Notice the area to the bottom left of the calender. The ICM algorithm only partially fixes this problem*



(a) (b)

(c) (d)

Figure 3.8: *Yosemite sequence (8x8 Blocks). (a) WBME result. (b) 3DRS result. (c) ICM smoothing. (d) BP smoothing.*



(a) (b)

(c) (d)

Figure 3.9: *Garden sequence (8x8 Blocks). (a) WBME result. (b) 3DRS result. (c) ICM smoothing. (d) BP smoothing.*

4

On-line Global Motion Estimation [1]

THE estimation of the global motion or the camera motion in a scene is a key process in video processing and computer vision applications. In the post-production industry accurate estimation of the camera motion is necessary for convincing visual effects. Applications such as removing unwanted camera shake [1, 169, 56] to rig-removal [103] rely on accurate knowledge of the motion in the scene. Global motion estimation (GME) is usually required as a pre-process in restoration for shake removal [32] before removing other artefacts such as flicker [158].

Image mosaicking is a method used to 'stitch' together multiple photographs of a scene to create an overall panoramic view of that scene. Video mosaicking [180, 33] aims to do something similar with image sequences. It is highly dependent on accurate global motion estimation to correctly register the images before mosaicking. Two examples of video mosaicking are surveillance applications [134] and sports summarisation [106].

GME can also lead to increased efficiency in video coding [45]. Dynamic sprites (DS) and global motion compensation (GMC) are two tools based on GME which have been incorporated into the MPEG-4 video compression standard [140]. Knowledge of the camera motion in a sequence can also be useful in content based information retrieval applications. For example in sports footage it is possible to extract interesting events based on the temporal evolution of the estimated motion parameters [105]. It is often this temporal evolution of the motion and not necessarily the accuracy of the estimate that is important for retrieval applications.

There are many different GME algorithms available in the literature as described previously

[1] Results from this chapter have been published in: Francis Kelly and Anil Kokaram. Online Global Motion Estimation. *In Proceedings of IEE Conference on Visual Media Production (CVMP'05),* November 2005 [96].

in chapter 2. This chapter presents a new global motion estimation algorithm bases on Particle Filters and compares it to an Iteratively Re-weighted Least Squares (IRLS) method. Also a hybrid algorithm combining these two methods is proposed.

4.1 Issues with GME

Certain properties within the image data, such as periodic structure, can lead to ambiguous matching in motion estimation. Some examples of periodic structure might be slates on a roof, a sliding gate or wooden fence etc. Figure 4.1 shows stills from various test sequences used in this work. These sequences can be seen at `http://www.mee.tcd.ie/~sigmedia/publications/publis/cvmp05/`. Figure 4.2 (c) shows an example of an artificial sequence with periodic structure (checkerboard pattern) in the image data. To demonstrate the problem, a motion estimate using a gradient based least squares algorithm [105] is compared with the true motion. As can be seen by Figure 4.3 (a), the Least Squares (LS) global motion estimates vary widely about the true camera motion. LS estimation can also fail if the sequence has more complex camera motions with large displacements. Figure 4.2(a) shows a frame from an artificial Lena sequence undergoing zoom and pan. In this case the zoom is estimated correctly but the pan is not, Figure 4.3 (b).

Research on early visual processes indicate that the human visual system may be based on gradient based motion estimation [54]. However when viewing the artificial sequence above it is still possible for humans to correctly identify and track the motion. This might[2] be due in part to the brain's ability to learn from the previous motion of the scene and perhaps a bias toward slow motion. In cases where LS fails it may be helpful to include temporal information via some history of previous motion estimates. For real-time algorithms it would also be useful if these estimates could be generated sequentially as new information becomes available on-line.

As already noted GME is necessary for video stabilisation. Two main types of unsteadiness or shake which arise in video are considered here: random shake and impulsive shake. Random shake is typically associated with hand-held cameras and may occur over long periods of the sequence. Impulsive shake usually only affects a few frames of the sequence and may be caused for example by a poor splice in a film reel. Figure 4.3 (c) and (d) show examples of LS estimation of the vertical translation component for sequences with random and impulsive shake respectively. Example frames from the corresponding sequences are shown in Figure 4.1 (b) and (d).

Stabilising these two types of shake generally requires different approaches. Compensating for random shake typically involves smoothing the motion estimates to reduce the 'jerkiness' of the viewed output while leaving the underlying global motion [32]. However if this method is applied to a sequence with impulsive shake it can disrupt frames either side of the shake, depending on the smoothing filter used. This can leave a visible artefact in the compensated video. Instead one method used is to try and 'lock' the global motion of the scene to give a

[2]It is accepted that this is a psychovisual question and we have done no experiments to test this.

steady output. By incorporating history of previous motion estimates it should be possible to diagnose the type of shake present in a sequence and apply the correct stabilisation algorithm accordingly. In the next section, the image sequence model used here is presented.

4.2 Global Motion Estimation

Recall from chapter 2 the following image sequence model for global motion estimation

$$I_n(\mathbf{x}) = I_{n-1}(F(\mathbf{x}, \Theta_n)) + \epsilon(\mathbf{x}) \tag{4.1}$$

I_n is the intensity function for the image at frame n, $\mathbf{x} = [x, y]^T$ is the spatial coordinate vector within a frame, and $\epsilon(\mathbf{x}) \sim \mathcal{N}(0, \sigma_\epsilon^2)$ represents any error in the model and is assumed to be Gaussian noise. The vector function $F(\mathbf{x}, \Theta_n)$ represents the transformation of image coordinates caused by the motion between frames I_n and I_{n-1}. To represent motion such as zooming, rotation, and translation, a six parameter affine transformation is used here

$$F(\mathbf{x}, \Theta_n) = A\mathbf{x} + \mathbf{d} \tag{4.2}$$

where A is a 2×2 affine transformation matrix and $\mathbf{d}$ is the displacement vector.

$$\begin{aligned} F(\mathbf{x}, \Theta_n) &= A\mathbf{x} + \mathbf{d} \\ &= \begin{bmatrix} a_1 & a_2 \\ a_3 & a_4 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} d_x \\ d_y \end{bmatrix} \\ &= \begin{bmatrix} a_1 x + a_2 y + d_x \\ a_3 x + a_4 y + d_y \end{bmatrix} \\ &= B(\mathbf{x})\Theta_n \end{aligned} \tag{4.3}$$

where

$$B(\mathbf{x}) = \begin{bmatrix} x & y & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x & y & 1 \end{bmatrix} \tag{4.4}$$

Θ_n is the vector formed by the motion parameters

$$\Theta_n = [a_1, a_2, d_x, a_3, a_4, d_y]^T \tag{4.5}$$

In the case of a translational only motion model the parameter vector is given by $\Theta_n = [d_x, d_y]^T = \mathbf{d}$, and $F(\mathbf{x}, \Theta) = \mathbf{x} + \mathbf{d}$.

From (4.1) it can be useful to define the difference between the current frame and the motion compensated previous frame, denoted Displaced Frame Difference (DFD) as

$$DFD(\mathbf{x}, \Theta) = I_n(\mathbf{x}) - I_{n-1}(F(\mathbf{x}, \Theta)) \tag{4.6}$$

As outlined in chapter 2, many GME algorithms aim to minimise some function of the DFD. One popular method considered here is the IRLS algorithm. However these algorithms are typically implemented off-line. The following section introduces a new on-line GME algorithm. This is based on a Bayesian filtering technique known as Particle Filters [4, 41, 124, 42], which is introduced first.

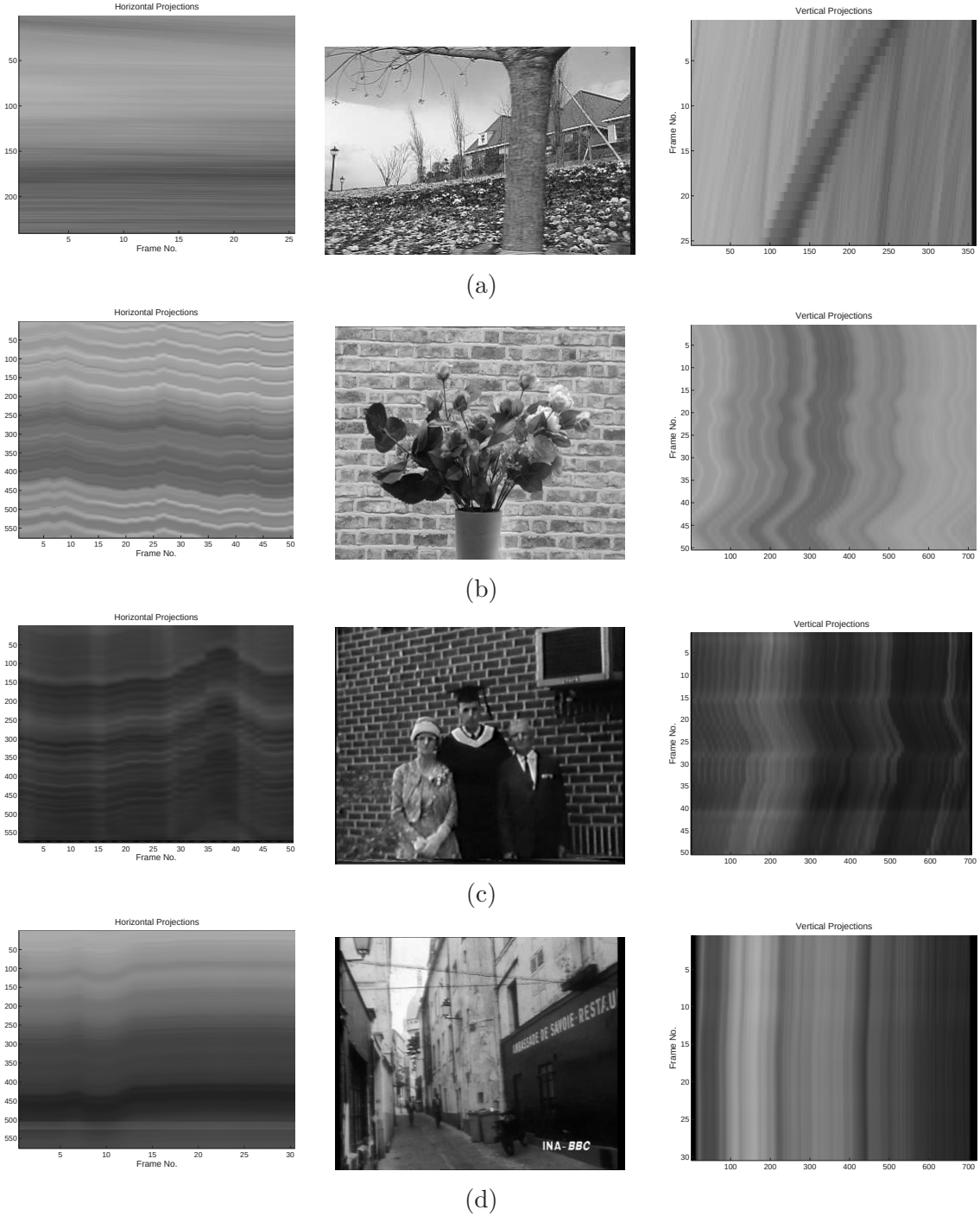
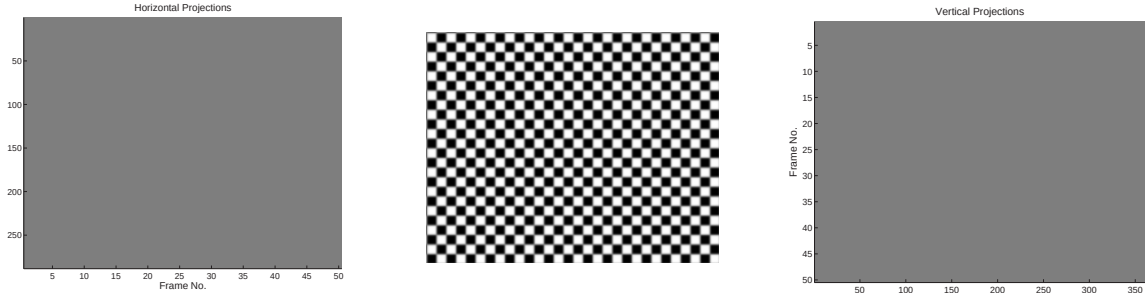


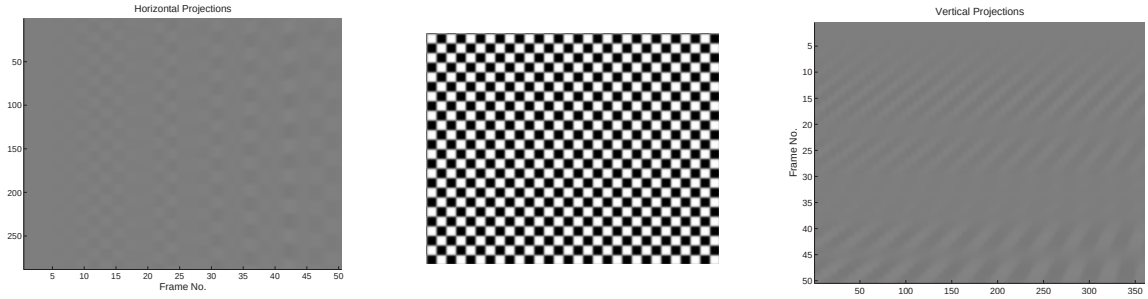
Figure 4.1: *Stills and TIP images from test sequences used in this work. (a) Garden test sequence (pan). (b) Flowers sequence shot with hand held camera (random shake). (c) Donegal home movie (random shake). (d) Film clip with poor splice (impulsive shake).*

(a)

(b)



(c)



(d)

Figure 4.2: *Stills and TIP images from test sequences used in this work. (a) Artificial Lena undergoing pan only. (b) Artificial Lean undergoing zoom and pan. The solid red line represents the true value. (c) Artificial checkerboard undergoing pan only. (d) Artificial checkerboard undergoing zoom and pan.*

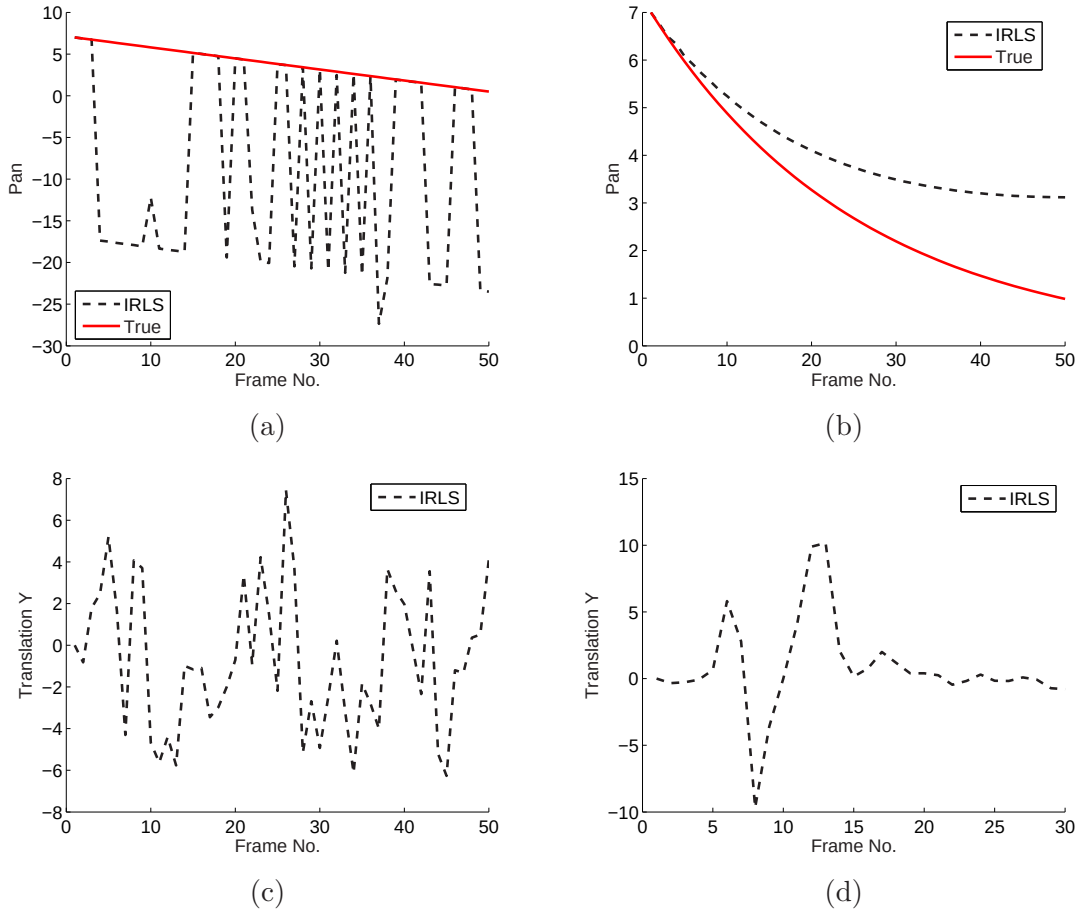


Figure 4.3: *IRLS parameter estimation on artificial and real sequences. (a) Estimated pan for artificial checkerboard sequence. (b) Estimated pan for artificial Lena sequence which contains zoom and pan. (c) Estimated vertical translation for real sequence with random shake. (d) Estimated vertical translation for real sequence with impulsive shake. Black dashed = IRLS. Red = True value.*

4.3 Bayesian Filtering

Consider a state space model for a system

$$\begin{aligned} \Theta_n &= f(\Theta_{n-1}, v_{n-1}) \\ I_n &= h(I_{n-1}, \Theta_n, u_n) \end{aligned} \tag{4.7}$$

where the image data, I_n, I_{n-1}, are the available observations and the motion parameters Θ_n the required states of the system. $f(\cdot)$ is a possibly non-linear function of state Θ_{n-1}, with process noise v_{n-1}, and models the expected behaviour of the system over time. Observations at time n are obtained from another possibly non-linear measurement function, $h(\cdot)$, with observation noise u_n, which is the source of information about the system. The objective of tracking is to recursively estimate the unknown state Θ_n given the measurements I_n.

Assume the states follow a first order Markov process,

$$p(\Theta_n|\Theta_{n-1}, \Theta_{n-2}, \ldots, \Theta_0) = p(\Theta_n|\Theta_{n-1})$$

and the observations are independent given the states, $p(I_n|\Theta_n, A) = p(I_n|\Theta_n)$. All relevant information on $\{\Theta_0, \Theta_1, \ldots, \Theta_n\}$ at time n is included in the posterior density

$$p(\Theta_0, \Theta_1, \ldots, \Theta_n|I_0, I_1, \ldots, I_n) = p(\bar{\Theta}_n|\bar{I}_n)$$

In probabilistic tracking it may be necessary to sequentially estimate the states of a system as a set of observations become available on-line. The distribution of interest is often a marginal of the posterior, also known as the filtering distribution [41],

$$p(\Theta_n|\bar{I}_n) = p(\Theta_n|I_0 \ldots I_n) \tag{4.8}$$

Doing so reduces the memory requirements of the filter as only Θ_n^i needs to be stored; the path $\Theta_{0:n-1}^i$ and the history of observations $I_{0:n-1}$ can be discarded. Also the states at each time step n can be processed sequentially as new information becomes available on-line. This problem is known as the Bayesian filtering problem. If the filtering density is available, a number of estimates of the system state at time n may be calculated including

- Optimal MSE estimate of state (or mean):

$$\hat{\Theta}_n = E[\Theta_n|\bar{I}_n] = \int \Theta_n p(\Theta_n|\bar{I}_n) d\Theta_n \tag{4.9}$$

- Maximum a posteriori (MAP):

$$\hat{\Theta}_n = \arg\max_{\Theta_n} p(\Theta_n|\bar{I}_n) \tag{4.10}$$

A recursive filtering approach means that received data can be processed sequentially rather than as a batch. It also means that it is not necessary to keep track of the complete history of the states of the system. In sequential Bayesian estimation the filtering distribution may be computed recursively using two stages: prediction and update.

1. **Prediction**: The prior for the current time step, $p(\Theta_n|\bar{I}_{n-1})$ is computed by propagating the posterior from the previous time step according to a state transition density $p(\Theta_n|\Theta_{n-1})$

$$p(\Theta_n|\bar{I}_{n-1}) = \int p(\Theta_n|\Theta_{n-1})p(\Theta_{n-1}|\bar{I}_n)d\Theta_{n-1} \tag{4.11}$$

   The transition density or state evolution model is defined by the system equation.

2. **Update**: Upon receiving a new observation, I_n, the posterior may be estimated through a direct application of Bayes rule

$$p(\Theta_n|\bar{I}_n) = \frac{p(I_n|\Theta_n)p(\Theta_n|\bar{I}_{n-1})}{p(I_n|\bar{I}_{n-1})} \tag{4.12}$$

In general this recursive propagation of the posterior density cannot be determined analytically. In a small number of restrictive cases solutions do exist. Probably the best known of these is the Kalman filter [90, 198] for linear Gaussian likelihood and state evolution models. For non-linear models approximation techniques for determining the filtering density are required.

4.3.1 Sequential Importance Sampling

Sequential Monte Carlo methods, also known as Particle Filters, are a numerical approximation for recursive estimation of a filtering density. They have become very popular in recent times due to their ability to deal with non-linear models and their reasonably simple implementation. The Sequential Importance Sampling (SIS) algorithm is a sequential Monte Carlo method that forms the basis for most recent sequential MC filters [4].

The basic idea is to represent the required posterior density function by a set of discrete samples (or particles) $\{\Theta_n^i\}_{i=1}^{N_s}$ with associated importance weights $\{w_n^i\}_{i=1}^{N_s}$, and to compute estimates based on these samples and weights. The samples are randomly selected from state space and the weights are based on a likelihood model and the principle of importance sampling. The approximated posterior may be thought of as a randomly sampled weighted approximation of the true posterior, $p(\Theta_n|\bar{I}_n)$.

Let $\{\Theta_{n-1}^i, w_{n-1}^i\}_{i=1}^{N_s}$ be a weighted set of N_s samples approximately distributed according to $p(\Theta_{n-1}|\bar{I}_{n-1})$ at time $n-1$. New samples, at time n, are generated from a suitable proposal distribution called the importance density, $q(\cdot)$

$$\Theta_n^i \sim q(\Theta_n|\Theta_{n-1}^i, I_n), i = 1 \ldots N_s \tag{4.13}$$

The weights are then updated according to

$$w_n^i = w_{n-1}^i \frac{p(I_n|\Theta_n^i)p(\Theta_n^i|\Theta_{n-1}^i)}{q(\Theta_n^i|\Theta_{n-1}^i, I_n)} \tag{4.14}$$

The new particle set $\{\Theta_n^i, w_n^i\}_{i=1}^{N_s}$ is then approximately distributed according to $p(\Theta_n|\bar{I}_n)$. The posterior density may be approximated as

$$p(\Theta_n|\bar{I}_n) \approx \sum_{i=1}^{N_s} w_n^i \delta(\Theta_n - \Theta_n^i) \tag{4.15}$$

It can be shown that as $N_s \to \infty$, the approximation given by (4.15) approaches the true posterior density $p(\Theta_n|\bar{I}_n)$ [4]. SIS then consists of the following steps as each measurement is received sequentially. For each sample $i = 1 : N_s$:

1. **Predict** new samples for time n by drawing samples from the proposal distribution $\Theta_n^i \sim q(\Theta_n|\Theta_{n-1}^i, I_n)$.

2. **Update** the sample weight, w_n^i, according to equation 4.14.

4.3.2 Sampling Importance Resampling

Over time the SIS algorithm can lead to degeneracy in the sample set used to represent the distribution of interest. After a few iterations of the algorithm, all but one of the normalised importance weights may be very close to zero. This is because the unconditional variance of the importance weights increases over time [41]. A resampling step is introduced which aims to reduce the variance of the importance weights. Resampling aims to eliminate particles that have small weights and to concentrate on particles with high weights. The resampling step involves generating a new particle set $\{\Theta_n^{i*}\}_{i=1}^{N_s}$ by resampling with replacement N_s times from $\{\Theta_n^i, w_n^i\}_{i=1}^{N_s}$. The weights for the new sample set are set to $\{w_n^i = 1/N_s\}_{i=1}^{N_s}$. Different resampling strategies lead to different particle filters.

One such strategy is the Sampling Importance Resampling (SIR) algorithm. In the SIR particle filter the proposal function $q(\cdot)$ is chosen to be equal to the prior $p(\Theta_n|\Theta_{n-1})$ and resampling occurs at every time step. This means (4.14) simplifies to $w_n^i = p(I_n|\Theta_n^i)$, with the weights dependent only on the likelihood. This is also known as the bootstrap filter. A well known example for tracking is the CONDENSATION algorithm [83].

4.4 Incorporating history into GME

A relatively simple approach to incorporating history into the GME process is to pose the problem in such a way as to yield a Particle Filter solution. In GME the unknown states are the motion parameters Θ_n and the observations are the image data I_n, I_{n-1}. The idea is to compactly represent the notion that current estimates for motion Θ_n must agree in some way with previous estimates Θ_{n-1} in past frames. In a Bayesian framework the current estimate for Θ_n may be modelled using the following conditional probability

$$p(\Theta_n|I_n, I_{n-1}, \Theta_{n-1}) \propto p(I_n, I_{n-1}|\Theta_n)p(\Theta_n|\Theta_{n-1}) \tag{4.16}$$

where the spatial coordinate vector $\mathbf{x}$ has been dropped for clarity. It is the prior term $p(\Theta_n|\Theta_{n-1})$ that incorporates the dependence of current estimates on previous solutions.

Likelihood: To limit the effect of local motion on the global motion estimate a robust error function is introduced. It treats DFD values arising from local motion as outliers and is noted here as the Robust DFD (RDFD)

$$RDFD = \begin{cases} |DFD|, & \text{if } |DFD| \leq t \\ t, & \text{if } |DFD| > t \end{cases} \tag{4.17}$$

where t is some threshold. The threshold is derived using the same robust histogram approach outlined earlier [45]. Again a value of $T = 10\%$ is used for finding t. Using (4.17) and (4.6), a robust likelihood may be written as

$$p(I_n, I_{n-1}|\Theta_n) \propto \exp\left(\frac{-\sum RDFD}{2\sigma_e^2}\right) \tag{4.18}$$

Prior: The prior on the estimate depends on the model chosen for the temporal evolution of camera motion in the scene. A simple linear model is shown below

$$\Theta_n = \Lambda_n \Theta_{n-1} + \epsilon_\Theta \tag{4.19}$$

This model encourages the temporal evolution of the motion parameters to follow a first order Auto Regressive process. If $\Lambda_n = 1$ then the current motion is expected to be a noisy version of the previous motion. Thus the prior is given by

$$p(\Theta_n|\Theta_{n-1}) \propto \exp\left(\frac{-\sum(\Theta_n - \Lambda_n\Theta_{n-1})^2}{2\sigma_\Theta^2}\right) \tag{4.20}$$

$\Lambda = diag(\lambda_z, \lambda_r, \lambda_x, \lambda_r, \lambda_z, \lambda_y)$ are variables related to the temporal smoothness of the motion parameters. λ_z, λ_r are variables based on zoom and rotation respectively, and λ_x, λ_y are variables based on translation. These are set adaptively with $\Lambda_n = \frac{\Theta_{n-1}}{\Theta_{n-2}}$.

Using the Particle Filter gives an elegant solution to this problem. By initialising a number of possible motion estimates in the first frame, it is possible to use the measured distribution of these *particles* in successive frames as an estimate for the posterior $p(\Theta_n|I_n, I_{n-1}, \Theta_{n-1})$. A key step in designing the particle filter is the choice of something called the proposal density. Here the proposal density $q(\cdot)$ is chosen as

$$q(\Theta_n|\Theta_{n-1}, \dots) = p(\Theta_n|\Theta_{n-1}) \tag{4.21}$$

This leads to the straightforward bootstrap filter implementation which proceeds as outlined below, with sampling, importance weighting and resampling at each time step. This version of the algorithm is denoted as the GME particle filter (GMEPF) in subsequent discussion.

1. For $i = 1, \dots, N_s$, sample $\Theta_n^i \sim q(\Theta_n|\Theta_{n-1}^i, \cdots)$.

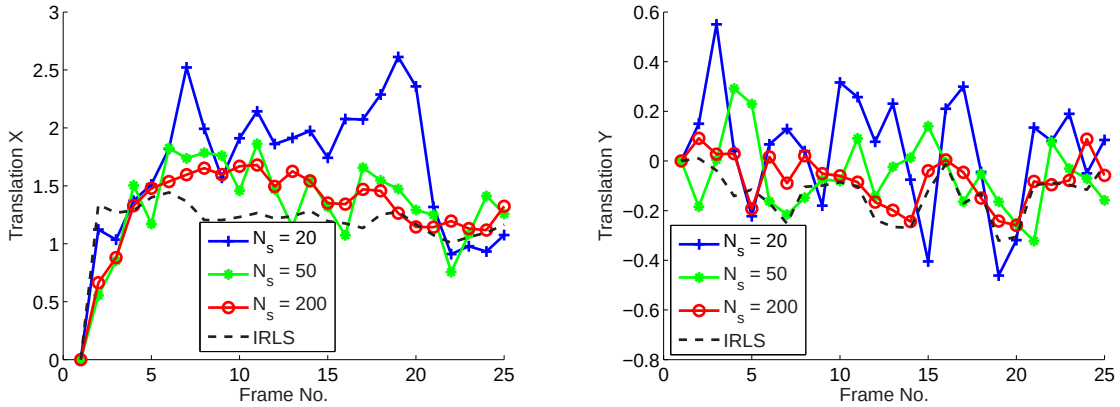


Figure 4.4: *GMEPF for garden sequence. As the number of samples, N_s increases the particle filter estimate approaches the IRLS estimate.*

2. For $i = 1, \ldots, N_s$, evaluate the importance weights w_n^i according to (4.14).

3. For $i = 1, \ldots, N_s$, resample with replacement the particles Θ_n^i, with probability based on the weights w_n^i.

4.4.1 Results using GMEPF

In all the results that follow the particle filter result is chosen as the maximum a posteriori (MAP) estimate. Figure 4.4 demonstrates the performance of the GMEPF on a real sequence. In this case it is the well known garden test sequence which contains a pan from left to right, Figure 4.1 (c). It can be seen that as the number of particles increases the GMEPF gets closer to the IRLS estimate. However the IRLS method still provides the best results.

Figures 4.5 and 4.6 compare results from GMEPF to IRLS for the artificial checkerboard sequence of Figure 4.1 (a). In this sequence only the pan parameter is changing significantly. Again as the number of particles, N_s increases, the estimate gets closer to the true motion. Compare this to the IRLS estimates which vary widely about the true motion, because of ambiguity in the matching process due to the image data. The history in the particle filter encourages the estimates to be temporally smooth.

Note however that this is dependent on the initialisation of the particle filter. Due to the periodic nature of the image data in this sequence, there will be multiple estimates which yield low Likelihoods. In this case the particle filter was initialised to zero and so successfully locks on to the correct Pan starting at ≈ 5 pels. If however the particle filter was initialised further away, say at ≈ -15 pels, then the GMEPF would probably follow a trajectory starting from this point and give the wrong estimate.

These results indicate that the PF is sensitive to its initial conditions. However since these initial conditions are not known a-priori it is not reasonable to ask for it to be initialised close to

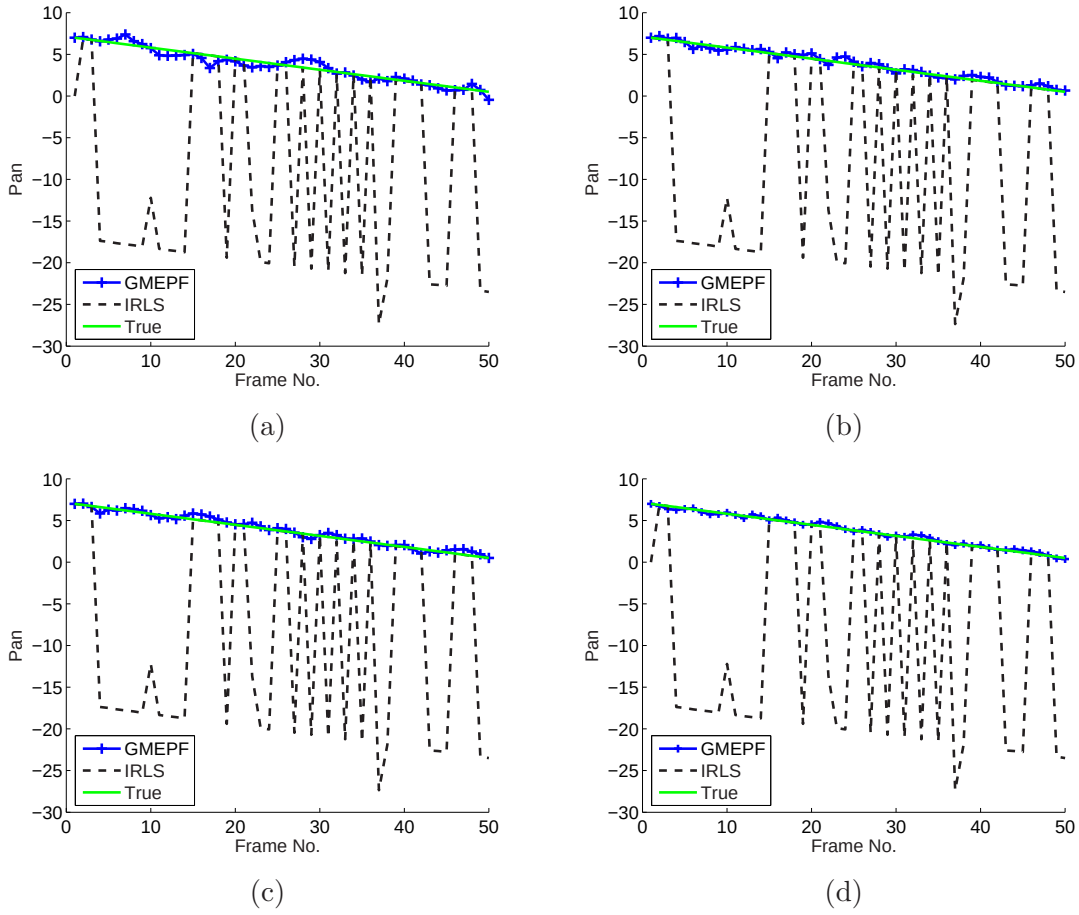


Figure 4.5: *Estimated pan with GMEPF on checkerboard sequence which contains only pan. (a) $N_s = 20$. (b) $N_s = 50$. (c) $N_s = 100$. (d) $N_s = 200$. Blue = GMEPF. Black = IRLS. Green = True Value.*

an unknown. Possible solutions to this problem include choosing a standard initial guess (zero was used here) or using some other method to give an initial estimate (discussed in the next section).

Figure 4.7 show results for the artificial Lena sequence in Figure 4.1 (b) undergoing both zoom and pan. In this case the GMEPF estimates are quite poor and increasing the number of particles does not seem to make any noticeable difference. Note that the while the IRLS method is unable to correctly estimate the zoom either, it does provide better results than the GMEPF. Figure 4.8 (a) and (b) shows the result for GMEPF estimates on the checkerboard sequence undergoing zoom and pan. Again the GMEPF estimate is not exact. However it does demonstrates good temporal smoothness compared to the IRLS estimate.

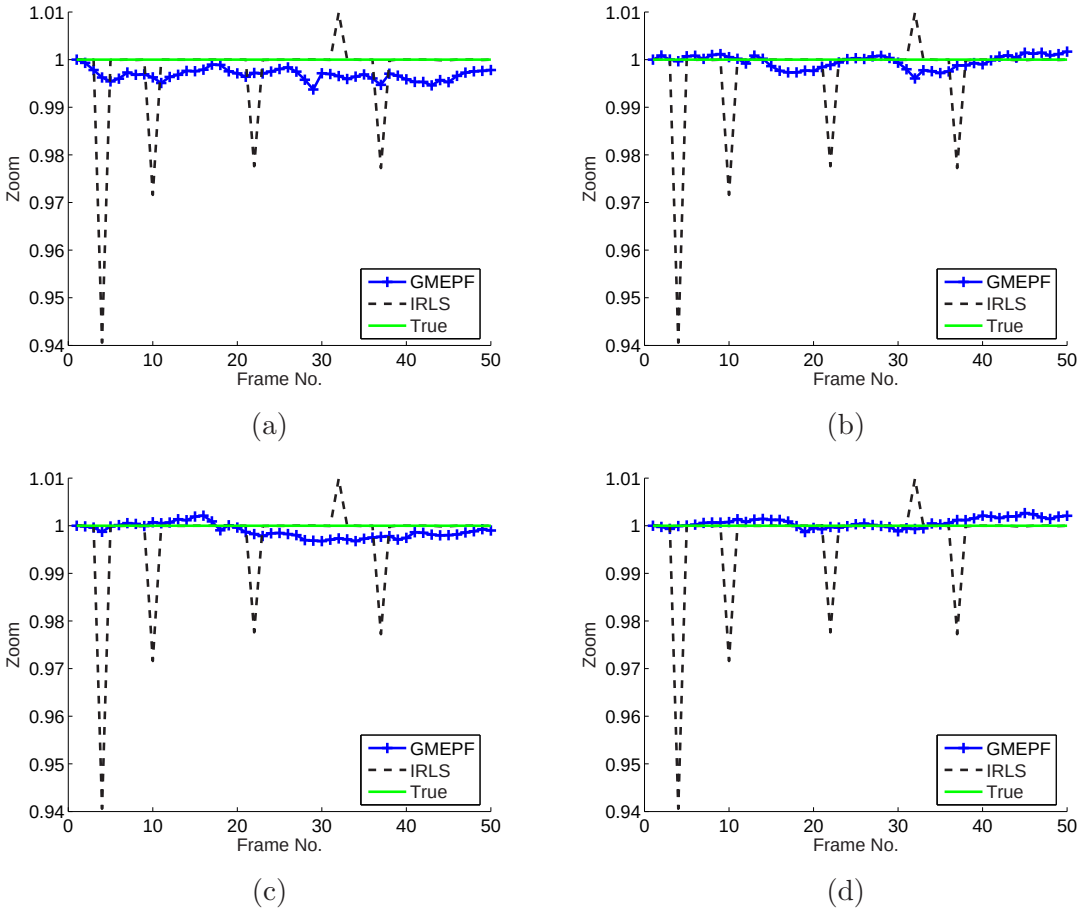


Figure 4.6: *Estimated zoom with GMEPF on checkerboard sequence which contains only pan. (a) $N_s = 20$. (b) $N_s = 50$. (c) $N_s = 100$. (d) $N_s = 200$. Blue = GMEPF. Black = IRLS. Green = True Value.*

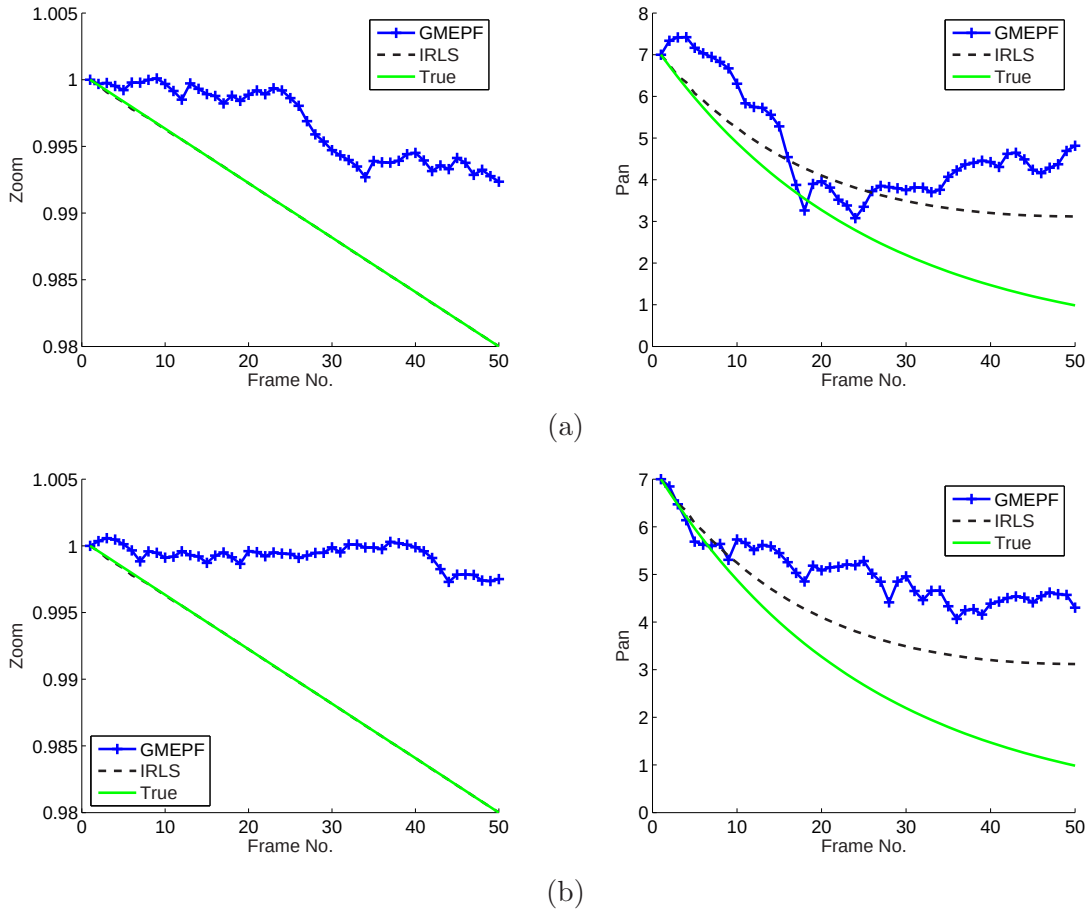


Figure 4.7: *GMEPF on artificial Lena sequence which contains zoom and pan. (a) $N_s = 50$. (b) $N_s = 200$. Blue = GMEPF. Black = IRLS. Green = True Value.*

4.5 Improved IRLS estimation

From these initial tests it is clear that for real sequences IRLS gives the best results. However, the performance of the GMEPF is encouraging, particularly its ability to provide temporal smoothness. Therefore, it may be useful to include estimates from the IRLS method in the particle filter. In this way when the IRLS estimate is a good one the particle filter will be biased towards this estimate. However in cases where the IRLS fails, the history built into the particle filter should push the particle filter towards the correct answer. The history and motion model in the particle filter should encourage temporal smoothness of the estimates.

IRLS Proposal: To do this a proposal density similar to that in [148] is introduced. The proposal density is given by the following mixture model

$$q_{LS}(\Theta_n|\Theta_{n-1},\dots) = \alpha q(\Theta_n|\Theta_{n-1}) + (1-\alpha)p(\Theta_{LS}) \tag{4.22}$$

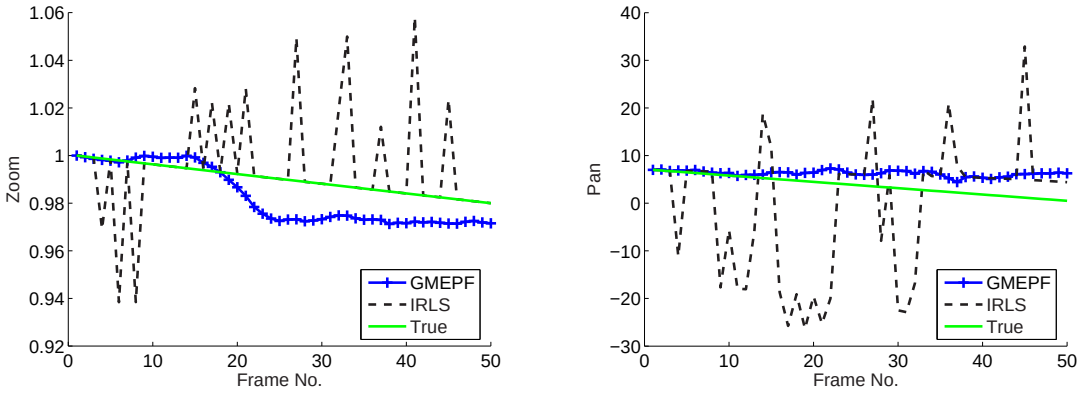
where q is the proposal density from before and Θ_{LS} is the current IRLS guess for Θ_n. The parameter α can be set dynamically without affecting the convergence of the particle filter. When $\alpha = 1$, the algorithm reduces to the GMEPF algorithm. This algorithm is called the IRLS particle filter (IRLSPF). In practice the samples for Θ_n^i are chosen as follows

$$\Theta_n^i \sim \begin{cases} q(\Theta_n|\Theta_{n-1}), & i = 1,\dots,N_s-1 \\ p(\Theta_{LS}), & i = N_s \end{cases} \tag{4.23}$$

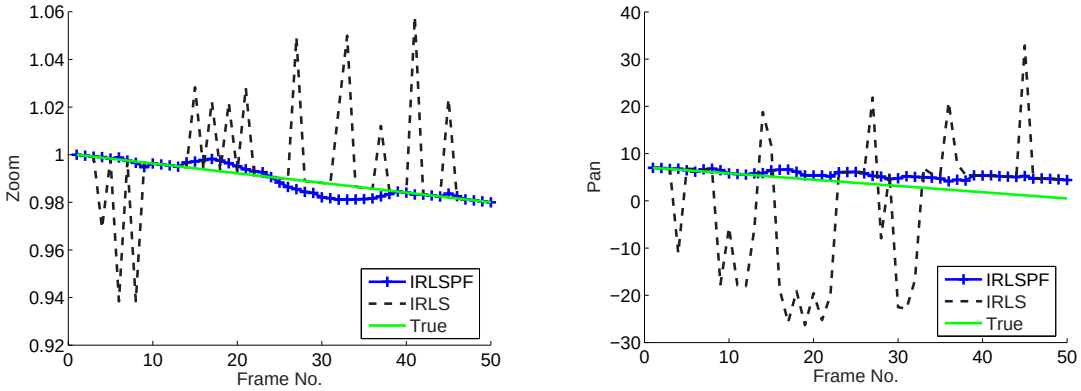
4.5.1 Results using IRLSPF

Figure 4.8 compares the GMEPF with the IRLSPF for estimating zoom and pan on the checkerboard sequence undergoing both zoom and pan. Here, when the IRLS estimate is good it biases the particle filter towards the true motion. When the IRLS is wrong, the history in the particle filter encourages temporal smoothness in the estimated motion. Figure 4.9 shows the results of estimating zoom and pan with IRLSPF for the checkerboard sequence which is undergoing pan only. Again the particle filter is guided by the IRLS estimate, but also preserves the temporal smoothness of the estimate.

Figure 4.10 shows the performance of IRLSPF on some real sequences. Figure 4.10 (a) is the result for the garden test sequence. With only 20 particles the IRLSPF matches the IRLS estimate quite closely. Figure 4.10 (b), (c), and (d) show examples of sequences with random and impulsive shake. For random shake the particle filter cannot track the shake but instead prefers a smoother estimate of the motion. Also shown is the difference between the IRLS and the IRLSPF estimates. It is clear in this case that as the difference signal is noisy across the whole sequence, the underlying shake is random. Contrast this to the impulsive shake case. Here the IRLSPF closely matches the IRLS estimate for the first few frames until the shake occurs. Then the IRLSPF prefers a smooth motion rather than tracking the noisy shake. However once the shake is over the IRLSPF settles back down to match the IRLS estimate. Looking at the



(a)



(b)

Figure 4.8: *Estimated zoom and pan on checkerboard sequence which contains zoom and pan. Blue = Particle Filter. Black = IRLS. Green = True value. (a) GMEPF, $N_s = 20$. (b) IRLSPF, $N_s = 50$.*

difference between the two estimates it is possible to diagnose this as impulsive rather than random shake.

4.6 Final Comments

In general the IRLS is effective at estimating the global motion in a scene. However in certain cases the image data can lead to ambiguity in the motion estimate. This chapter proposed a method for using the temporal information of the motion via a particle filter to try and diagnose such cases. In cases where only one of the motion parameters is changing significantly the particle filter can correct the IRLS estimate with very few particles (20-50). However in cases where more than one parameter has to be estimated then the particle filter can fail. There are particle filter methods which aim to solve this problem by partitioning the parameter space and estimating the parameters one at a time. This is known as partitioned sampling [122] and

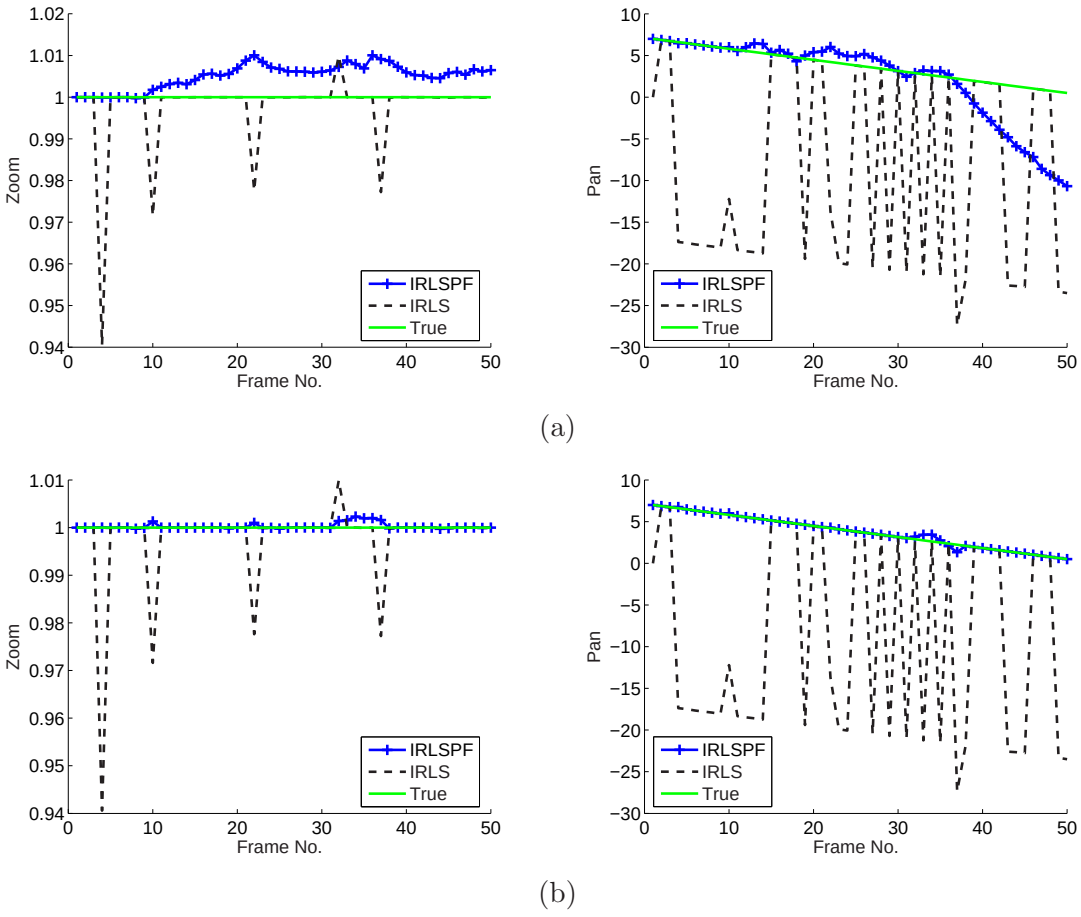


Figure 4.9: *Estimated zoom and pan with IRLSPF on checkerboard sequence which contains only pan. (a) $N_s = 20$. (b) $N_s = 50$. Blue = IRLSPF. Black = IRLS. Green = True value.*

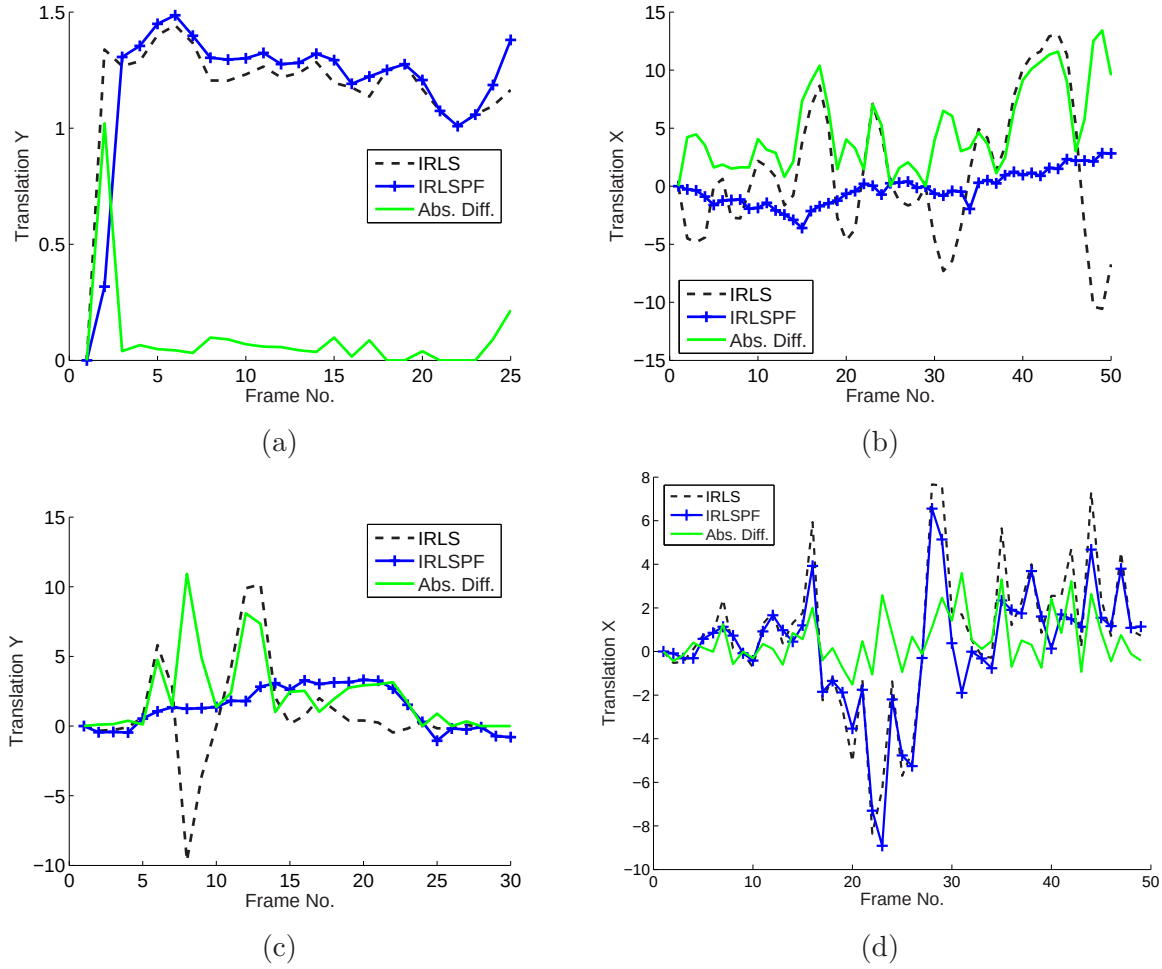


Figure 4.10: *IRLSPF results for real sequences, $N_s = 20$. (a) Garden (Pan). (b) Flowers (Random shake). (c) Film clip (Impulsive shake). (d) Donegal clip (Random Shake). Blue = IRLSPF. Black = IRLS. Green = absolute difference between the IRLS estimate and the IRSLPF estimate.*

extending the GMEPF to include partitioned sampling is an area for future work.

The chapter introduces just one of the mechanisms for building such hybrid solutions that combine history with standard LS estimation. An alternative strategy is to switch on a particle filter, or some method with history, only when the LS estimator is failing for some reason. This is an interesting idea, made complicated by the fact that DFD alone is unlikely to be a good indicator of LS failure. It is however an interesting consideration for further work.

The IRLSPF was also considered as a tool for diagnosing the type of shake, random or impulsive, present in a sequence. It is important for stabilising the video to the know the nature of any unsteadiness present. By comparing the IRLS estimate to the IRLSPF estimate it is possible to distinguish between the two types of shake.

Part II

GPU Accelerated Video Processing

5

General Purpose Computation on GPUs: An Introduction and Review[1]

HIGH performance 3D graphics systems are becoming increasingly common. Today's computer systems typically contain two major computational components; a central processing unit (CPU) and a graphics processing unit (GPU). The GPUs in modern graphics hardware are extremely powerful, with performance increasing rapidly year on year. These GPUs are also becoming much more programmable as well as providing increased precision such as support for floating point data. These developments, allied with their high processing power, mean that these chips are capable of performing more than the specific graphics computations for which they were designed. Modern GPUs may in fact be thought of as useful co-processors to the CPU.

Utilizing the incredible processing power of modern graphics hardware for general purpose computing is a fast growing area of research [186, 119, 87, 153]. It is often referred to as General Purpose Computations on GPUs (GPGPU) [66]. This chapter gives an introduction to modern graphics architectures and how they may be used for GPGPU and video processing in particular. To demonstrate some of the techniques which are used, an image processing algorithm known as Texture Synthesis is outlined as an example. Finally there is a brief review of some of the relevant literature on GPGPU. Chapter 6 then outlines some video processing algorithms which

[1] Results from this chapter have been published as: Fast Scale Invariant Texture Synthesis and GPU accelerated block searching. Claire Gallagher, Francis Kelly, and Anil Kokaram. *In Proceedings of IEE Conference on Visual Media Production (CVMP'05)*, November 2005 [60].

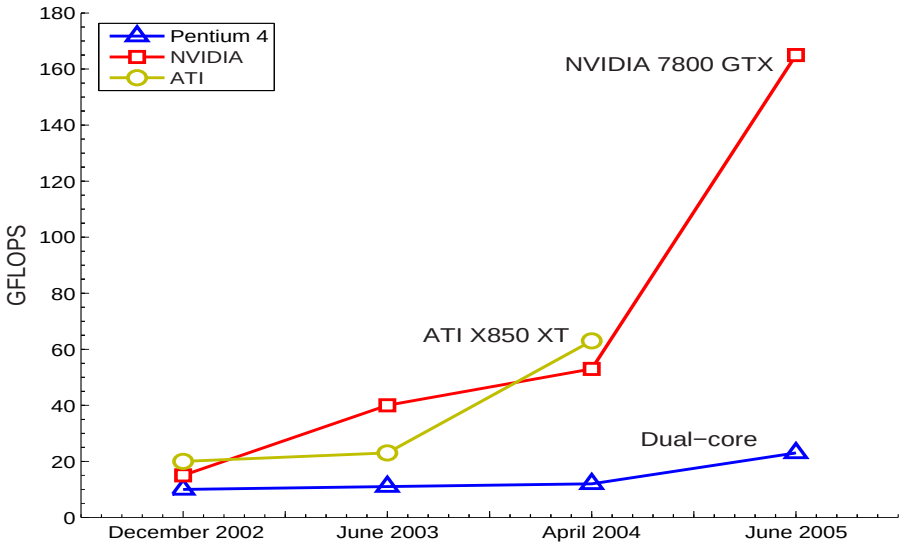


Figure 5.1: *Programmable floating-point performance of GPUs compared to CPUs over the last four years. These are observed values using synthetic benchmarks. The two graphics chips shown are the very latest top of the range chips available from ATI and NVIDIA. ATI are due to release their next generation chip soon (Autumn 2005). The final Intel Pentium 4 number is for a dual-core processor. The data was obtained from [153, 70]*

have been adapted to use the GPU as a co-processor for efficient implementation.

5.1 Why use GPUs?

Current graphics architectures provide incredible computational performance and memory bandwidth. The two biggest players in the graphics hardware market at the moment are NVIDIA [146] and ATI [5]. The NVIDIA GeForce 6800 Ultra has a peak memory bandwidth of 35.2 GB/sec and can achieve 53 GFLOPS [70], while the ATI X800 XT can sustain over 63 GFLOPS [153]. Compare this to the 14.8 GFLOPS theoretical peak for a 3.7 GHz Intel Pentium 4 SSE unit [22].

Not only are GPUs incredibly powerful, but the performance of these GPUs is also growing at an extraordinary rate. In fact over the last decade or so the processing power of GPUs has been growing at a rate faster than Moore's Law [135], which governs the performance growth rate of CPUs [119]. For example, the measured output of the GeForce 6800 is more than double that of the GeForce 5900, NVIDIA's previous flagship architecture [153]. According to Moore's Law, CPUs performance growth rate is approximately 1.5× per year, or about 60× per decade [49]. GPU performance on the other hand has been growing at approximately 2.0× per annum, or more than 1000× per decade. In other words, GPU performance is roughly doubling every six months, see Figure 5.1.

This phenomenal growth is driven by the demand of the video game market, a multi-billion dollar industry which is gaining quickly on the movie industry in terms of revenue and profit. In addition to consumer demand, the fundamental architectural differences between CPUs and GPUs make it easier for GPUs to achieve higher arithmetic intensity with the same transistor count. Large portions of a CPU transistor count are given over to non-computational tasks such as branch-prediction and caching. However, the highly parallel nature of graphics computations allows GPUs to dedicate additional transistors to computation. This allows for easier scaling of performance with each generation of graphics chip. Also this transistor count is quite significant. A relatively recent 3.2 GHz Intel Pentium 4 Extreme Edition processor contains approximately 180 million transistors. Compare this to the newest GPU available from NVIDIA, the GeForce 7800 GTX, which contains approximately 300 million transistors.

The latest generation of graphics hardware now contain programmable GPUs. Previously the number of operations that could be performed on the GPU was limited to fixed functionality such as Texture and Lighting. However the GPU has evolved to a situation where now, there are user programmable vertex and texture units, commonly referred to as vertex shaders and fragment shaders respectively. These programmable units allow for much more realistic visual effects in today's computer games especially, which is the main driving force behind the computer graphics hardware industry.

A further improvement in GPUs in the last 2–3 years is the increase in pixel accuracy from 32 bits per pixel to 128 bits per pixel with the introduction of the NVIDIA GeForceFX and the ATI 9700 series GPUs. This means that each pixels red, green, blue, and alpha component can now have 32-bit floating point accuracy throughout the graphics pipeline [2]. This increase in data accuracy combined with the increased programmability of the GPU makes it an attractive option for doing more general purpose computing than the graphics operations for which it was designed. In particular it may be thought of as a useful co-processor to the CPU for image processing tasks.

This computational power is readily available and relatively inexpensive with commodity off the shelf graphics cards incorporating the latest chips. Typically the latest generation top of the range graphics cards cost 400–600 euros when released, with the price dropping rapidly as new hardware emerges. In fact many PCs today ship with the newest graphics cards included; essentially providing a powerful co-processor for free. The following section outlines in more detail the architecture of modern GPUs and highlights the components which are used in GPGPU.

5.2 GPU Architecture

A GPU is basically a very efficient vector processor. It is optimised for processing the four-component vectors used to represent position [x,y,z,w] and colour [r,g,b,a] information in 3D graphics environments. Since graphics computations are highly parallelisable, GPUs typically

[2] This is true for NVIDIA GPUs. ATI however use a 24-bit floating point representation in their hardware.

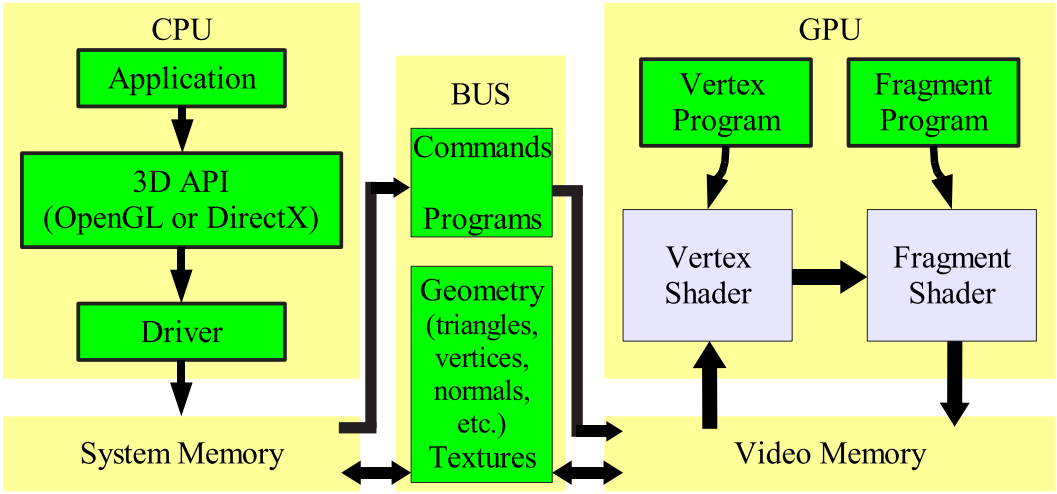


Figure 5.2: *3D Graphics Software Architecture. The application sits on top of a graphics API such as OpenGL, which is turn sits on top of the hardware driver. All commands, programs, textures, and geometry are passed between system and video memory via a bus. This bus is one of PCI, AGP, or the new PCI-Express.*

contain many pipelines working in parallel. NVIDIAs GeForce 6800 Ultra GPU for example has 16 pipelines. Figure 5.2 illustrates the 3D graphics software architecture of a modern GPU. The key components in this architecture are discussed next.

5.2.1 Interface

Exploiting graphics hardware acceleration for any application, be it 3D graphics or general purpose computing, requires an application programming interface (API). The interface consists of a set of several hundred procedures and functions that allow a programmer to specify the objects and operations involved in producing high-quality graphical images, specifically colour images of three-dimensional objects. There are two main APIs in use today: OpenGL [200, 149, 144] and Microsoft's DirectX [131]. These are software interfaces to the underlying graphics hardware.

OpenGL, which stands for Open Graphics Library, was introduced in 1992 as a cross-platform, independent 2D and 3D graphics API. OpenGL runs on every major operating system including Mac OS, OS/2, UNIX, Windows, and Linux. It is also callable from many programming environments such as C, C++, Java, and Fortran to name a few. DirectX on the other hand runs only on the Windows operating system.

The OpenGL specification is governed by an independent consortium formed in 1992, called the OpenGL Architecture Review Board (ARB) [149]. Composed of members from many of the industry's leading graphics vendors, the ARB defines conformance tests and approves OpenGL

enhancements. Through its use of extensions, OpenGL hardware developers can implement extended functionality and acceleration that is not part of the OpenGL core. For example, until recently the size of textures in the OpenGL standard were limited to dimensions which were a power of 2. NVIDIA however, incorporated support for rectangular textures (e.g 720×576) in their GPUs. This was exposed through an extension called `NV_texture_rectangle`, and allowed developers to use rectangular textures in their applications. Recently this functionality has been added to the OpenGL core specification. The OpenGL Extension Registry is maintained by SGI [150] and contains specifications for all known extensions, written as modifications to the appropriate specification documents. The registry also defines naming conventions, guidelines for creating new extensions and writing suitable extension specifications, and other related documentation.

OpenGL was chosen as the graphics interface here. The two main reasons for this were the platform independence of OpenGL and its extensions mechanism. Firstly, the platform independence of OpenGL is important if GPUs are to become viable co-processors in standard PCs. Secondly, through its use of extensions graphics card vendors have more flexibility in designing their GPUs. They can make available new functionality by simply releasing an extension to the OpenGL core, and are not dependent on any other chip manufacturers supporting the new functionality. With extensions the manufacturer simply releases a new driver for their card and the new functionality is ready to use. This is in contrast to say Direct X where it takes much longer for new technology to become available. Typically there is a new release of DirectX every 12–16 months which may or may not include the extended functionality provided by the OpenGL extensions. For a full list of current extensions see [150] and for further information on NVIDIA and ATI specific extensions see [146] and [5] respectively.

5.2.2 CPU–GPU Communication

As shown in Figure 5.2, all communication between the graphics hardware and the host system is done over a bus. The requirements of this bus have changed significantly over the years, ranging from the initial PCI bus standard of 1991 through to the latest specification, PCI Express, in 2004. A brief review of these standards follows.

PCI

The PCI (Peripheral Component Interconnect) bus [154] was first proposed by Intel in 1991 but did not obtain widespread popularity until the introduction of the Microsoft Windows 95 operating system. Intel at this time also created the Plug and Play standard and incorporated it into the design for PCI. Plug and Play is a feature that allows new devices to be inserted into a computer and be automatically recognized and configured to work in that system. Windows 95 was the first operating system to provide system level-support for Plug and Play. The ease with which new components could be added to a computer system using Plug and Play accelerated

the demand for PCI and it quickly became the bus of choice. In fact the PCI bus is still in widespread use today. The typical configuration is a 32 bits wide bus running at 33 or 66 MHz. This provides a peak bandwidth of 132 or 264 MB/sec respectively. However, this bandwidth is shared across all devices connected to the bus. When multiple devices are connected to the bus this bandwidth can be quickly taken up.

AGP

Initially the PCI bus was quite adequate, providing enough bandwidth for all the peripherals that most users needed, except one: graphics hardware. Particularly, 3D games were becoming much more detailed requiring more complex geometry and higher resolution textures. This fuelled the increase in the performance of GPUs from about 1995 onward. These new graphics cards enabled users to run such classical games as DOOM (1993) and Quake (1996) at higher frame rates with increased visual quality. The PCI bus quickly became unable to handle all the information passing between the main processor and the graphics processor. As a result, Intel developed the Accelerated Graphics Port (AGP) standard [2].

AGP is a bus dedicated completely to graphics cards; the bandwidth across the AGP bus is not shared with any other components. The AGP bus runs at a multiple of the system PCI bus speed. The original AGP standard provided twice the bandwidth of PCI. Further revisions of the AGP standard, AGPx2, AGPx4, and AGPx8, increased the bandwidth in multiples of 2, 4, and 8 times the AGP bandwidth respectively. The most recent AGP standard, AGPx8, provides up to 2.1 GB/sec of peak bandwidth to the graphics card. However it is worth noting that this bandwidth is only available when transferring data *to* the graphics card. Data travelling *from* the graphics card to the host system is over the slower PCI bus. The significance of this will become apparent in later sections.

PCI Express

With computers becoming ever more powerful and increasingly being used as an entertainment hub for multimedia storage and processing, the PCI bus is starting to show its age. With sound cards, TV tuner cards, interfaces to digital cameras, and host of other devices putting increasing strain on the limited PCI bus bandwidth, a new standard was needed. PCI Express (PCIe) [82, 154] has been developed by Intel and is a new approach to the peripheral bus problem. It abandons the shared-bus technology used in PCI and instead uses a point-to-point protocol. This means that a direct connection between two devices (nodes) on the bus is established while they are communicating with each other. While these two nodes are talking, no other device can access that path. By providing multiple direct links, such a bus can allow several devices to communicate with no chance of slowing each other down. Note that PCIe should not be confused with PCI-X and PCI-X 2.0. These were fast buses developed for high-end servers and were not suitable for the home computer market as it was very expensive to build motherboards

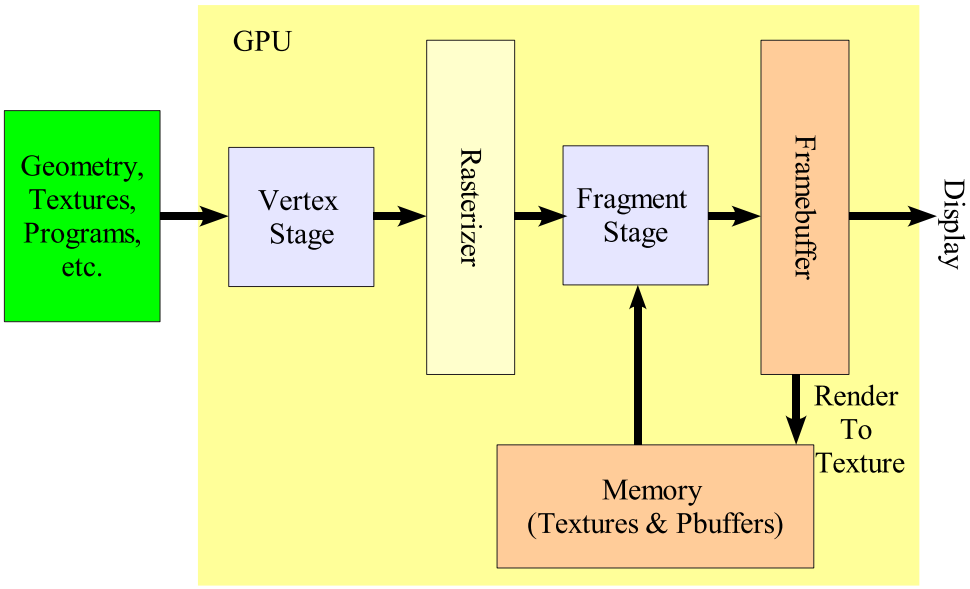


Figure 5.3: *The basic graphics hardware pipeline. The input geometry (vertices) are in local space relative to the object they describe. These get transformed at the Vertex Stage into screen space, and may have lighting applied. Triangles get rasterized from these vertices resulting in fragments. Each fragment is processed in the Fragment Stage where texture mapping and colouring are applied. Following this the fragment goes through a series of visibility tests before it finally results in a pixel in the framebuffer.*

with PCI-X.

The PCIe standard is also scalable. A basic PCIe slot, which replaces the traditional PCI slot, is known as a 1x connection. The 1x indicates that there is one lane to carry data. Should a component requires more bandwidth, PCIe x2, x4, x8, and x16 slots can be built into motherboards, adding more lanes and allowing the system to carry more data through the connection. The PCIe x16 slot is mainly used for graphics and replaces AGP as the standard way of communicating with graphics cards. PCIe x16 provides twice the bandwidth of AGPx8, or over 4 GB/sec of peak bandwidth. A key feature of PCIe, that is very relevant here, is that the bandwidth is available in both directions, both *to* and *from* the device. This is very important for GPGPU where it is necessary to get data back from the GPU to system memory as fast as possible.

5.2.3 The Graphics Pipeline

Figure 5.3 illustrates the basic graphics pipeline in GPUs. This overall model of the graphics pipeline has not changed much in the last ten years or so. Rather the capabilities of individual stages have advanced over the years. Graphics acceleration of the form which is used today first arrived around 1995. The origins of today's GPUs may be traced back to the introduction of

texture mapping and a z-buffer to graphics hardware. A time-line of the major developments in GPUs over the last ten years may look as follows:

- 1995-1998: Texture mapping and z-buffer.
- 1998: Multi-texturing.
- 1999-2000: Transform and Lighting.
- 2001: Programmable vertex shader. Pbuffers.
- 2002-2003: Programmable fragment shader. Floating point precision.
- 2004: Shader Model 3.0 and 64-bit colour support.

The more recent developments of programmable shaders, Pbuffers, and floating point precision are key to using the GPU for general purpose computing. The following sections briefly outlines the various parts of the pipeline necessary for GPGPU. A more detailed description of the overall pipeline may be found in the OpenGL Programming Guide [200].

Vertex Stage

Vertices are points in 3-D space which define graphics primitives like triangles, polygons, rectangles etc. These primitives are used to build up the geometry of the scene and define any 2-D or 3-D models to be displayed. The vertex stage processes the input vertices of the scene. It transforms these vertices from the object space in which they are defined to a screen space for viewing. It also assembles the vertices into triangles and can perform lighting calculations on each vertex. In older hardware the geometry sent to the graphics card was static. If this geometry or model data had to be altered in some way, it had to be transformed on the CPU and the new vertices downloaded to the GPU. Vertex shaders were designed to allow more control over vertex transformation on the GPU itself. This has obvious benefits as it frees up the CPU for other processes and also eliminates the need to download the same model over and over again.

Rasterizer

The rasterization stage determines the screen position covered by each triangle (defined by 3 vertices) and interpolates vertex parameters, such as colour and texture coordinates, across the triangle. The output from this stage is a fragment for each pixel location covered by a triangle. Fragments are the name given to the data in the pipeline before it gets output as pixels to the screen. Fragments are slightly different to pixels because some fragments can occupy more than one pixel on the screen. Also fragments can contain depth information about their visibility on screen. Fragments which will be occluded by other fragments can be discarded in a process known as the `DepthTest`.

Fragment Stage

The fragment stage computes the colour for each fragment, using the interpolated values from the vertex stage. Typically the fragment colour is composed of values sampled from textures. Textures are images which can be mapped onto the models built using any of the graphics primitives to add detail to a scene. For example a texture of skin can be mapped on to a model of a human to make it look more life-like. The texture image is looked up for the correct colour to add to the fragment. In order to use texture mapping, texture coordinates detailing how the texture is to be applied to the model are required. These coordinates can be defined explicitly in the application code or generated and manipulated in a vertex program. Similar to vertex shaders, early graphics hardware fragment units were limited in the number and type of operations they could perform. Programmable fragment shaders were introduced to allow improved control of the manner in which textures are applied to the fragments and how the final colour for a pixel is computed. Fragment shaders also tend to be more powerful than vertex shaders as they are usually operating on higher volumes of data, i.e. *millions* of fragments per frame compared to *thousands* of vertices.

Multi-texturing enabled the GPU to access more than one texture at the fragment stage and alloweds the blending of results from multiple textures to build up the final picture.

Render To Texture

In the final stage fragments are assembled into an image of pixels, usually by choosing the closest fragment to the camera at each pixel location. Pixels are rendered into a conceptual device called the framebuffer. Generally pixels rendered to the framebuffer are made visible on the screen to the user. However this framebuffer data can become corrupted if other windows on the screen are moved and overlap with the current window. This is because the same framebuffer, which is a piece of GPU memory, is shared for all applications on the desktop. To overcome this problem another useful feature of modern graphics hardware is exploited: off-screen rendering buffers called pixel buffers or *Pbuffers*. These Pbuffers reside in GPU memory and are similar to the framebuffer, except they are generated and controlled by the application. Also, the framebuffer does not benefit from the increase in pixel depth and is limited to 32 bits per pixel. Pbuffers are necessary when a 128 bits per pixel rendering buffer and floating point computation are required.

If further processing of the results generated is needed, the Pbuffer can be treated as a texture and passed through the graphics pipeline again. This was a key development in increasing the speed of visual effects on the GPU. Previously a function was provided called `CopyToTexture`, which copied the contents of the framebuffer into texture memory. However this was very inefficient as it involved unnecessary memory accesses to copy the data from the framebuffer to the texture. Rendering directly to the texture via a Pbuffer removed this copy. Using a Pbuffer texture as an intermediate storage area is known as a *multi-pass* algorithm; at least one more

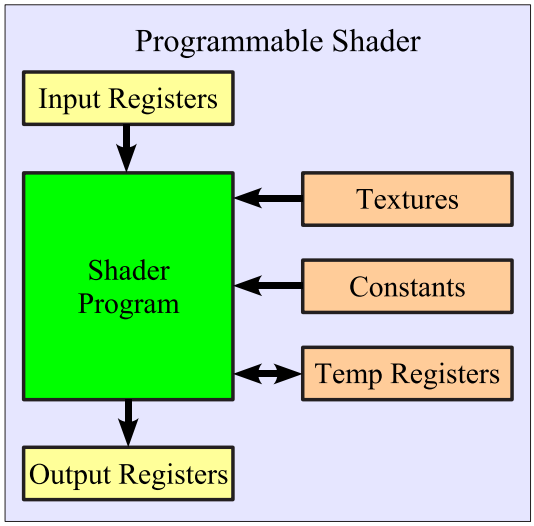


Figure 5.4: *Programmable Shader on a GPU. A shader program operates on a single input element (vertex or fragment) stored in the input registers and writes the results into the output registers.*

pass through the pipeline is needed to process the Pbuffer data. In a multi-pass algorithm, a new fragment program may be loaded on each pass to perform new calculations on the results of the previous pass. This is a method commonly used in GPGPU.

5.2.4 Programmable GPUs

GPUs have been evolving gradually from a fixed-function pipeline into a more flexible programmable pipeline. This extra programmability has been primarily concentrated on the vertex and fragment stages of the pipeline. In the fixed-function pipeline there were rather limited options for geometry processing in the vertex stage and pixel colour processing in the fragment stage[3]. In the programmable pipeline these fixed-function operations are replaced by user defined programs. These units are also sometimes known as vertex and pixel *shaders* and the programs they execute are called *shader programs*.

Shader Programs

The gradual increased programmability of GPUs fuelled the initial interest in GPGPU. However the vital step was the introduction of fully programmable hardware and an assembly language for specifying programs to run on each vertex [120] or fragment. As with most of advances in graphics hardware, increased programmability came about by the growth of the computer games industry and the need for more realistic scenes and effects. Following the success of offline rendering systems such as Pixar's RenderMan [189], GPUs evolved to include programmable

[3]For more information on this and the programmable pipeline see Appendix C.

shader units [153]. In contrast to the traditional limited fixed Texture and Lighting functions, RenderMan evaluated a user-defined shader program on each primitive, with impressive visual results. This encouraged the GPU manufacturers to follow suit.

Figure 5.4 shows the basic execution model of the programmable shader units on the GPU. For every vertex or fragment to be processed, the graphics hardware places a graphics primitive in the read-only input registers. The shader is then executed and the results written to the output registers. During execution, the shader has access to a number of temporary registers as well as constants set by the host application. There is however no communication between shader programs; there are no registers for instance that a shader can write to which another shader can access. The only viable inputs to a shader program is via textures.

High Level Languages

Typically programming for any development platform is most successful while using high-level programming languages, e.g. C, C++, Java etc. That is, languages which are a step above the native assembly language of the underlying processor. GPUs are no different and there are many different high-level languages to choose from including: Cg [127], Microsoft's HLSL [132], and the OpenGL Shading Language [98]. These languages are often referred to as shading languages as they compile into vertex and fragments shaders which are then executed on the GPU. They all abstract the capabilities of the underlying GPU and allow programming to be done in a more familiar C-like programming language. They provide abstractions which are close to the hardware, with instruction sets that expand as the underlying hardware capabilities expand.

Ashli [15] is a language which works one step again above the level of Cg, HLSL, or the OpenGL Shading Language. Ashli takes as input shaders from these languages and automatically compiles and partitions them to run a programmable GPU. Finally, the *Sh* shading language, which was developed by McCool et al. [129], is a metaprogramming shading language embedded in C++. Sh provides something called shader algebra for defining and executing procedurally parameterised shaders.

These shading languages, by their very nature, retain graphics specific constructs and terminology which can make GPGPU difficult for developers without a good knowledge of graphics hardware and graphics programming. Sh does incorporate the idea of streams and kernels but does not provide some of the basic operations common in general purpose computing, such as gathers and reductions.

The *Brook* stream programming model by Buck et al. [23] was initially developed as a language for streaming processors and later adapted to the capabilities of graphics hardware. Brook extends ANSI C with concepts from stream programming. Through the use of streams, kernels, and reduction operators, Brook abstracts the GPU as a streaming processor. It is intended to provide developers with a view of the GPU as a streaming co-processor without requiring an in-depth understanding of the latest graphics APIs or of the features and limitations

of modern graphics hardware. Brook automatically maps kernels and streams into fragment programs and texture memory.

All of these languages became available during the course of this work, but they are not considered here. Early releases of the compilers for these high level languages often contained many bugs and did not produce highly optimised assembly. As most of the necessary vertex and fragment programs implemented here were relatively simple, it was sufficient to write them directly in the OpenGL assembly languages. This allowed the programs to be fully optimised for the underlying hardware and avoid the use of any unnecessary registers and instructions. However as these high-level languages and their compilers mature, they are becoming the common way of writing vertex and fragment programs. Also if the GPU is to become widespread as a co-processor for general purpose computing then languages such as Brook, which hide GPU-specific details from the programmer, will become more and more important.

5.3 GPGPU: A Review

In the last few years there has been much research in the area of using graphics hardware for general purpose computing. In fact so much so that it is impractical to give a thorough review here. However, Owens et al. [153] provide a comprehensive state of the art report on GPGPU research as of Autumn 2005. Also a good source of information for implementation details and GPGPU publications may be found online at `www.gpgpu.org` [66].

5.3.1 Early Work

While programmable GPUs in todays graphics hardware are a recent development, the use of computer graphics hardware for general-purpose computation has been an area of active research for many years. For example, Pixar's Chap processor [117] in 1984 was one of the earliest processors to explore a programmable SIMD computational organization. Early work on procedural texturing and shading performed on the UNC Pixel-Planes 5 [163] in 1992 and PixelFlow machines [52] 1997, may be seen as the precursor to the high-level shading languages used with today's GPUs. The PixelFlow SIMD graphics computer was used to crack UNIX password encryption in 1999 [92].

The earliest work on desktop graphics processors used non-programmable GPUs. A major limitation of this generation of GPUs was the lack of floating-point precision in the fragment processor. Strzodka [175] showed how to combine multiple 8-bit texture channels to create virtual 16-bit precise operations. Often the work was done as a proof of concept while waiting for precise implementation when floating-point support became available. Larsen and McAllister [112] for example describe a technique for doing fast matrix multiplies on this pre-floating-point hardware. Other work includes techniques for the computation of Voronoi diagrams [78], proximity detection [79, 80], and interactive geometric computations [119].

5.3.2 Scientific Calculations and Visualisations

There has been a lot of work accelerating scientific calculations and visualisations, particularly simulating fluid flows. Much of this work is based on Partial Differential Equations (PDEs) which arise from solving the pressure-Poisson equation in the discrete form of the Navier-Stokes equations for incompressible fluid flows. When solving PDEs, the two common methods of sampling the domain of the problem are finite differences and finite element methods (FEM). Finite differences is more common in GPU applications due to the natural mapping of regular grids to the texture sampling hardware of GPUs [153]. The methods which have been described include conjugate gradients [16, 109], the multi-grid method [16, 65], and simple Jacobi and red-black Gauss-Seidel Iterations [71]. There has also been work on interactive volume segmentation and visualisation of MRI data [115] and biomedical images [75].

5.3.3 Audio Processing

The wide variety of applications that the GPU has been used for even extends to audio processing. Jedrzejewski [88] used ray tracing techniques on GPUs to compute echoes of sound sources in highly occluded environments. BionicFX is a company who have developed software [13] that accelerates audio effect calculations using the GPU.

5.3.4 Image Processing

Yang and Welch [202] show how to perform fast image segmentation and smoothing using graphics hardware. The segmentation was primarily based on simple thresholding techniques and the solution was refined using erosion and dilation functions, all implemented on the GPU. They state that this implementation is over 30% faster than a similar CPU implementation running on a 2.2 GHz Intel Pentium 4 processor. *Level-set* techniques are a more powerful tool for seqmentation but require significantly more compuation. Rumpf et al. [166] were the first to implement 2D level-set segmentation on GPUs. Lefohn et al. [116, 115] extended this work to 3D level-set segmentation.

There are also various implementations of the Fast Fourier Transform implemented on the GPU including [136, 23, 86]. Other transforms which have been accelerated using graphics hardware are the generalized Hough Transform [178] and the Discrete Wavelet Transform [193]. In addition there has been work on image registration using the GPU [176] as well as stereo and depth calculations [201, 63].

5.3.5 Video Processing

While there has been much work on image processing and other applications on the GPU, there is relatively little in the literature on video processing. This may be due in part to the data read-back bandwidth problem identified earlier, which is much more significant if trying to achieve

real-time or near real-time video processing. Despite this there have been some notable cases. In 2005 Apple released the Core Image framework which allows GPU acceleration of image and video processing tasks [3]. This has been integrated into the latest revision of the Mac operating system called Mac OS X Tiger. The OpenVIDIA project [151] is an open source project which implements computer vision algorithms on computer graphics hardware, using OpenGL and Cg.

Work similar to that presented in the next chapter was presented by Shen et al. [167] in 2003. They used the GPU to do fast motion compensation to accelerate decoding of video streams such as MPEG. This allowed real-time playback of high definition video on a PC with an Intel Pentium III 667 MHz CPU and an NVIDIA GeForce 3 GPU. However, it should be noted that this is only for the *decoding* stage. The work in chapter 6 is concerned with the more complicated task of motion estimation which is a key part of the *encoding* stage. Strzodka and Garbe [177] in 2004 presented a motion estimation and visualization system for use on graphics cards. Motion estimation is based on an optic-flow algorithm. Real-time performance is claimed for sequences at 320x240 image resolution running on a 2 GHz Intel Pentium CPU with a NVIDIA GeForceFX 5800 Ultra GPU.

Witt [199] demonstrated various video effects on a Sony Playstation2. Effects such as fades, wipes, page turns, etc. were all done in real-time using the video processing units of the Playstation2. A similar example for NVIDIA GPUs is available in the NVIDIA developer SDK [146]. Much of the work in this area, and in GPGPU in general, has occurred since about 1999/2000 onwards. Hence the work presented in this thesis was developed simultaneously with most of that in the literature. There is now however a broad body of work available to provide the interested GPGPU researcher with a solid foundation.

5.4 Using the GPU for General Purpose Computing

Since the GPU is primarily a graphics device, it can be difficult to see how it may be used for more general purpose computing. To this end a processing architecture known as *stream processing* is often used to explain the usefullness of GPUs for non-graphics orientated applications.

5.4.1 Stream Processing

A common analogy for GPU processing is that of stream computing or stream processing [123, 23, 190, 58]. A stream processor consists of two basic elements: *streams* and *kernels*. A stream is a collection of data requiring similar computation while kernels are functions applied to each element of a stream. A streaming processor executes a kernel over all elements of an input stream, addressing each element independently and placing the results into an output stream. This output stream can in turn be passed into a new processor executing a different kernel.

The GPU may be thought of as a stream processor designed for processing large amounts of data in parallel. This data is made up of large, usually contiguous, streams of vertex and

fragment data. In 1988 Fournier and Fussell [58] first presented a view of the GPU as a stream computing engine. In 2002 Purcell et al [160] described how the GPU can be considered as a streaming processor that executes kernels, written as fragment or vertex shaders, on streams of data stored in geometry and textures. Here streams are conceptually similar to arrays, except that all elements can be operated on in parallel. For image processing the framebuffer may be thought of as a large SIMD (Single Instruction–Multiple Data) processor where each pixel in the framebuffer is the result of a fragment operated on by its own fragment shader. While the fragment program executed is the same for all pixels, all pixels are processed independently. This independence is what enables many pixels to be processed in parallel and give GPUs their impressive processing power. For example, the latest NVIDIA GeForce 7800 GPU has six groups of four fragment shaders operating in parallel. This means that this GPU can process up to 24 fragments simultaneously.

Even with a good understanding of the basic architecture and software interface of a modern GPU, utilising all this processing power for general purpose computations is not immediately obvious. The following section outlines the basic model for programming the GPU, focusing in particular on using the GPU for image and video processing.

5.4.2 Programming the GPU

In order to use the GPU for general purpose computing the problem has to be mapped onto the GPU programming architecture. The basic model employed is that textures are used as memory, and fragment programs are used as kernels. Drawing geometry then amounts to invoking the kernel on the stream (the data stored in memory as textures). In the case of image and video processing, typically pixel-wise operations on the image data is required. To achieve this the images are downloaded to the graphics hardware as textures. The GPU can then be used to sample values from these textures and perform the necessary pixel-wise operations in a fragment program. This involves rendering a rectangle and texture mapping the images to this rectangle. If this rectangle is the same size as the images then the fragments will correspond on a one to one basis with the pixels in the texture. Fragment programs can then be used to do the required per-pixel calculations on the GPU. The results can then be read back to the CPU or output to a Pbuffer for further processing on the GPU.

Programming the GPU involves writing custom vertex and fragment programs to be executed by the vertex and fragment shaders. Vertex and fragment programs are written in a special type of assembly language for these programmable GPUs. The code in this thesis is based on the OpenGL `ARB_fragment_program` and `ARB_vertex_program` standards [150]. The following is the instruction set for fragment programs; the instruction set for vertex programs is very similar.

```
ABS ADD CMP COS DP3 DP4 DPH DST EX2 FLR FRC KIL LG2 LIT LRP MAD MAX
MIN MOV MUL POW RCP RSQ SCS SGE SIN SLT SUB SWZ TEX TXB TXP XPD
```

There are common operations such as add, subtract, multiply, absolute value, maximum, mini-

mum, etc., as well as many graphics specific instructions. More information on these instruction sets may be found in Appendix C. As an example the fragment code for calculating the difference between two images is outlined next.

5.4.3 Image Differencing

Subtracting two images is a simple but ubiquitous operation that illustrates image processing on the GPU. Firstly the two images are loaded into two texture units, `texture[0]` and `texture[1]`. A rectangle the same size as the image is then drawn and this invokes the fragment program below. The `TEX` instruction is used to sample the required pixels from each texture unit (lines 3 & 4). This uses the texture coordinates passed to it from the vertex stage to sample a value from a texture, with the result stored in a temporary register. The texture unit can perform bilinear interpolation if necessary when sampling the texture value. Finally the `SUB` instruction is used to subtract the two values and output the result (line 5). This output corresponds to either the framebuffer or to a Pbuffer.

```
!!ARBfp1.0
1 OUTPUT output = result.color;
2 TEMP frame1,frame2;

3 TEX frame1, fragment.texcoord[0], texture[0], RECT;
4 TEX frame2, fragment.texcoord[1], texture[1], RECT;
5 SUB output, frame1, frame2;
END
```

5.4.4 Scatter & Gather

Two fundamental features of programming on any device are the ability to read and write memory. If the read and write operations access memory indirectly, they are called *gather* and *scatter* respectively. In C code a gather operation looks like `a = b[i]`. A scatter operation is the opposite of a gather operation and may be represted by, `a[i] = b`.

Since the fragment processor can access memory in the form of textures it is capable of gather. However, fragments have an implicit destination address associated with them: their location in the framebuffer. A scatter operation would require that a program change the framebuffer write location of a given fragment. This is not currently possible and so scatter is not feasible using today's hardware. This has implications for the type of algorithm which can be accelerated using the GPU and the way in which these are implemented. For example recursive algorithms, where the results of one pixel are dependent on results of other pixels at the same time step or iteration are not possible.

Also operations which have a high spatial coherency are not efficient. Consider summing up the all pixels in say an 8×8 block. On the CPU this might be done using an operation such as

`sum += image[i]`, and looping over the 64 pixels in the block. It is possible to write a fragment program with 64 texture accesses and 64 additions. However this is a very in-efficient use of the GPU resources and is not a feasible option on current hardware. Instead alternative methods have to be found. The following section outlines two block summation methods on the GPU.

5.4.5 Block Summation

Generating error metrics over a block of pixels is a common operation required in many image and video processing tasks. Typically these operations involve some function of the sum over the pixels in the block, such as the Mean Square Error (MSE), Mean Absolute Difference (MAD), etc. Due to the lack of a scatter operation in a fragment program, this is not a trivial thing to do on the GPU. Two common block summation methods are presented next.

Automatic MipMapping: The image is first copied into a texture which has *automatic mipmapping* enabled. Mipmapping is a technique used in graphics programming when sampling from full-resolution textures may not be necessary, e.g. mapping the texture to an object whose size is smaller than the texture resolution. Instead it may be sufficient to sample from a lower resolution texture. Mipmaps are a multi-resolution representation of a texture image. Each level is sub-sampled by a factor of two from the previous level. GPUs which support automatic mipmapping can generate all the mipmap levels for the texture image in hardware very efficiently. Because the mipmapping filter used is a simple 2×2 box filter, every pixel in a given mipmap level is effectively an average of 2×2 pixels in the next highest resolution mipmap. Thus each pixel at say mipmap level 3 contains an average for blocks of size 8×8 pixels at level 0, the original image resolution.

Sum-Reduce: Similar to mipmapping, the image is copied into a texture. However in this case a multi-pass algorithm is used for summing a block of pixels. At each pass a 2×2 block of pixels is summed using a fragment shader and the output image sub-sampled by a factor of 2. This sub-sampled image is then passed back through the graphics pipeline as a Pbuffer texture. This process is repeated until a block of size $w \times w$ pixels has been summed. A block size of $w \times w$ requires $\log_2 w$ passes to generate the necessary summation. The resulting image is then returned to the CPU.

While the mipmapping method is the faster of the two, it is limited to 8-bit accuracy. This is because on current GPUs automatic mipmap generation is only supported for textures with 8 bits per channel. This can lead to overflow problems while doing the summation, and only gives an approximation to the mean value for a block of pixels. The sum-reduce method can take advantage of the floating point support in Pbuffers and hence give much more accurate results. However the multi-pass approach needed means this method is slower than the mipmap method.

5.5 Example: GPU Accelerated Texture Synthesis[4]

The problem of texture synthesis has received much interest in recent years. The idea behind a successful texture synthesis algorithm is to use a small texture sample image to create a much larger texture image by synthesising *new* texture which will be perceived to be visually similar to the original texture sample. This kind of operation is often required in the post-production of digital images when a large area is to be covered with texture that *looks like* some small example or 'seed' texture. Picture editing often requires filling in missing information or removing certain objects [8, 30], texture synthesis processes are thus often used to fill in such holes with reasonable material [102].

This description tries to strike a balance between algorithm understanding and implementation. Here the focus is on the GPU implementation of the Efros and Leung [47] texture synthesis algorithm. Hence a detailed description of this algorithm and texture synthesis in general is not provided here. However, the interested reader may find the following papers useful [47, 195, 102, 26, 28, 61].

The Efros search entails comparing the block around the current pixel in the texture to be synthesised against all possible blocks in the example texture. A 'distance' measure for each block is computed and the pixel corresponding to the block with the smallest distance is chosen as the new synthesised pixel. This pixel is placed in the output texture and the process repeats for all pixels to be synthesised, effectively 'growing' the new texture. Here the distance measure used is the mean absolute error (MAE) between two blocks. This type of block-by-block comparison is a process which is highly parallelisable and is hence well suited for implementation on the GPU.

In order to exploit the processing power of the GPU, the block search problem has to be rearranged. This is a key step and is necessary because the GPU is unable to perform a sliding window operation in order to search all possible blocks in the seed texture. Instead the data rearrangement allows such an operation by presenting data to the GPU which is pre-translated in a sense. Consider a typical CPU implementation. In that case seed blocks would be compared one at a time with the block for the current pixel to be synthesised (the current neighbourhood). On the GPU however this is not efficient. Therefore the only way to achieve acceleration via parallelising the operation is to compare the block to be synthesised to all possible seed blocks in one pass. This means that if the seed texture is $m \times n$ in size, then the rearranged data must be $(m \times w) \times (n \times w)$.

Figure 5.5 illustrates this technique. Two textures are created; one large seed image containing all possible blocks in the seed texture, and one containing the current block in the new synthetic texture. By tiling the current block texture $m \times n$ times and subtracting from the

[4]Results from this section have been published as: Fast Scale Invariant Texture Synthesis and GPU accelerated block searching. Claire Gallagher, Francis Kelly, and Anil Kokaram. *In Proceedings of IEE Conference on Visual Media Production (CVMP'05)*, November 2005 [60].

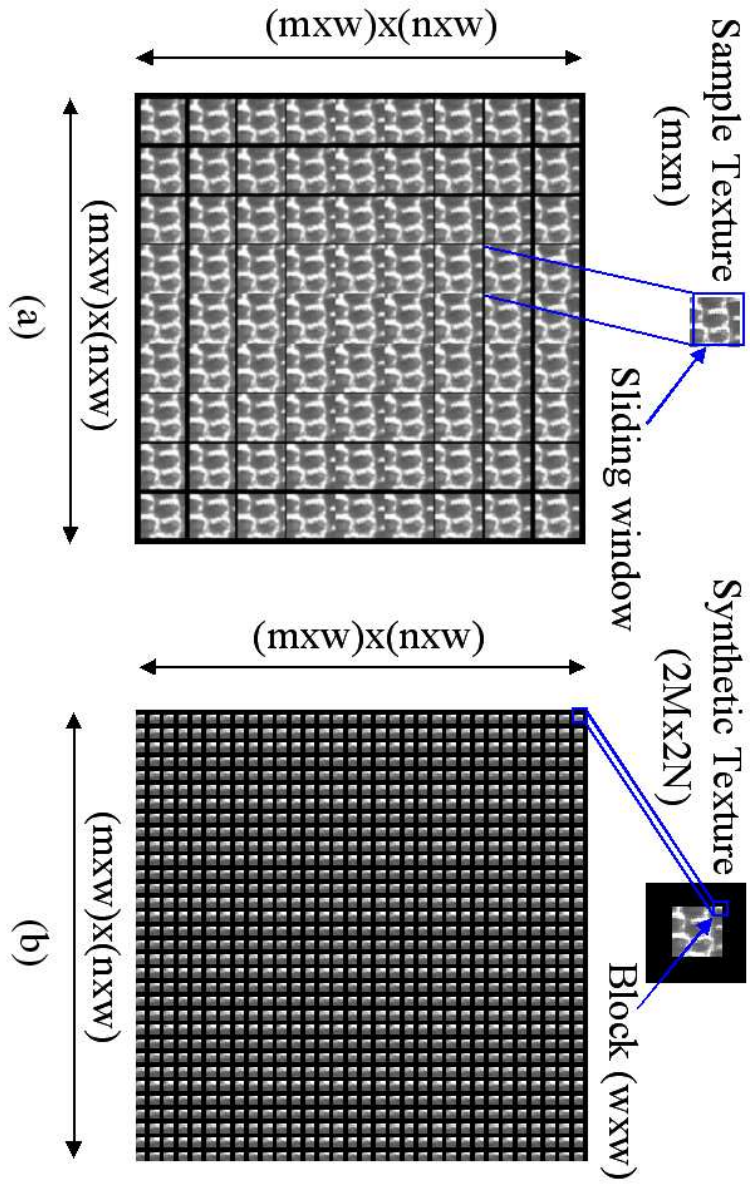


Figure 5.5: *Neighbourhood searching on the GPU. (a) Large example image containing all possible neighbourhoods in the example texture. Each cell contains shifted versions of the example image corresponding to each possible position of the example block. The shifts across are $-w : 1 : w$ (Matlab notation), and similarly for each row. The shifts are not easily noticeable due to the scale of this picture. (b) Current pixel to be synthesised neighbourhood block tiled $m \times n$ times. The image in (b) is subtracted from (a) yielding in one go the difference for all possible blocks. This is then summed on a block basis to give the distance measure for each pixel.*

large seed image texture, the difference between the current block and all possible seed blocks is generated. This difference image is output to a Pbuffer. A second pass of the algorithm then calculates the distance measure $d(\cdot,\cdot)$ for each block using the Pbuffer texture. This type of data re-arrangement is key to getting optimum performance from the GPU for general purpose computing.

Synthesising texture using the Efros method with block searching on the GPU proceeds as follows

1. A large seed image is created which contains all possible blocks in the original example texture, laid out in a gridlike fashion. This is stored as a texture on the GPU during initialisation and has size $(m \times w) \times (n \times w)$, Figure 5.5(a).

2. A $w \times w$ texture containing a block for the neighbourhood of the current pixel is uploaded to the GPU. This block texture is tiled $m \times n$ times, Figure 5.5(b).

3. Using pixel shaders the current block texture is compared to all possible blocks in the large example texture. The result is rendered to a Pbuffer.

4. A distance measure for the current pixel is generated by summing up blocks of size $w \times w$ in the resulting difference image.

5. An image containing a distance measure for each possible pixel in the seed texture is then returned to the CPU which chooses the output pixel.

6. Goto step 2 and repeat for all pixels to be synthesised.

The output from step 3 above is a large difference image containing the difference between the a block around the current pixel and all possible seed texture blocks. In order to generate a distance measure for each block a sum over each $w \times w$ block has to be performed. Two methods for doing this were outlined earlier: *automatic mipmapping* and *sum-reduce*. The limited 8-bit accuracy of the automatic mipmapping method can lead to an incorrect distance measure for a pixel. As can be seen in Figure 5.6 (a), this can cause the wrong example pixel to be chosen. The sum-reduce method however can take advantage of the full 32-bit floating point capabilities of the GPU. This method gives almost identical results to a C++ implementation running on the CPU, Figure 5.6 (b), (c).

5.5.1 GPU Texture Synthesis Performance

Because there is a limited amount of memory available on the GPU, typically 128 or 256MB, there are hardware limitations on the maximum size of textures and Pbuffers that can be created. In this case the amount of memory needed depends on the seed texture size, $m \times n$, and the block size, $w \times w$. The large seed image size is $(m \times w) \times (n \times w)$ and hence so is the difference image generated in step 3 above. These must be stored as two textures on the GPU. Due to

m × n	w × w	CPU	8-bit GPU	Speedup	32-bit GPU	Speedup
64x64	8x8	64s	18s	3.5	21s	3.0
64x64	16x16	250s	32s	7.8	50s	5
128x128	8x8	1140s	240s	4.8	287s	4.0
128x128	16x16	4442s	455s	9.8	N/As	N/A

Table 5.1: *Comparing speed of CPU and GPU Efros texture synthesis with Intel P4 1.6GHz NVIDIA GeForce 6800 AGP. 8-bit GPU and 32-bit GPU refer to the automatic mipmapping and sum-reduction methods for block summation respectively. N/A = not enough memory for difference texture. The output textures were each twice the size of the input textures.*

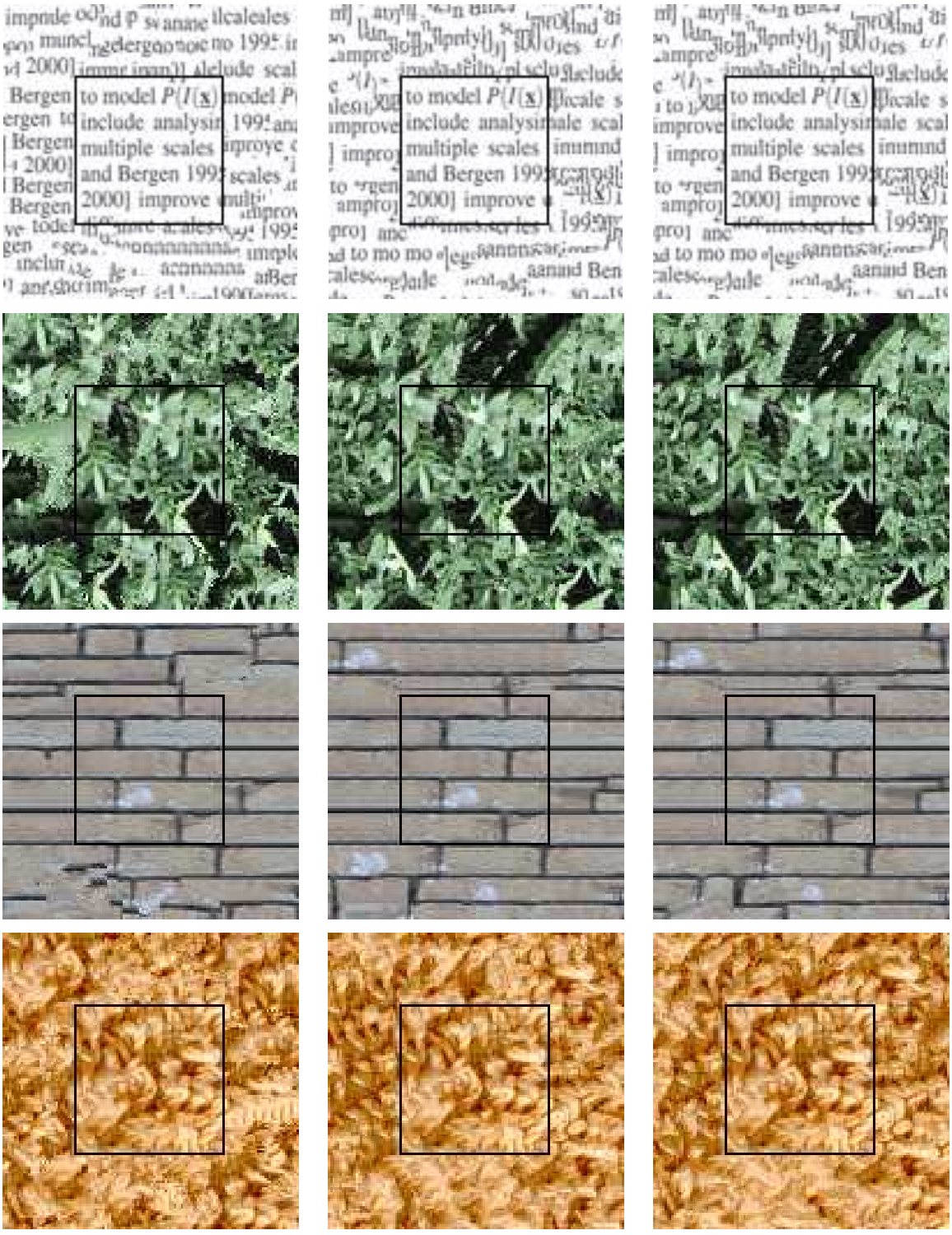
the texture size limitations on the test graphics card the maximum example texture and block sizes for mipmapping method were 128×128 pels and $w = 16$ respectively. Since the sum-reduce method uses floating point for each channel in the difference image, and hence needs more memory, the maximum example and block sizes are 128×128 pels and $w = 8$ respectively. Future work involves dividing the seed texture in sections and doing each section individually to deal with larger seed textures and block sizes.

GeForce 6800 AGP: Table 5.1 compares the speed of Efros texture synthesis on both the GPU and CPU. Given a window size of $w = 8$ the GPU is on average 3.0 to 3.5 times faster than a similar CPU implementation. For $w = 16$ the GPU is on average 5.0 to 8.0 times faster. Because of the implementation used here for the distance measure evaluation, the window size in both methods is limited to $w \times w$ pixels where $w = 2^n, n > 0$. The results presented here were generated on a 1.6GHz Intel Pentium 4 with a NVIDIA GeForce 6800 AGP graphics card with 128MB on-board memory.

GeForce 6800 PCIe: The results for a machine with Intel Dual-Xeon 2.8 GHz processors and a NVIDIA GeForce 6800 PCIe GPU with 256MB on-board memory are illustrated in Table 5.2. The increased memory on this graphics card enabled the sum-reduce method to be used for the 128×128 seed texture with $w = 16$. In this case the GPU implementation is 5.0 to 7.0 times faster than the CPU only implementation for floating point summation. While using the lower precision mipmapping method, the GPU is up to 17.0 times faster than the CPU. While the lower precision does mean that results are not identical to the CPU, the synthesised textures are still quite acceptable, see Figure 5.6.

5.6 Final Comments

As noted at the beginning of this chapter, modern GPUs are now very powerful processors in their own right. A vibrant area of research known has GPGPU has developed, focusing on using these processors for more general purpose computations than that for which they were designed. In recent years in particular there have been some key developments which have made this type



(a) (b) (c)

Figure 5.6: *Results from the Efros texture synthesis. (a) Using 8-bit textures for distance measure. (b) Using 32-bit floating-point textures for distance measure. (c) CPU result. The original seed texture is shown inside the black square.*

m × n	w × w	CPU	8-bit GPU	Speedup	32-bit GPU	Speedup
64x64	8x8	42s	6s	7	8s	5.25
64x64	16x16	142s	10s	14.2	20s	7.1
128x128	8x8	663s	73s	9.1	101s	6.6
128x128	16x16	2261s	130s	17.4	308s	7.3

Table 5.2: *Comparing speed of CPU and GPU Efros texture synthesis with Intel Dual-Xeon 2.8 GHz NVIDIA GeForce 6800 GT PCIe. 8-bit GPU and 32-bit GPU refer to the automatic mipmapping and sum-reduction methods for block summation respectively. The output textures were each twice the size of the input textures.*

of processing possible. These include the introduction of programmable stages in the pipeline, the ability to render directly to a texture, and the support of 32-bit floating point processing. Perhaps the key development though has been the new PCI Express bus standard, giving much higher bandwidth between CPU and GPU.

From the results reported in the GPGPU literature and the texture synthesis example shown here, it is clear that the GPU has much potential as a co-processor to the CPU. However it was noted that using the GPU in this way involves a significant learning curve. A good understanding of the architecture of the GPU and the type of operations which can be implemented efficiently on the GPU is necessary. The problem has to be expressed using a graphical programming interface which often is not straightforward. The introduction of high level languages that abstract the graphics specific nature of the underlying hardware away from the programmer should help alleviate this problem and encourage more widespread use of the GPU for non-graphics processing.

The impressive processing power of modern GPUs allied to these developments have helped to make the GPU a viable option for fast image and video processing. The next chapter outlines using the GPU as an image co-processor for real-time video processing.

6

Real-Time Video Processing Algorithms on GPUs [1]

Real time image-centric video processing has traditionally been the domain of hardware manufacturers. Digital video coding for example has long used dedicated motion estimation chips for real-time performance. In the television industry, hardware units that perform such tasks as image stabilisation and brightness flicker compensation have recently become available from companies such as Snell & Wilcox [169] and For-A [56]. In that industry real time processing is standard because of the demand on throughput. In the film post-production industry the recent trend toward totally digital processing is increasing the demand for similar units that perform these tasks in the digital film domain at even higher resolutions. Image shake for instance, affects not only archive footage, but also modern stock.

Real-time processing of this type has not been possible on standard desktop PCs. This chapter outlines how to use GPUs to accelerate some video processing algorithms, providing in cases real-time speed on industrial standard video formats. The following video processing algorithms are implemented using the GPU as an image co-processor to the CPU.

- Local Motion Estimation and Compensation. (Block matching, interpolation, generating gradients, multi-resolution.)
- Global Motion Estimation and Compensation. (Affine transformations. Shake removal.)
- Flicker Stabilisation.

Motion estimation is one of the basic tools in video processing. However real-time perfor-

[1] Results from this chapter have been published in [93, 94, 95, 158, 32, 96].

mance on a desktop PC is currently not possible; typically dedicated hardware is necessary for real-time processing. Two region based motion estimation algorithms are implemented here using the GPU as a co-processor for common image processing tasks such as generating DFDs and image gradients. These are a simple Full search block matching algorithm (FBM) and a Wiener based motion estimation algorithm (WBME).

Flicker [158] is a temporal intensity fluctuation problem. It is typically caused by different exposure times on the frames of a sequence, but can also be induced in the telecine transfer of film to digital format. Flicker removal has become particularly important in modern footage because 'inbetweening' with non-calibrated cameras is a typical task especially for outdoor nature shots. Inbetweening [29, 187], also known as view interpolation, is a special effect which gives the impression that time is frozen while a camera is allowed to move around the object, capturing views from all angles. New frames are interpolated 'inbetween' some number of fixed frames which are captured by an array of cameras positioned around the object. Because there are multiple cameras involved, there can be intensity fluctuations between the frames which needs to be resolved before the new frames can be produced.

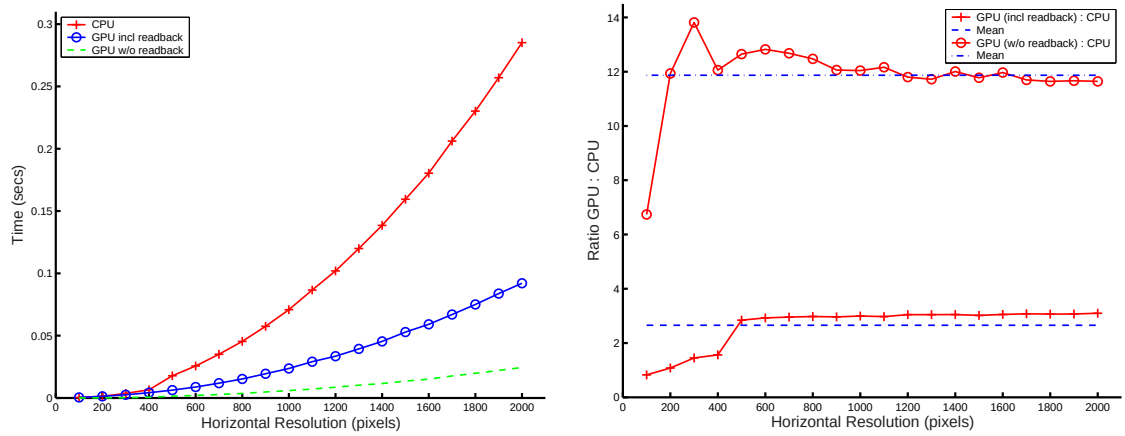
To begin this discussion the following section describes using the GPU for fast bilinear interpolation.

6.1 Image Interpolation on the GPU [2]

Image interpolation is a fundamental requirement for many image and video processing applications. It can however often be a bottleneck in terms of performance for these applications. Motion estimation is one such application where image interpolation can cause a large performance hit. Video codecs such as H.263 and MPEG-2 recommend using at least 1/2 pel accuracy for the motion vectors. MPEG-4 and H.26L use higher accuracy motion vectors of 1/4 pel and 1/8 pel. For restoration and movie post-production purposes motion vectors with a minimum of 1/8 pel accuracy are typically required. However in order to estimate and compensate fractional pel displacements, the image signal has to be interpolated. The type of image interpolation used is normally governed by an accuracy/speed trade-off. As interpolation accuracy increases so does computational cost. In motion estimation algorithms bilinear interpolation is usually a sufficient compromise between interpolation accuracy and speed.

As already noted in chapter 5, the GPU has dedicated interpolation units which are used while sampling texture data. Hence to use the GPU for image interpolation involves downloading the image as a texture to the graphics card. The texture is then mapped onto a rectangle, drawn parallel to the screen and which is the same size as the image. If this rectangle is moved, resized, or warped in any way, the texture unit will interpolate the texture at the correct position

[2] Results from this section have been published in: Fast Image Interpolation for Motion Estimation using Graphics Hardware. Francis Kelly and Anil Kokaram. *In Proceedings of SPIE Conference on Real Time Imaging VIII*, January 2004 [94].



(a) (b)

Figure 6.1: *GeForce 5600 AGP: Using the GPU for bilinear interpolation. Only the horizontal resolution is shown, but all images had an aspect ratio of 4:3. Results show times for both reading images back from GPU and for not. (a) Time in seconds for GPU vs CPU. (b) Speed-up of GPU over CPU including mean values.*

before applying it. The resulting on-screen image will be correctly interpolated for the given transformation, and may be read back to system memory for further processing. The texture unit can be set to perform either nearest neighbour interpolation or bilinear interpolation. In this case it is set to do bilinear interpolation.

The following code is a simple fragment program to sample an input texture. Lines 1 and 2 assign names to the output register and a temporary register respectively. The `TEX` instruction is used to interpolate the image in `texture[0]` at the position given by the texture coordinates `fragment.texcoord[0]`. These coordinates are passed from the vertex stage of the pipeline. The vertex stage can be used to manipulate these coordinates to apply a transformation to the image such as translation, rotation, etc. The result is placed in the temporary register `image` which is then moved to the output register.

```
!!ARBfp1.0
1 OUTPUT output = result.color;
2 TEMP image;
3 TEX image, fragment.texcoord[0], texture[0], RECT;
4 MOV output, image;
END
```

6.1.1 GPU Image Interpolation Results

Two sets of timings for image interpolation are given here. The first set were generated on an Intel Pentium 4 1.6 GHz machine with a NVIDIA GeForce 5600 GPU running on a AGPx4 bus. The second set were generated on a Intel Dual-Xeon 2.8 GHz machine with a NVIDIA

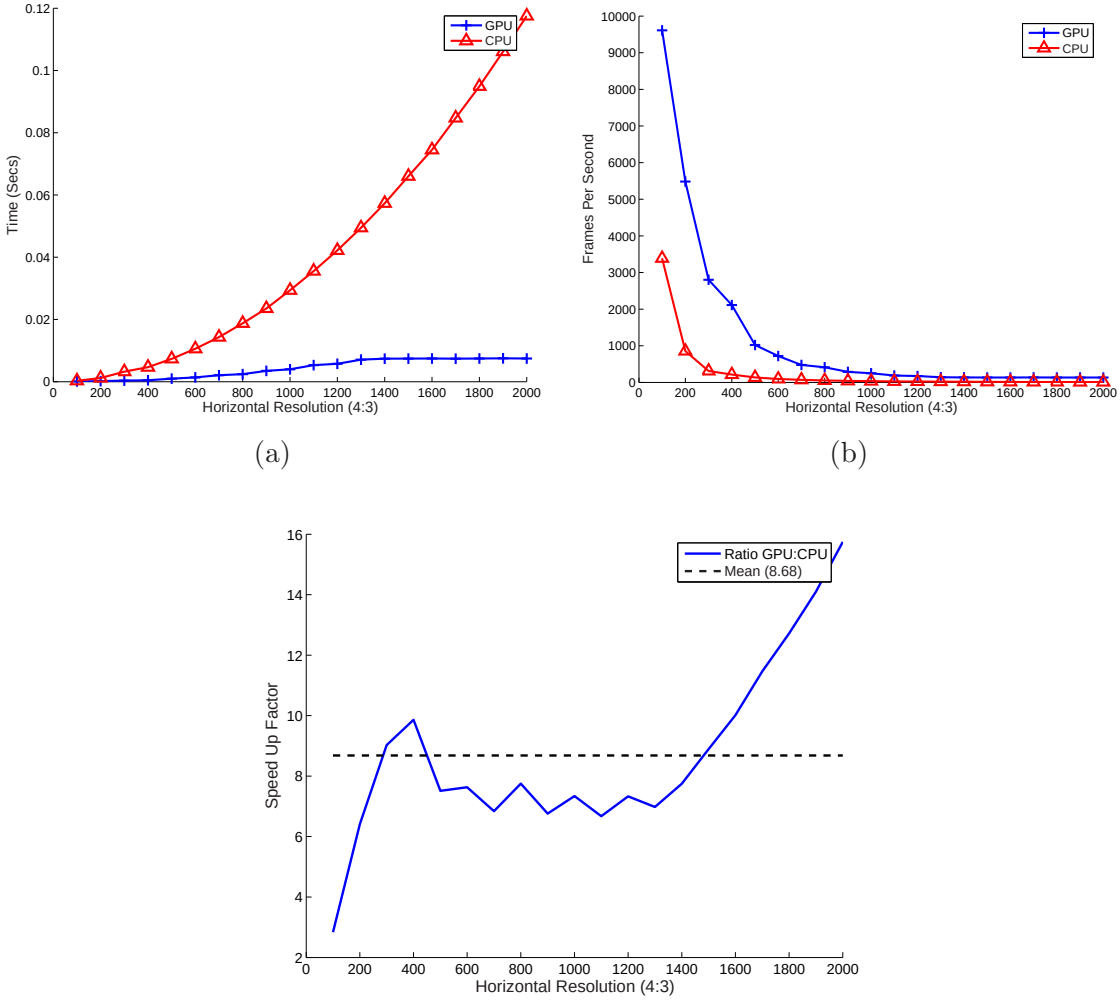


Figure 6.2: *GeForce 6800 PCIe: Using the GPU for bilinear interpolation. Only the horizontal resolution is shown, but all images had an aspect ratio of 4:3. Results include the time taken to read the images back from the GPU. (a) Time in seconds for GPU vs CPU. (b) Frames per second for GPU vs CPU. (c) Speed-up of GPU over CPU including mean value of 8.68.*

GeForce 6800 GT GPU, which used a PCI Express bus. At the time of writing (Autumn 2005), the first system is 2-3 years old, while the second system represents a modern high performance CPU/GPU combination.

GeForce 5600 AGP: Figure 6.1 (a) compares the time taken to bilinear interpolate an image using both the GPU and CPU. Two GPU times are shown: one including reading the image data back from the GPU and the other without data readback. Only the horizontal resolution of the images used is shown for clarity, but all images had an aspect ratio of 4:3. Figure 6.1 (b) gives an indication of the speed up achieved over the CPU when using the GPU for interpolation. Again the two cases for data readback are shown. As can be clearly seen the GPU version is on average almost three times faster than a CPU only version, even with the time taken to read the interpolated image back to the CPU included. The CPU version uses the Intel Integrated Performance Primitive (IPP) libraries which are highly optimised for image processing on Intel processors [81].

Compare this to the time taken to do interpolation on the GPU without reading the data back. In this case the GPU is approximately twelve times faster than the CPU implementation. This shows that almost 75% of the time taken to use the GPU for interpolation is spent on reading the data back from the graphics hardware. This is due to the PCI bottleneck. Based on these results, if data readback bandwidth was the same as the AGP bandwidth, then the GPU would be almost ten times faster for bilinear interpolation than the CPU.

GeForce 6800 PCIe: Again the results are for bilinear interpolation and are compared to the very efficient IPP libraries. Figure 6.2 (a) and (b) compare the time in seconds and the frames per second respectively for interpolation on the GPU and the CPU for various image resolutions. Figure 6.2 (c) shows the relative speed up of the GPU interpolation over the CPU. Note that these results include the time taken to read the data back from the GPU to the CPU. However, here the bus is the faster PCIe bus and the results are close to those predicted from the previous system. The GPU is on average over 8.5 times faster than the CPU for bilinear interpolation. There is however a noticeable band of resolutions, as seen in Figure 6.2 (c), where the GPU speed up over the CPU is almost constant. The reasons for this are not known. However it may be due that the IPP libraries are particularly well optimised for these resolutions or that these resolutions correspond to good caching performance, yielding few cache misses.

These results are very impressive, particularly those for the PCIe case. Since image interpolation is a fundamental operation in motion estimation, the next section outlines how Global Motion Estimation (GME) and Compensation (GMC) may be accelerated using the GPU.

Resolution	CPU GMC	GPU GMC	Speedup Factor
180x144	960	5100	5.3
360x288	245	2050	8.3
720x576	60	480	8.0
1080x720	32	228	7.1
1920x1080	12	130	10.8
2048x1536	8	129	16.1

Table 6.1: *GPU vs CPU for Global Motion Compensation with six parameter affine model. Results are in frames per second and include the time to read the images back from the GPU.*

6.2 Global Motion Estimation & Compensation [3]

As shown in chapter 4, estimation of the update term in least squares estimation or calculation of the likelihood function in the particle filter requires global motion compensation (GMC) of a frame with the current estimate of the motion parameters. In order to accurately represent the global motion of a frame, a sub-pixel accurate motion vector is typically required. Interpolation of the image signal is required to motion compensate a frame with a fractional motion vector. Typically bilinear interpolation is sufficient. Following on from the last section, an efficient way of performing GMC is to harness the power available in today's graphics hardware.

GPUs are designed for efficient processing of complex 3D models and scenes, mainly for use with the latest computer games. Usually 2D texture maps are used to apply detail to these 3D models which can undergo any number of transformations. They are therefore quite adept at dealing with 2D image manipulation. By treating the frames of the video sequence as textures GMC may be performed very quickly on the GPU. As before the texture is mapped to a 2D rectangle the same size as the image and drawn parallel to the screen. By stretching, rotating, resizing the rectangle onto which the texture is mapped, the GPU interpolates the texture to fit accordingly. By warping the rectangle with a six parameter affine model the necessary frame compensation can be achieved. All of this is done on the GPU.

Table 6.1 compares the speed of performing GMC with a six parameter affine model on the CPU and the GPU for a range of image resolutions. On average the GPU is four times faster for GMC than the CPU. This includes the time taken to read the compensated image back from the GPU to the CPU. The CPU transformations were performed using the latest Intel IPP performance libraries which are highly optimised for image processing on Pentium and Xeon processors. The test system here was the GeForce 6800 PCIe one outlined earlier. All tests were performed five times and the results averaged.

[3]Results from this section have been published in: Online Global Motion Estimation. Francis Kelly and Anil Kokaram. *In Proceedings of IEE Conference on Visual Media Production, CVMP'05*, November 2005 [96].

6.2.1 Shake Compensation [4]

Video stabilisation, or shake removal, also requires full frame compensation. In [32] the motion of the largest object in the image is estimated using integral projections [114] for global motion estimation. The high frequency motion components (the shake) are removed with an FIR filter and the input images are compensated according to this removed component. To create the final images for output, each image must be shifted to compensate for the unwanted motion component. In this algorithm the compensation stage is the bottleneck, since the integral projections method of estimation is very efficient. Using the GPU for compensation enables real-time hand held shake removal on relatively low performance processors. For example a hand held camera with a PAL format video feed was connected to a 1.0 GHz laptop and the stabilised video sequence displayed on the screen in real-time i.e. 720×576 at 25 frames per second. This is comparable performance to commercial video stabilisation hardware such as the *For-A IVS-300A* video stabiliser [56].

6.3 Local Motion Estimation & Compensation

As shown in previous chapters, motion estimation is a key component in video processing. Motion estimation is however very computational intensive, especially at high resolutions. Typically dedicated hardware such as on a MPEG encoding chip is needed for accurate real-time motion estimation. On the PC real-time motion estimation is some way off. Because image sequences typically exhibit non-integer motion, sub-pel accurate motion vectors are required to accurately motion compensate the scene. This means interpolation of the image signal is required, which adds further computational demands. The strong results of the previous section suggest that the GPU can be used for fast image interpolation and may help to accelerate motion estimation. In the following section the GPU implementation of a Full Search Block Matching algorithm (FBM) is outlined.

6.3.1 Full Search Block Matching [5]

Block matching is a very common approach to motion estimation. It is quite simple to implement as it requires highly regular operations which are easily parallelizable. As such it has been implemented in video coding hardware [170, 156]. The image is divided up into blocks of size $M \times N$ pixels, and a motion vector $\mathbf{d}$ for each block in the current frame is estimated by finding

[4] Results from this section have been published in: Gradient Based Dominant Motion Estimation with Integral Projections for Real Time Video Stabilisation. Andrew Crawford, Hugh Denman, Francis Kelly, Francois Pitie, and Anil Kokaram. *In Proceedings of IEEE International Conference on Image Processing, ICIP'04*, October 2004 [32].

[5] Results from this section have been published in: Fast Image Interpolation for Motion Estimation using Graphics Hardware. Francis Kelly and Anil Kokaram. *In Proceedings of SPIE Conference on Real Time Imaging VIII*, January 2004 [94].

the best matching block in a defined search space in the previous frame.

Block matching algorithms usually search a known set of candidate motion vectors, choosing the motion vector for each block which minimises some function of the Displaced Frame Difference (DFD). The search space is usually given as $\pm w$ pixels for the horizontal and vertical components. For integer accurate motion estimation there are $(2w+1)^2$ vectors in the candidate set. For fractional motion vectors there are even more, depending on the accuracy required. The matching criteria used here is the sum of absolute differences (SAD).

In exhaustive search or full search block matching (FBM) all possible vectors in the search space are checked. This is very computationally intensive. As stated earlier there are $(2w+1)^2$ possible motion vectors for integer accurate block matching with a search space of $\pm w$ pels. For sub-pel accuracy there are $(2w/\alpha+1)^2$ motion vectors in the candidate set, where $0.0 < \alpha \leq 1.0$ is the accuracy required. This is approximately a factor of $(1/\alpha)^2$ increase over the number of integer vectors to search. Also $(2w/\alpha+1)^2 - (2w+1)^2 \approx (1/\alpha)^2 - 1$ of these vectors have fractional components, which means that the image must be interpolated to check these vectors. Bilinear interpolation can take up to four multiplies and three additions extra per pixel. This means approximately $7 \times ((1/\alpha)^2 - 1)$ additional operations per pixel are required for full search sub-pel accurate motion estimation. For example, an accuracy of $\alpha = 0.5$ means approximately 21 extra operations per pixel, and $\alpha = 0.25$ gives approximately 105 extra operations per pixel.

6.3.2 FBM using the GPU

The approach taken for FBM on the GPU is slightly different than a CPU implementation. The GPU is designed to work on large volumes of data in parallel. To exploit the GPU hardware efficiently, it is necessary to restructure the standard block matching algorithm. Typically block matching implementations shift blocks around the search space individually, calculating the SAD for each motion vector. On the GPU however it is more efficient to shift the entire frame at once for each motion vector, with the SAD then calculated on a block basis.

An appropriate algorithm flow for FBM using the GPU as a co-processor for the CPU is shown in figure 6.3. Firstly the two frames are downloaded as textures to the graphics hardware. These are noted as `Texture0` and `Texture1` respectively. The current motion vector to be checked is passed as a parameter to the GPU. The DFD for the two frames is then generated using vertex and fragment programs. In this way if a frame needs to be interpolated it will be interpolated on the GPU. This results in an image which is the absolute value of the DFD for the two frames. This image is then read back to the CPU where a SAD measure for each block in the image is generated. This is repeated for each motion vector in our candidate set. The motion vector which yielded the smallest SAD for each block is chosen as the motion vector for that block.

In order to alleviate the PCI bottleneck in data read mode on the AGP system, it is necessary to perform more of the algorithm on the GPU. It is possible to do this by calculating the blocks

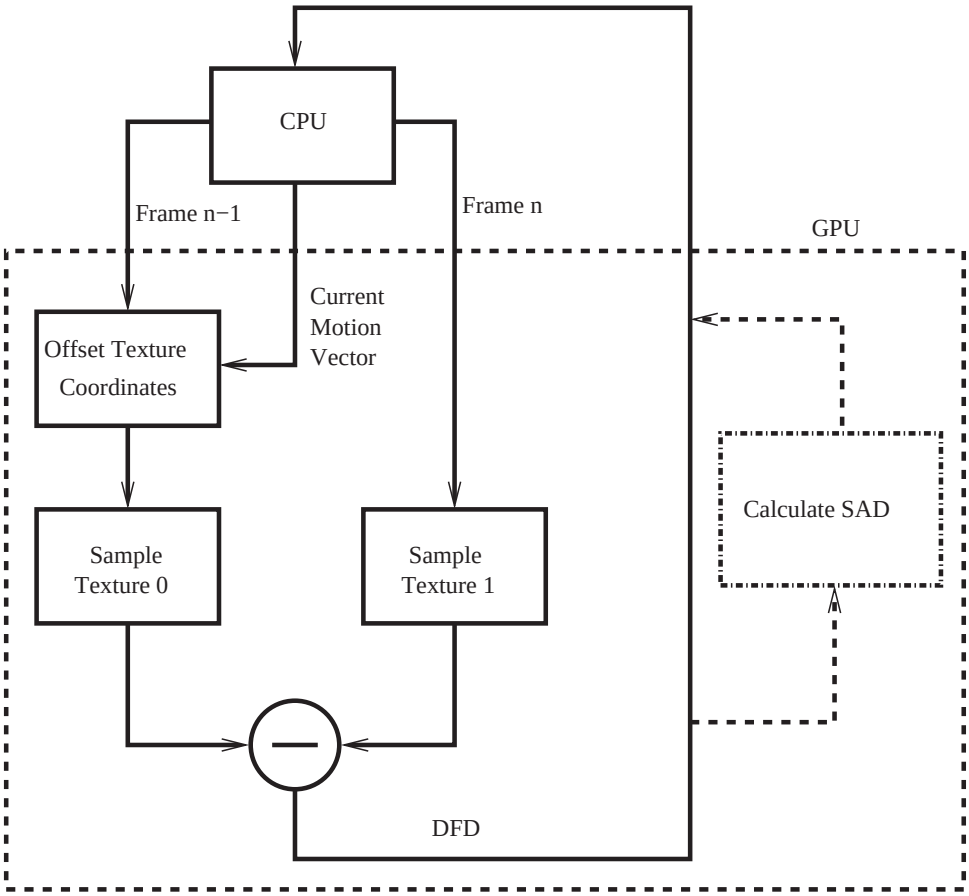


Figure 6.3: *Block diagram showing the combined GPU and CPU operation for the FBM algorithm.*

SAD using fragment programs. This is shown as the dashed option on the data read path in Figure 6.3. If the SAD is generated on the GPU only one value per block has to be read back, rather than $N \times N$ pixels for each block. The SAD is calculated using the sum-reduction technique outlined in the previous chapter. This is a recursive algorithm where many passes through the pipeline are necessary to generate the SAD for the $N \times N$ block. On each pass 2×2 blocks of pixels are summed and the image sub-sampled by a factor of two. Every pass through the pipeline reduces the frame rate of the GPU. Generating the SAD on the GPU in this manner requires $\log_2 N$ additional passes. However, using the GeForce 5600 AGP system, it was found to be on average 43% faster to do the SAD calculation on the GPU than to read back the whole DFD to the CPU and generate the SAD there.

GPU FBM Vertex & Fragment Programs

This first segment of code shows the vertex program used to set up the texture coordinates for the two textures. For clarity only the necessary lines of code are shown. It is these texture

coordinates which will be used by the texture units to look up values in the textures. For `Texture1` we just pass through the current input position as the texture coordinates (line 12). This is because `Texture1` corresponds to the current frame of the video sequence, it does not need to be shifted or interpolated. The current motion vector is passed by the CPU as a parameter to the vertex program (line 6). This is then added to the current input position to generate the offset texture coordinates for `Texture0`, the previous frame of the video sequence (line 11).

```
!!ARBvp1.0
....
2 OUTPUT oTexture0 = result.texcoord[0];
3 OUTPUT oTexture1 = result.texcoord[1];
4 ATTRIB iPosition = vertex.position;
6 PARAM vector = program.local[1];
....
11 ADD oTexture0, iPosition, vector;
12 MOV oTexture1, iPosition;
END
```

The following section of code shows the fragment program used to generate the DFD for the two textures. This fragment program uses the texture coordinates generated in the vertex program to sample the current pixel for each frame (lines 3 & 4). The texture unit will do bilinear interpolation when sampling the pixel from the texture data. These two values are then subtracted (line 5) and the absolute value of the difference is output as a color to the current rendering buffer (line 6).

```
!!ARBfp1.0
1 OUTPUT output = result.color;
2 TEMP frame1,frame2;

3 TEX frame1, fragment.texcoord[0], texture[0], RECT;
4 TEX frame2, fragment.texcoord[1], texture[1], RECT;
5 SUB frame1, frame1, frame2;
6 ABS output, frame1;
END
```

6.3.2.1 GPU FBM Results

In the FBM scheme used all blocks in the image were searched, there was no motion detection used to reduce the number of blocks searched. Also all possible vectors in the search space were checked. This means that the results are a poor indication of the performance of the block matching algorithm. However the goal of this work is to compare the performance of the GPU to

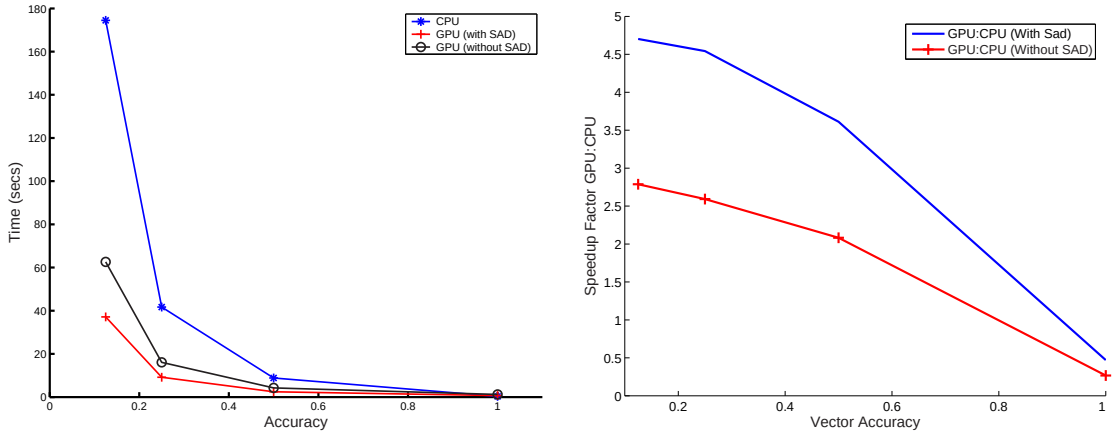


Figure 6.4: *GPU FBM using GeForce 5600 AGP test system at PAL resolution and various pixel accuracies. (a) Time in seconds for FBM using GPU and CPU. (b) Speed up factor of GPU over CPU for FBM.*

that of the CPU for similar heavy computational loads. It is the relative speed increase between a CPU only and a GPU-CPU implementation that is of interest.

GeForce 5600 AGP: The search space was ±4 pixels and the block size was 16×16 pixels. The experiments were carried out on the well known calendar/mobile sequence at PAL resolution. Figure 6.4 shows the average time in seconds to perform FBM using the GPU and the CPU. As can be seen for integer accurate motion vectors the CPU is faster than the GPU method. Because integer accuracy searching requires no interpolation, a very efficient implementation can be done on the CPU. For sub-pel accurate FBM however, there is a big difference in performance between the two. Generating the DFD on the GPU and reading it back to the CPU is on average 2.5 times faster for FBM than generating the DFD on the CPU. If the SAD for the blocks are calculated on the GPU, FBM is approximately four times faster than the CPU version.

GeForce 6800 PCIe: In this case the search space was ±10 pixels with a block size of 16×16 pixels. The results in Figure 6.5 show the speed up factor of the GPU over the CPU for various pixel accuracies. At integer accuracy the GPU is on average four times faster than the CPU implementation. At higher accuracies the difference is much more pronounced with the GPU 14-18 times faster than the CPU at certain resolutions. The time in seconds for the various vector accuracies are shown in Figure 6.6.

6.3.2.2 GPU FBM Accuracy

Figure 6.7(a) shows the PSNR of the motion compensated DFD at different motion vector accuracies for the caltrain sequence. For integer accurate motion vectors the results for the CPU and GPU versions are identical. For sub-pel accuracy there is a very slight difference between the CPU and GPU results, which cannot be distinguished on the graph. These plots also show

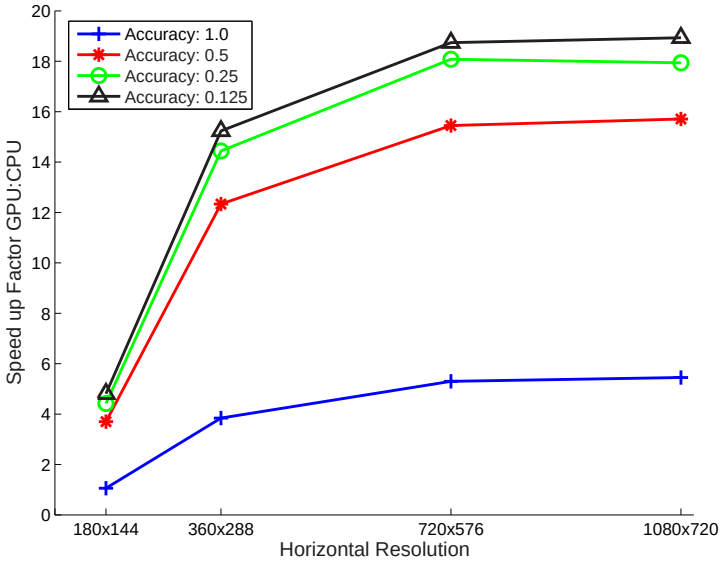


Figure 6.5: *Speed up factor of GPU over CPU for FBM at various pixel accuracies.*

the improvement in PSNR that comes from sub-pel accurate motion estimation. The PSNR increase as motion vector accuracy increases, which should lead to better coding performance.

Figure 6.7(b) shows a close up of the PSNR for the calendar/mobile sequence motion compensated at 0.25 pel accuracy. This shows the slight difference in results between the GPU and CPU implementations of FBM. In this case the mean error between the two is only 0.0042 dB. However due to some unresolved accuracy issues in the block summation method there are blocks for which the SADs are not equal. This difference in SAD is what causes the different motion vectors shown in Figure 6.8 and which gives the slight difference in PSNR between the CPU and GPU. It is mainly in flat areas however that this SAD error causes the motion vectors to be different. This is because in flat image areas there is not enough gradient or texture information to yield a unique match for motion. In such blocks all candidate motion vectors yield similar SAD measures. Therefore a slight change in the SAD for one candidate vector can cause a different motion vector to be chosen on the CPU and GPU. However as the motion vectors for these areas tends to be poor anyway this has only a slight impact on performance. In blocks where there is good image structure a slight difference in SAD has a much smaller impact on the motion vector chosen.

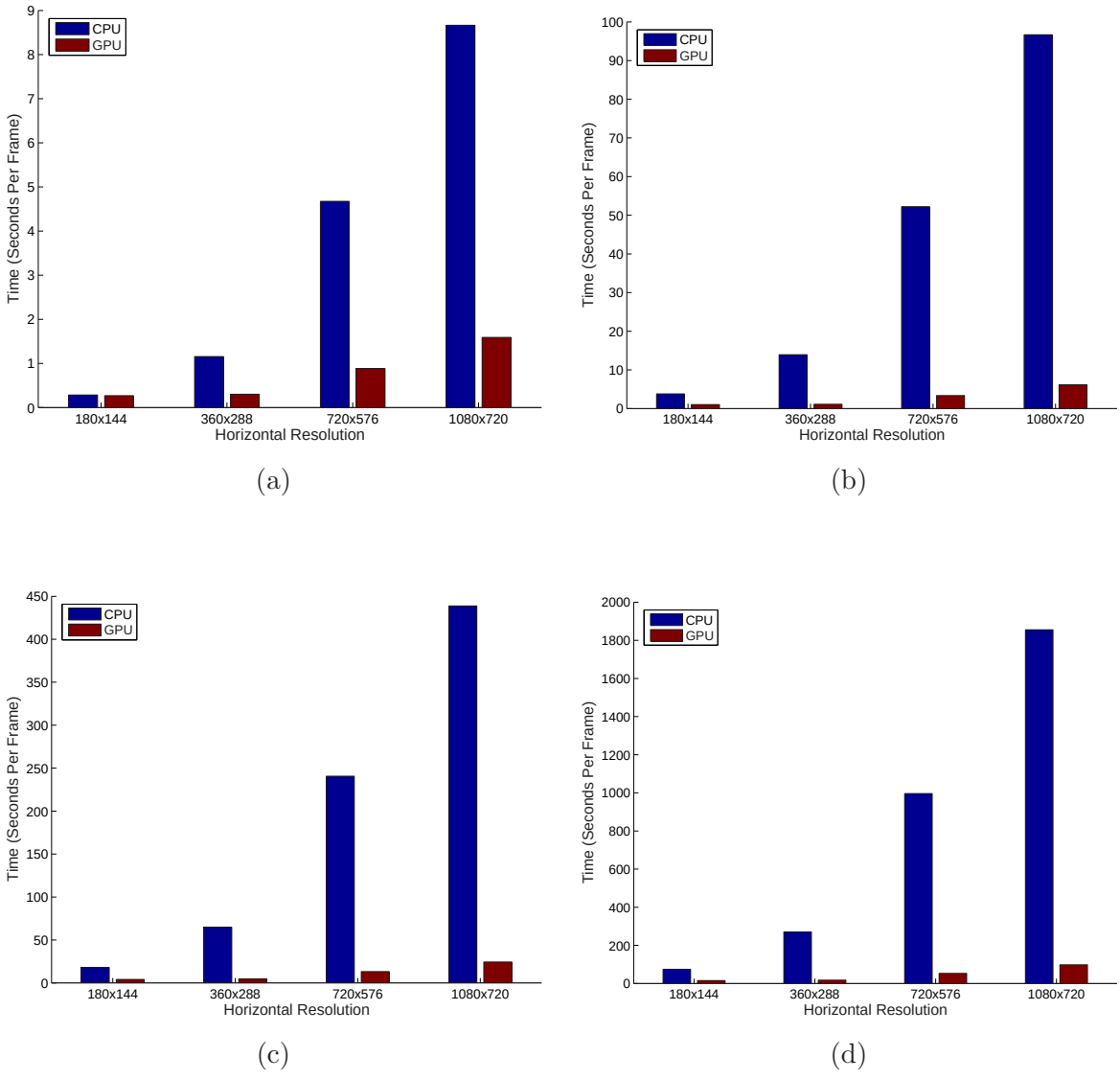


Figure 6.6: *Average time in seconds for FBM at various motion vector accuracies. (a) Accuracy = 1.0. (b) Accuracy = 0.5. (c) Accuracy = 0.25. (d) Accuracy = 0.125.*

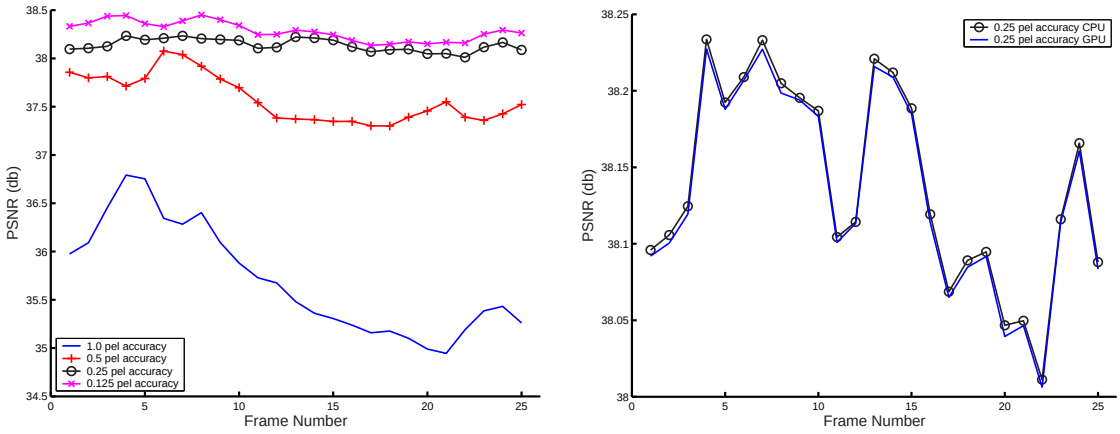


Figure 6.7: *GPU FBM results for caltrain sequence. (a) PSNR at various pixel accuracies. (b) Close up of PSNR for 0.25 pel accuracy. As shown there is very little difference between the CPU and GPU results.*

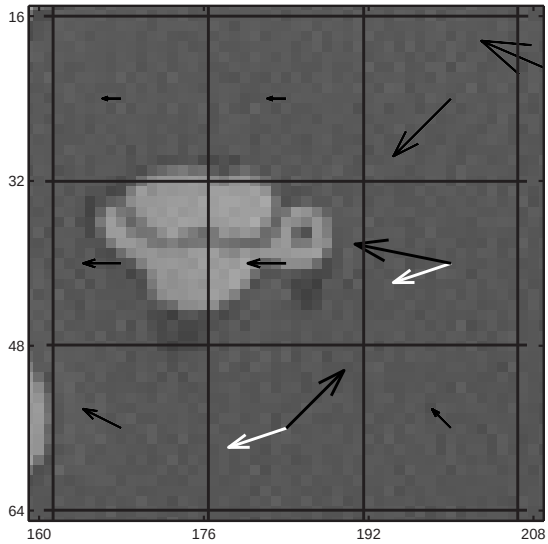


Figure 6.8: *Section of vector field for caltrain sequence. CPU vectors are in black and GPU vectors are in white. Vectors are scaled for clarity. In sections with low image gradient, the CPU and GPU can give different motion estimates. However in either case the motion estimate in these areas tend to be wrong and the difference between CPU and GPU is not relevant.*

6.3.3 Gradient Based Motion Estimation using the GPU [6]

Recall from chapter 2 the Wiener Based Motion Estimation algorithm (WBME) is an iterative procedure which computes the following update at each iteration

$$\begin{aligned} \mathbf{u}_i &= [\mathbf{G}^T\mathbf{G} + \mu\mathbf{I}]^{-1}\mathbf{G}^T\mathbf{z} \\ \mu &= \frac{\sigma_{ee}^2}{\sigma_{uu}^2} \end{aligned} \tag{6.1}$$

The motion estimate for the next iteration is then given by $\mathbf{d}_{i+1} = \mathbf{d}_i + \mathbf{u}_i$. Figure 6.9 shows a flow diagram depicting the operation of the WBME algorithm. Using the current estimate for the displacement of the block, $\mathbf{d}_i$, the DFD and gradients of the block are generated. These results are then grouped into the $\mathbf{z}$ vector and $\mathbf{G}$ matrix. An update for the displacement, $\mathbf{u}_i$, is then generated. A convergence test is finally performed to see if it is time to stop estimation for the block. There are usually various different checks to see when convergence is reached, including

- Stop if the magnitude of the update is below some threshold.
- Stop if the number of iterations passes some predefined limit.
- Stop if the mean square error of the DFD for the block is below some threshold.
- Stop if current mean square error is less than previous one.

If any one of these tests fail the current estimate for the displacement is output as the motion vector for that block. If all tests are passed then a new estimate for the displacement is generated using the current update and the algorithm is repeated to generate a new update.

To exploit the processing power of the GPU for WBME it is necessary to modify this approach a little. The GPU is designed for working on large volumes of data. Fragment programs operate independently for each pixel in a texture. Typically there will be more than one pixel pipeline executing fragment programs in parallel. Utilising this processing power efficiently involves operating on all the pixels in the image at once, rather than working on a block-by-block basis as on the CPU. There are two main bottlenecks in the WBME as implemented on the CPU. These are generating the DFD and gradients for each block, which requires image interpolation, and generating the matrix multiplications for the update equation. The image interpolation can be done quickly on the GPU if the images are treated as textures.

Generating the DFD and gradients requires motion compensation. In FBM it is possible to test all blocks with each vector in the search space at the same time. This means that the whole image can be translated by the corresponding vector and an error calculated for each block. In

[6] Results from this section have been published as: Graphics Hardware for Gradient Based Motion Estimation. Francis Kelly and Anil Kokaram. *In Proceedings of SPIE Conference on Embedded Processors for Multimedia and Communications*, January 2004 [95].

the WBME however at any one iteration blocks will have a different motion vector to test and so this approach is not possible. Instead each block has to be motion compensated for its current estimate of the motion. The following section outlines how this is done.

6.3.3.1 Block Based Motion Compensation

A 2D grid of vertices representing the blocks in the image is generated. A corresponding grid of texture coordinates is also established. Using these coordinates the image is texture mapped onto the grid. Offsetting the coordinates of blocks on the grid effectively means displacing the blocks in the image. The GPU will sample the blocks from the offset position before applying the texture. Each blocks texture coordinates can be offset to correspond to the motion vector for that block. The GPU can perform bilinear interpolation while sampling from the texture if necessary. In this way it is possible to compensate all the blocks in the image at once. Using fragment programs, the per-pixel operations of generating the DFD and gradients can be done quickly on the compensated image. Generating the DFD then involves using the same fragment program as for FBM. The shader programs for generating the image gradients are detailed next.

6.3.3.2 Image Gradients

Generating image gradients on the GPU proceeds as follows. Here the gradients are calculated using the central difference formula

$$\begin{aligned} \nabla I_{n-1}(\mathbf{x}) &= [\frac{\partial I_{n-1}(\mathbf{x})}{\partial x} \ \frac{\partial I_{n-1}(\mathbf{x})}{\partial y}] \\ \frac{\partial I_{n-1}(\mathbf{x})}{\partial x} &= \frac{I_{n-1}(x+1,y) - I_{n-1}(x-1,y)}{2} \\ \frac{\partial I_{n-1}(\mathbf{x})}{\partial y} &= \frac{I_{n-1}(x,y+1) - I_{n-1}(x,y-1)}{2} \end{aligned} \tag{6.2}$$

In order to calculate the gradient at the current pixel, the pixels directly above and below, and to the left and right of the current pixel are needed. This can be done using the following vertex program to change the texture coordinates of four texture units. This program outputs four sets of texture coordinates, two each for the vertical and horizontal gradient calculation (lines 13-16).

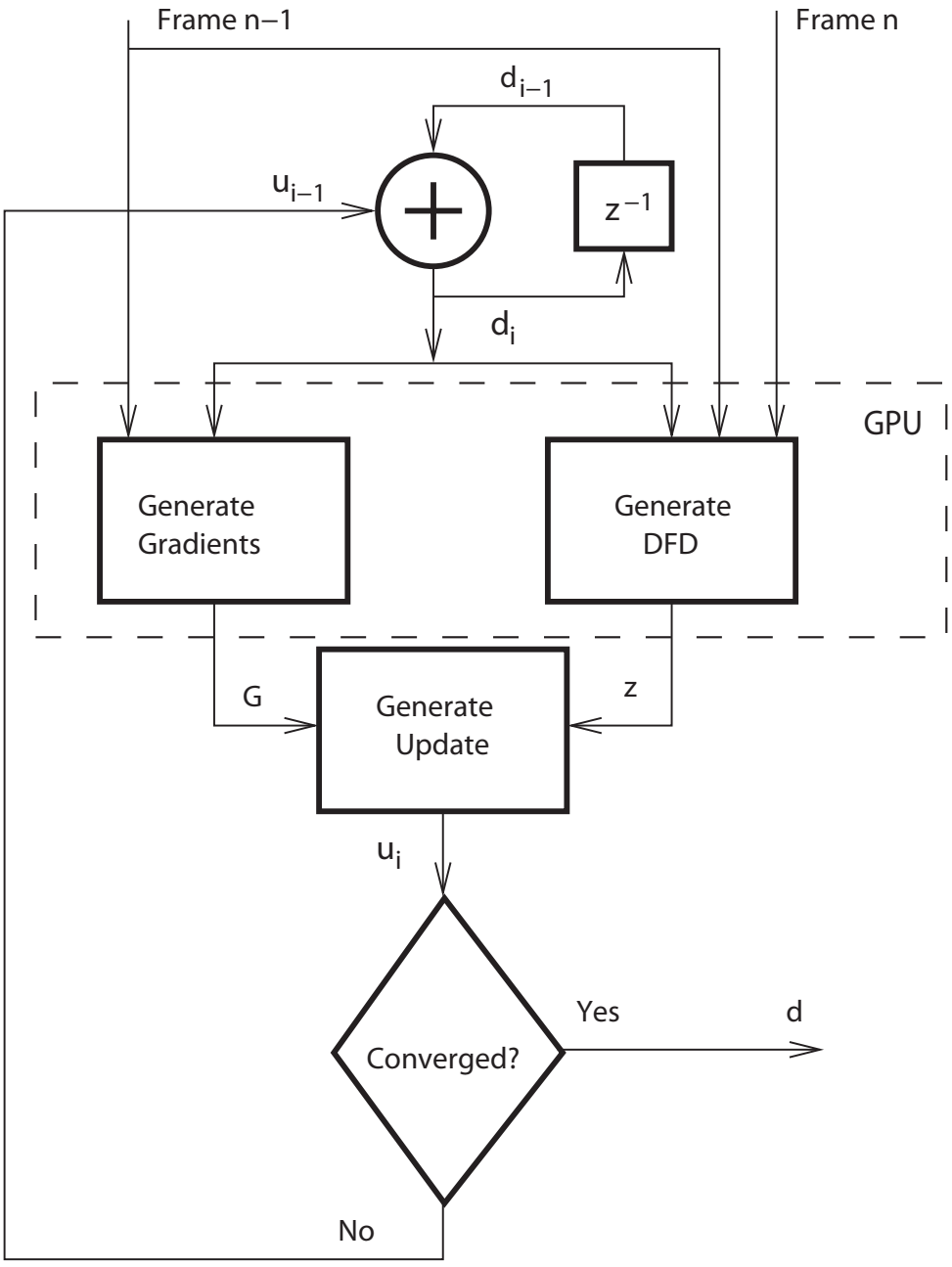


Figure 6.9: *Flow diagram for gradient based motion estimation.*

```
!!ARBvp1.0
1 OUTPUT oPos = result.position;
2 OUTPUT oTex0 = result.texcoord[0];
3 OUTPUT oTex1 = result.texcoord[1];
4 OUTPUT oTex2 = result.texcoord[2];
5 OUTPUT oTex3 = result.texcoord[3];
6 ATTRIB iPos = vertex.position;
7 ATTRIB iTex0 = vertex.texcoord[0];
⋮
13 ADD oTex0, iTex0, {1.0, 0.0, 0.0, 0.0};
14 ADD oTex1, iTex0, {-1.0, 0.0, 0.0, 0.0};
15 ADD oTex2, iTex0, {0.0, 1.0, 0.0, 0.0};
16 ADD oTex3, iTex0, {0.0, -1.0, 0.0, 0.0};
END
```

A fragment program similar to the one for subtracting two images is then used to generate the horizontal and vertical gradients for the image. It uses four sets of texture coordinates, generated in the vertex program, to access the pixels required for the gradient operation. The correct pixels needed for the gradient of the current pixel, according to (6.2), are then subtracted and the gradients are output to two channels[7] of the output Pbuffer.

```
!!ARBfp1.0
1 OUTPUT output = result.color;
2 TEMP grad,frame1,frame2;

3 TEX frame1, fragment.texcoord[0], texture[0], RECT;
4 TEX frame2, fragment.texcoord[1], texture[0], RECT;
5 SUB grad, frame1, frame2;
6 MUL output.x, {0.5, 0.5, 0.5, 0.5}, grad;

7 TEX frame1, fragment.texcoord[2], texture[0], RECT;
8 TEX frame2, fragment.texcoord[3], texture[0], RECT;
9 SUB grad, frame1, frame2;
10 MUL output.y, {0.5, 0.5, 0.5, 0.5}, grad;
END
```

Having generated the DFD and gradients for each block in the frame, the following section outlines how the matrix calculations, G^TG and G^Tz, may be implemented on the GPU.

[7]Since each pixel represents a colour it has four colour channels, red, green, blue, and alpha, and can hence store four values. This can be exploited to process four values at once or to store up to four output values from a fragment program.

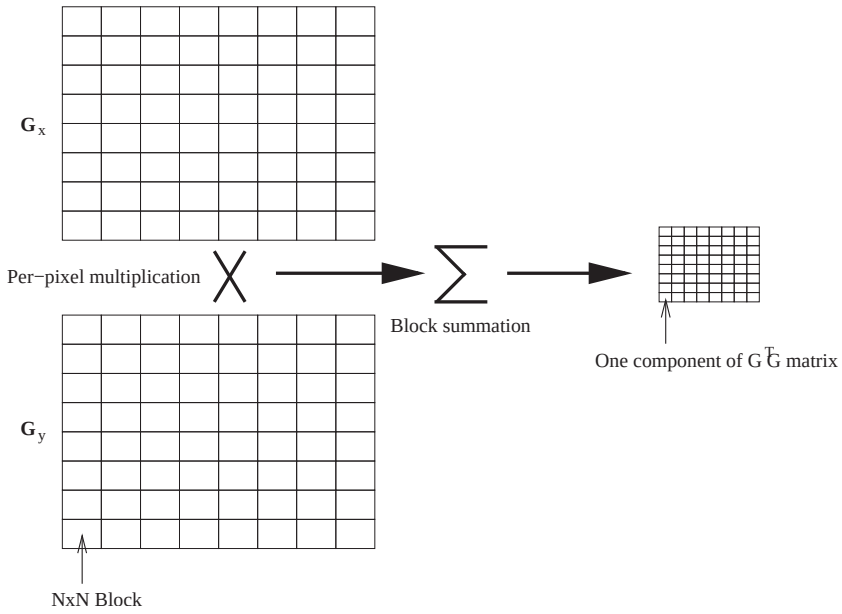


Figure 6.10: *Generating G^TG matrix for all blocks in parallel. The two gradient images, G_x and G_y, are multiplied pixel-by-pixel. The resulting image is summed up on a block basis yielding the second and third elements of the G^TG for each block as given by (6.4). A similar operation for the first and fourth elements is obtained using the G_x and G_y images respectively.*

6.3.3.3 Generating G^TG and G^Tz

Instead of finding a motion vector for each block before moving onto the next block, an update is generated for each block in the image at each iteration. In this way the GPU can be used to accelerate the generation of the updates for each block. This involves taking advantage of the structure of the **G** and **z** matrices. For a block size of $N \times N$ pixels the composition of these matrices is as follows

$$\mathbf{z} = \begin{bmatrix} DFD(\mathbf{x}_1, \mathbf{d}_i) \\ DFD(\mathbf{x}_2, \mathbf{d}_i) \\ \vdots \\ DFD(\mathbf{x}_{N^2}, \mathbf{d}_i) \end{bmatrix}$$

and (6.3)

$$\mathbf{G} = \begin{bmatrix} \frac{\partial}{\partial x} I_{n-1}(\mathbf{x}_1 + \mathbf{d}_i) & \frac{\partial}{\partial x} I_{n-1}(\mathbf{x}_1 + \mathbf{d}_i) \\ \frac{\partial}{\partial x} I_{n-1}(\mathbf{x}_2 + \mathbf{d}_i) & \frac{\partial}{\partial x} I_{n-1}(\mathbf{x}_2 + \mathbf{d}_i) \\ \vdots & \vdots \\ \frac{\partial}{\partial x} I_{n-1}(\mathbf{x}_{N^2} + \mathbf{d}_i) & \frac{\partial}{\partial x} I_{n-1}(\mathbf{x}_{N^2} + \mathbf{d}_i) \end{bmatrix}$$

Looking at the resulting G^TG matrix shows the following structure

$$G^TG = \sum_{p=1}^{N^2} \begin{bmatrix} G_x^2(p) & G_x(p)G_y(p) \\ G_x(p)G_y(p) & G_y^2(p) \end{bmatrix} \tag{6.4}$$

where $G_x(p)$ and $G_y(p)$ are the horizontal and vertical gradients of the block with displacement, $\mathbf{d}_i$. Generating G^Tz is a similar problem. In this case $\mathbf{z}$ is a $N^2 \times 1$ element matrix. This results in a 2×1 matrix as follows

$$G^Tz = \sum_{p=1}^{N^2} \begin{bmatrix} G_x(p)z(p) \\ G_y(p)z(p) \end{bmatrix} \tag{6.5}$$

where $z(p)$ is the p^{th} element of the DFD for the block with displacement, $\mathbf{d}_i$, and G_x, G_y are the gradients of the displaced block as before.

As (6.4) and (6.5) show, these matrix multiplications may be generated by a point-wise multiplication of the matrices followed by a summation. This multiplication can be done very efficiently on the GPU in a fragment program. Figure 6.10 shows how the second and third elements of the G^TG matrix can be generated for all the blocks in the image simultaneously. Firstly the image gradients for the compensated frame I_{n-1} are generated on the GPU, using the vertex and fragment programs described earlier. These are saved in Pbuffers as two textures, G_x and G_y. These textures are then passed through the graphics pipeline again and a different fragment program is used to multiply the two images together pixel-by-pixel. Finally the image is summed up on a block basis, giving a single value for each block in the image. This summation can be done either on the graphics card or on the CPU. Calculating the other elements of G^TG and G^Tz is performed in a similar manner.

To do the summation on the GPU the results of the matrix multiplication must be saved as textures in a Pbuffer. The sum-reduction operation outlined in chapter 5 is then used for the block summation. For a block size of $N \times N$ pixels, $\log_2 N$ passes are required. Alternatively the gradient and DFD images can be read back to the CPU for summation. On graphics hardware with an AGP bus reading these images back was very time consuming due to the PCI bandwidth problem. In this case it was found to be faster to do the summation on the GPU than reading back the full images across the slow PCI bus. Doing the summation on the GPU meant that only value per block was transferred over the bus, rather than one value per pixel.

6.3.4 Multi-resolution and the GPU

As noted in chapter 2 a multi-resolution scheme is often used in motion estimation algorithms. The low-pass filtering is usually performed with a gaussian filter kernel. The Intel IPP library

has a fast function for pyramid generation which uses the following gaussian kernel

$$K = \frac{1}{256}\begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix} \tag{6.6}$$

This corresponds to a 2-D gaussian with variance 1.0. A simpler low-pass filter to use is the box filter[8]. This simply replaces each pixel by a local mean

$$K = \frac{1}{4}\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \tag{6.7}$$

Modern graphics hardware have built in pyramid generation schemes which use a box filter to low-pass filter each level. This is known as automatic mipmap generation. Mipmaps is the name in the graphics industry given to a hierarchical representation of textures. While downloading textures to the graphics hardware the GPU can be set to perform automatic mipmap generation. The GPU can then choose the correct mipmap level to be used depending on the size of the object that the texture is being mapped to. The Intel IPP library pyramid generation scheme is highly optimised for the Pentium processor and does not have any effect on the motion estimator speed. However, using the built-in pyramid generator with the GPU makes implementation slightly more straightforward. The images for the current frames only have to be downloaded to the graphics hardware once at the start of each frame, rather than downloading each level of the pyramid separately.

6.3.4.1 GPU WBME Results

Figure 6.11 shows a sample frame from the four test sequences used. These are the well known caltrain sequence, foreman sequence, and salesman sequence. Also included is a very noisy clip from a 1911 Irish silent movie, called Rory O' More. These sequences consisted of three different frame resolutions, 180x144, 360x288, and 720x576, and were used in testing the GeForce 5600 AGP system. A fifth, high definition sequence, was used while comparing execution times for the GeForce 6800 PCIe system, with resolutions of 1080×720 and 1920×1080. A three level hierarchical WBME scheme was used in all cases, with the maximum number of iterations at each level set to ten.

GeForce 5600 AGP: Table 6.2 shows the average frames per second for WBME on both the CPU and GPU. The GPU implementation is approximately twice as fast as the CPU implementation, for the three different resolutions tested. Real-time performance of 25 frames per second is achieved at QCIF resolution.

[8]While acknowledging subsequent algorithm performance issues.

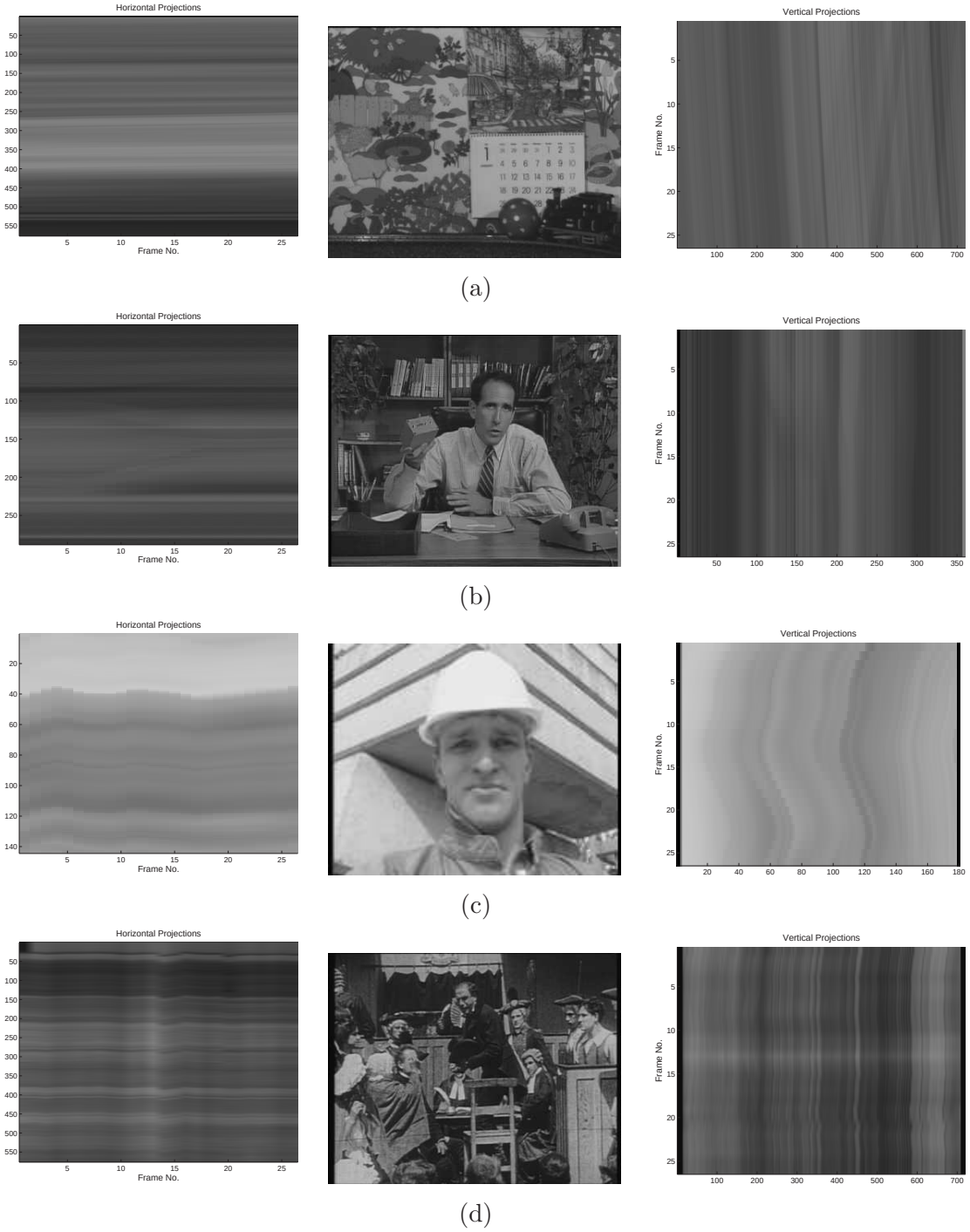


Figure 6.11: *Frames and TIP images from the four test sequences used. (a) Caltrain. (b) Salesman. (c) Foreman. (d) Rory.*

Frame Size	WBME CPU	WBME GPU	Speedup Factor
720 × 576	1.6	3.4	2.1
360 × 288	4.2	12.0	2.9
180 × 144	18.3	26.5	1.5

Table 6.2: *NVIDIA GeForce 5600 AGP WBME results. Results are average speed in frames per second.*

Frame Size	WBME CPU	WBME GPU	Speedup Factor
1920 × 1080	0.6	1.0	1.6
1080 × 720	1.5	2.5	1.6
720 × 576	3.6	4.7	1.3
360 × 288	13.7	13.7	1
180 × 144	53.9	27.4	0.5

Table 6.3: *NVIDIA GeForce 6800 PCIe WBME results. Results are average speed in frames per second.*

GeForce 6800 PCIe: Table 6.3 shows the results for WBME on the Intel Dual-Xeon processor system with GeForce 6800 PCIe graphics card. The speed up of the GPU over the CPU ranges from 1.3 times at PAL resolution to 1.6 times at high definition. For lower resolutions the CPU is as fast or faster than the GPU. In these cases the overhead associated with using the GPU, such as the block summation, dominates the GPU timings. This suggests that for this application at least, the GPU should be only used for sequences at PAL resolution or higher.

While at higher resolution the GPU is faster than the CPU the results are a bit unexpected with the speedup over the CPU not as great as before. One possible reason for this is the increased processing power of the CPU in this system compared to the previous one. The WBME is quite a complex algorithm and is more suited to CPU implementation; it does not seem to scale as well with a faster GPU as with a faster CPU. This is demonstrated by the relative increase in speed of the CPU results on the Dual-Xeon system over the CPU results on the old system. On the newer system it is approximately 3 times faster. Compare this to the GPU where there is only a slight increase in performance with the new graphics card. This also indicates that the GPU implementation is not as efficient as it could be. The implementation here was designed with the limited bandwidth of the AGP bus in mind. Changing this to exploit the increased bandwidth of the PCIe bus may yield further improvements in speed.

6.3.4.2 GPU WBME accuracy

Figure 6.12 details the peak signal to noise ratio (PSNR) of the motion compensated DFD for four test sequences. There are four sets of results for each sequence: a CPU implementation and

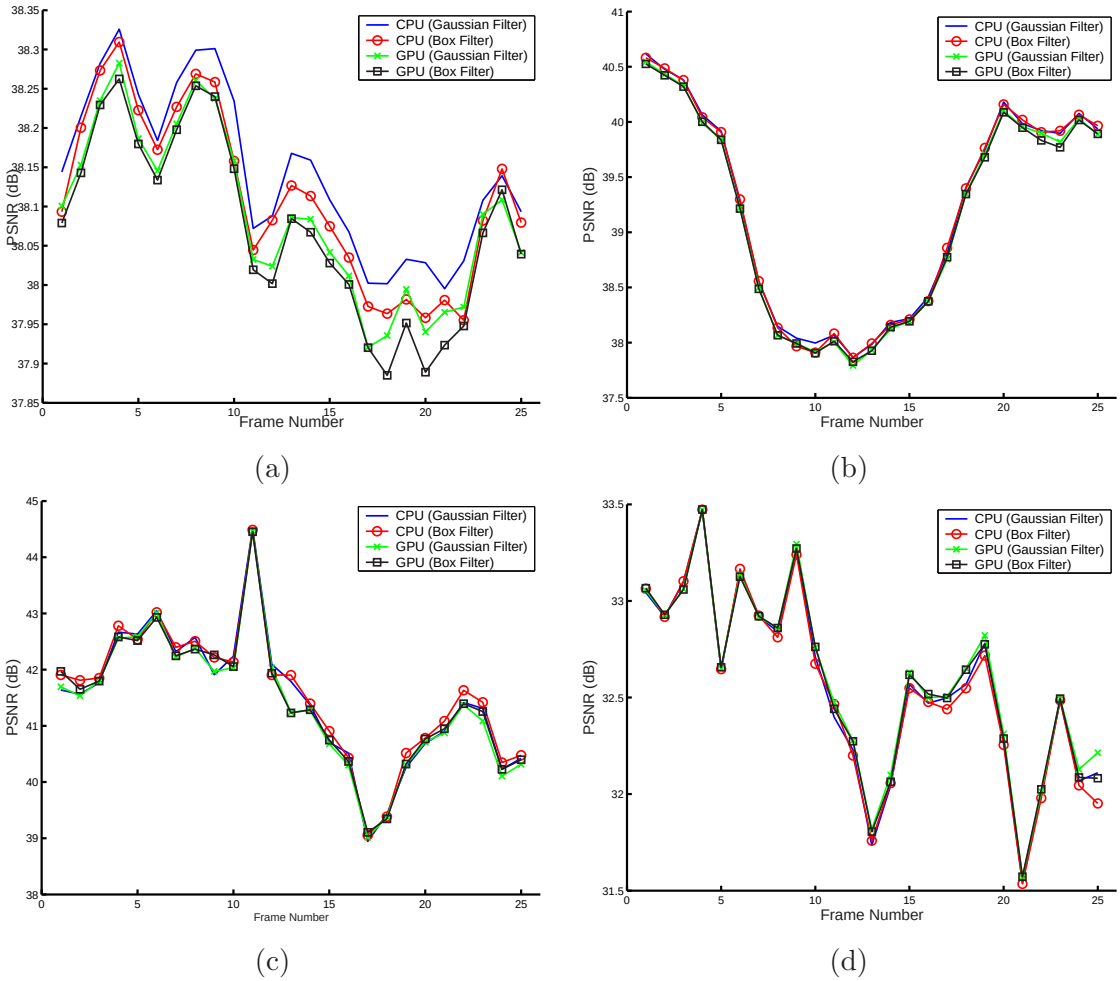


Figure 6.12: *WBME PSNR results for the four test sequences. (a) Caltrain. (b) Salesman. (c) Foreman. (d) Rory.*

GPU implementation using both gaussian and box filtering in the pyramid generation scheme. Looking firstly at the CPU results, the effect of the gaussian filtering kernel and the box filter in pyramid generation for a multi-resolution WBME algorithm can be compared. As shown there is very little difference in PSNR between the two filters. This suggests that box filtering is sufficient for pyramid generation in multi-resolution WBME, in particular for video coding where PSNR is a good indicator of performance. It should be noted that the accuracy of the motion vectors using the two filters may be different. This is not always evident in a PSNR comparison where different motion vectors can yield similar low prediction errors. However only one of these vectors may correspond to the true motion of the block. In applications such as video restoration and frame-rate conversion vector accuracy can be more important than PSNR performance.

A gaussian filter is typically recommended when generating a hierarchical pyramid for use in multi-resolution motion estimation schemes for reasons outlined in chapter 2. If the PSNR performance using the box filter is acceptable then the GPU may be used to automatically generate the pyramid by downloading the images to the GPU as textures with mipmap generation enabled. The GPU can also choose the appropriate pyramid level automatically. By drawing the grid of blocks with resolution corresponding to the current pyramid level, the GPU will choose the correct mipmap level to sample the pixels from. This means the multi-resolution algorithm can be implemented on the GPU with minimal extra effort.

A comparison between the GPU and CPU PSNR performance also shows similar results. The differences in PSNR are due to the same unresolved inaccuracies in the block summation on the GPU which affected the FBM algorithm. The implementation of the sum-reduction method does not yield quite the same results as the CPU. Although not that important in a SAD calculation for instance, while generating the G^TG and G^Tz matrices this small error manifests as differences in the update equation. This leads to slightly different final vector values and hence the the inconsistencies in PSNR performance between CPU and GPU. It is expected that the inaccuracies in the GPU block summation can be resolved and hence identical results on the CPU and GPU should be achievable.

6.4 Flicker Stabilisation [9]

The flicker algorithm is based on work by Pitie et al. [104, 158]. Flicker may be described as the temporal fluctuation in the brightness intensity level of a sequence. This is illustrated in Figure 6.13 using a TIP image of the tunnel sequence. The variation in the brightness levels of the frames is clearly visible as bands on the projection image on the left. The projection image on the right shows the same sequence after Flicker stabilisation.

[9] Results from this section have been published as: A New Robust Technique for Stabilizing Brightness Fluctuations in Image Sequences. Francois Pitie, Rozenn Dahyot, Francis Kelly, and Anil Kokaram. *In 2^{nd} Workshop on Statistical Methods in Video Processing in conjunction with ECCV*, May 2004 [158].

(a) Before (b) After

Figure 6.13: *Horizontal temporal projections of tunnel sequence (a) before and (b) after flicker stabilisation.*

Flicker removal may be broken into two stages: estimation and compensation. The estimation stage involves temporal histogram matching to generate a set of coefficients describing the flicker between frames in the video sequence. These coefficients vary spatially over the image using splines as a parametric fit. Given this set of estimation coefficients an efficient compensation method can be implemented by using a look-up-table (LUT) followed by an interpolation for evaluating the compensating brightness function at the relevant pixel.

6.4.1 Flicker Compensation using Graphics Hardware

Flicker compensation is the most time consuming stage of the flicker restoration process. It was found that on average flicker compensation accounted for 80% of the total restoration time. It is possible to increase the performance of the flicker restoration process by using the GPU to perform the compensation stage.

$$I_R(\mathbf{x}) = \sum_{i}^{N} w(\mathbf{x} - \mathbf{x}_i) f^i(I_n(\mathbf{x})) \tag{6.8}$$

Flicker compensation involves evaluation of (6.8) for every pixel in the image. The estimation process yields the function $f()$. The weight function $w()$ is a spline function and is known in advance. To do this on the graphics card involves using textures as look up tables for the weights w, the mapping function f^i, and also a texture for the observed frame, I_n. For each frame the mapping function texture and the observed frame texture are uploaded to the GPU. The weight texture is uploaded only once, at initialisation. The compensation is then performed entirely on the GPU and only the compensated frame is read back from the GPU to the CPU. This is done by using a fragment program to lookup the mapping function texture based on the current pixel intensity. This is then multiplied by the correct weight, which is also looked up in a texture, and the result output to a Pbuffer texture. A multi-pass algorithm is done N times, each time the result is added to the previous result according to (6.8).

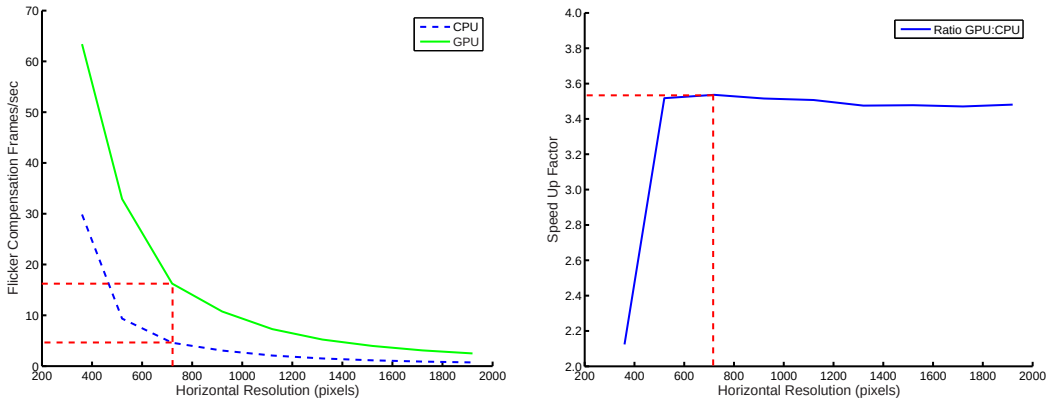
The following fragment code shows how the flicker compensation is performed. The image to be compensated is loaded into `texture[0]`, a LUT of the coefficients for the function $f()$ in `texture[1]`, and the weight function $w()$ in `texture[2]`. The image texture is sampled to get the intensity of the current pixel (line 5). Using this intensity and a texture coordinate passed from the vertex stage, a new texture coordinate is generated (lines 6 & 7). This new coordinate is then used to access the LUT (line 8). Finally the value for the weight at the current pixel site is obtained (line 9) and multiplied by the value from the LUT to yield the output pixel intensity for this pass (lines 10 & 11).

```
!!ARBfp1.0
1 OUTPUT output = result.color;
2 PARAM yOffset = program.local[0];
3 TEMP intensity, functionOfi, weight, newCoordinate;
4 PARAM scale = {255.0, 1.0, 1.0, 1.0};
5 TEX intensity.x, fragment.texcoord[0], texture[0], RECT;
6 ADD intensity.y, intensity.y, fragment.texcoord[1].y;
7 MUL newCoordinate.x, intensity.x, scale.x;
8 TEX functionOfi, newCoordinate, texture[1], RECT;
9 TEX weight, fragment.texcoord[2], texture[2], RECT;
10 MUL intensity, functionOfi, weight.x;
11 MOV output, intensity;
END
```

6.4.2 GPU Flicker Performance

GeForce 5600 AGP: Figure 6.14 (a) shows the results of using the GPU for flicker compensation compared to the CPU. These results are in frames per second (fps) and were obtained on a 1.6GHz Intel Pentium 4 machine, with an NVIDIA GeForce FX5600 graphics card. Using the GPU implementation reduced the time taken for flicker compensation from 80% of the total restoration time to 55%. At PAL resolution the GPU can achieve almost real-time performance with 16 fps, compared to 5 fps on the CPU. On average the GPU implementation is 3.5 times faster than the CPU implementation on this system, Figure 6.14 (b).

GeForce 6800 PCIe: Figure 6.15 illustrates the results using the Dual-Xeon NVIDIA GeForce 6800 GT PCIe system. Two sets of results are given: one for the total flicker reduction performance and one for just the flicker compensation part. Figure 6.15 (a) shows the relative performance of GPU and CPU implementations for flicker removal. At PAL resolution the GPU is capable of over 50 fps compared to 14 fps on the CPU. At film resolution of 2048×1536 the GPU can reach almost real-time with 22 fps. At this resolution the CPU can only achieve 2 fps. The speed up of the GPU over the CPU for the various resolutions tested is shown in Figure 6.15 (b). Figure 6.15 (c) shows the performance of the compensation stage only on the CPU



(a) (b)

Figure 6.14: *NVIDIA GeForce 5600 AGP Flicker Stabilisation. (a) Compensation frames per second for CPU Vs GPU. (b) The speed up factor of the GPU over the CPU. The performance at PAL resolution is illustrated with the dashed red lines.*

and the GPU. Here the difference between the two is even greater. At the highest resolution tested, 2048×1536, the GPU can compensate flicker at over 100 fps, while the CPU can only achieve 2 fps. The ratio of GPU to CPU fps for the compensation only is shown in Figure 6.15 (d) and gives an indication of the performance difference between the GPU and CPU.

6.5 Final Comments

This chapter has demonstrated the use of commodity graphics hardware as a useful co-processor to the CPU for some video processing applications. The results shown here indicate that there is huge potential in using the GPU for video processing. Two algorithms, flicker and video stabilisation, were shown to run much faster than real-time on standard definition and higher resolution sequences. This exceeds the performance of any commercial hardware solutions available today, for a fraction of the price.

There have been some recent developments in GPU technology which were not exploited during this work. The latest revision of the fragment shader standard, commonly know by its DirectX name of Shader Model 3.0 [131], supports looping and conditional flow control in the fragment processor. These features should allow for more complex operations to be performed in a single pass on the GPU. Also there is a new extension known as `ARB_framebuffer_object` [150] which replaces the Pbuffer framework. This extension is more efficient than Pbuffers when doing operations such as render-to-texture. It also more platform generic and easier to implement than the Pbuffer method.

A basic feature of the GPU, that was mentioned in passing but not exploited fully here, is the fact that the GPU is designed for efficient vector processing, i.e. four component vertex and colour data types. The algorithms outlined here were mainly concerned with processing

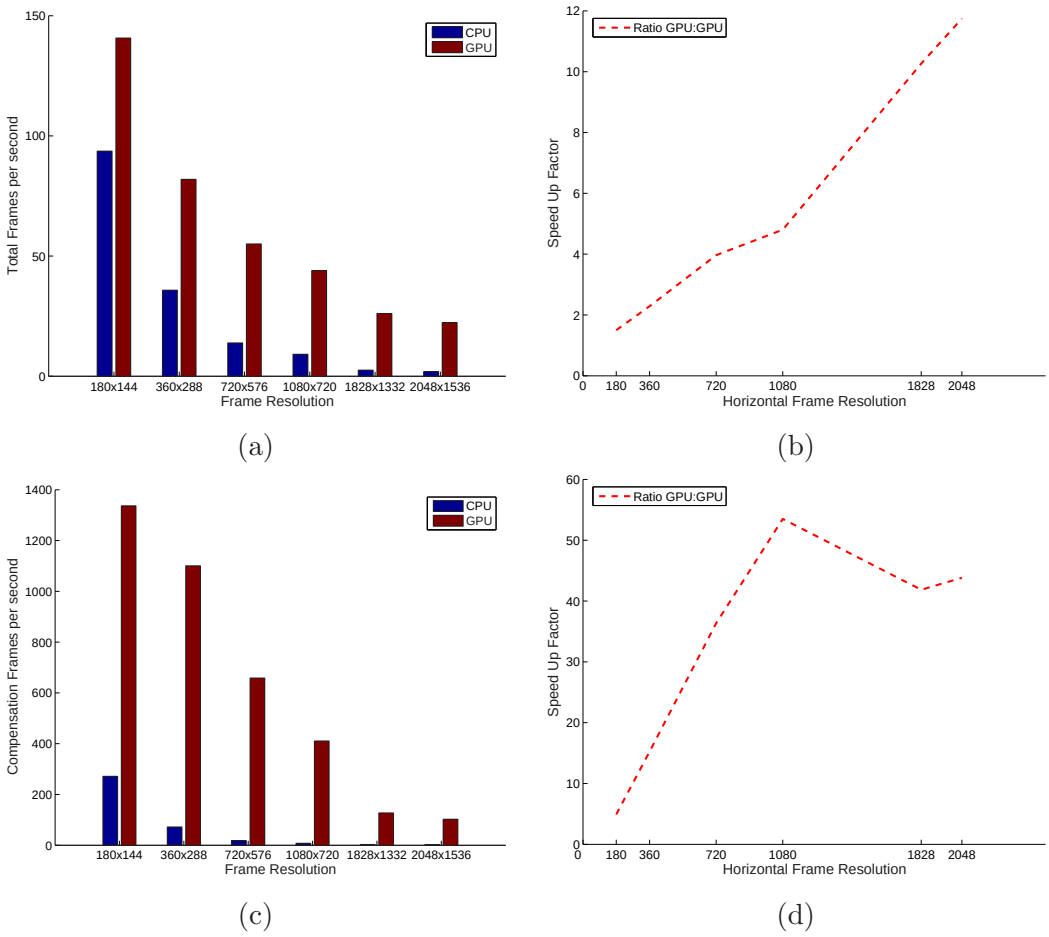


Figure 6.15: *NVIDIA GeForce 6800 PCIe Flicker Compensation Frames Per second for GPU v's CPU. (a) Total Flicker removal frames per second. (b) Total flicker removal speed up of GPU over CPU. (c) Compensation stage only frames per second. (d) Speed up factor for compensation only.*

luminance images. These were typically loaded into a single channel of a texture for processing on the GPU. Hence in some cases only 1/4 of the processing power available was being fully utilised. For example, it is possible to motion compensate four luminance frames at once by placing the frames in each of the four colour channels of a single texture. This would lead to a four-fold increase in speed over the results reported earlier. Alternatively, colour compensation can be performed with the same speed as those earlier results.

Although modern GPUs support 32-bit floating point processing, some accuracy issues arose during the course of this work. The cause of these inaccuracies are still unknown. However, identical results to those on the CPU are necessary if the GPU is to become a widespread option for general purpose computing. Future work involves resolving the accuracy issues which arose in the block summation methods on the GPU. Also algorithms which better exploit the new PCIe bus bandwidth and the fact that the GPU is a vector processor should be investigated.

7 Conclusion

This thesis focused on two main topics: a probabilistic approach to motion estimation and the potential of the GPU as a co-processor for fast video processing. As such the thesis was divided into two parts. However it was shown that these two areas are in fact complementary. While at this stage the speed of the new motion estimation algorithms is not paramount, they could easily benefit from the processing power of modern GPUs. The following sections give a brief review of the work presented in this thesis and propose some ideas for future work in both these areas.

7.1 Probablistic Motion Estimation

As well as providing a review of the state of the art in motion estimation, chapter 2 established that these methods can be unified within a Bayesian framework. It was shown how in this framework specific constraints, such as spatial smoothness, can be incorporated into the estimation process through the use of priors. This led to a classification of motion estimation algorithms depending on whether or not they include a prior in the estimation, yielding maximum-likelihood and maximum-a-posteriori (MAP) methods.

The MAP estimation approach was further investigated in chapter 3. This chapter presented a new algorithm for MAP motion estimation based on Belief Propagation (BP). This new algorithm also exploited a technique known as candidate selection for motion estimation. In a candidate selection algorithm only a limited number of candidate vectors are tested at each motion site. These candidates are chosen to maximise the probability that the correct motion

for a region will be estimated, subject to certain constraints. Candidate selection was shown to be a useful technique for simplifying MAP estimation while still providing better results than a maximum-likelihood algorithm. In motion estimation candidate selection was shown to be a powerful method for motion refinement, given some initial estimate of the motion field. The new BP based algorithm was shown to match and in some cases exceed the performance of another MAP estimation technique known as Iterated Conditional Modes (ICM). It also performed well when compared to a candidate selection based motion estimator called 3DRS (3D Recursive Search). Of course a larger set of test sequences and more application specific testing is necessary before making a full comparison with 3DRS.

Chapter 4 presented a new algorithm for on-line global motion estimation using Particle Filters. It was shown how this addressed some problems which can occur in maximum-likelihood based algorithms such as Iterative Re-weighted Least Squares (IRLS). In general the IRLS approach is effective at estimating the global motion in a scene. However in certain cases the image data can lead to ambiguity in the motion estimate. This chapter proposed a method for using the temporal evolution of the motion parameters via a particle filter to try to diagnose such cases. Two new algorithms were presented, one based on the Particle Filter alone which was denoted GMEPF (Global Motion Estimation with Particle Filter), and a hybrid algorithm which included the IRLS estimate called IRLSPF (IRLS with Particle Filter).

While the GMEPF could successfully estimate a small number of motion parameters, it typically needed a considerable number of particles. Hence it is not well suited for fast accurate global motion estimation. Instead the IRLSPF was considered as a tool for diagnosing the type of shake, random or impulsive, present in a sequence. The IRLSPF could track the underlying trend of the camera using relatively few particles, and by comparing the IRLS estimate to the IRLSPF estimate it was possible to distinguish between the two types of shake.

7.2 GPU Video Processing

It is established from chapter 5 that modern GPUs are very powerful processors. It was shown how the increased precision and programmability of these GPUs have helped to make the GPU a viable option for fast image and video processing. Chapter 6 then introduced the use of the GPU as an image co-processor for real-time video processing. The results presented in this chapter indicate that there is huge potential in using the GPU for video processing. Two algorithms, flicker and video stabilisation, were shown to run much faster than real-time on standard definition and higher resolution sequences. This exceeds the performance of any commercial hardware solutions available today, for a fraction of the price.

It was also shown how the GPU is not suitable for accelerating all algorithms. It is well suited to operations which can benefit from the highly parallelised nature of the GPU pipeline. As such the typical approach is to use the GPU to accelerate only those parts of an algorithm which are computational bottlenecks and which will map well onto the GPU. The CPU is then

used for the other tasks such as algorithm flow control and complex processing.

7.3 Future Work

Motion estimation is still a vibrant area of academic research which shows no signs of slowing down. Similarly the use of GPUs for general purpose computing is a new, fast growing area of research. This section outlines some possible extensions to the motion estimation algorithms presented here and to GPU video processing techniques.

Candidate Selection Motion Estimation

The performance of candidate selection for motion estimation gave very encouraging results. Using a small number of candidates, the BP and ICM methods were able to generate consistently good estimates, on a limited set of sequences, using both quantitative and qualitative measures. At this point it is unclear whether the test set and the measures are relevant. To further prove the usefulness of the proposed local motion estimators in a given application, more elaborate tests are necessary on test sets that are statistically relevant for that specific applications and using metrics that are meaningful to that application. Also the complexity of the proposed algorithms needs to be benchmarked as this is information important for many applications.

Future work in this area should include extending these methods to include temporal candidate vectors. Also worth considering is using the fast 3DRS algorithm as a 'kick start' to BP, ICM, and possibly WBME. The WBME could be used to refine the 3DRS estimate in problem areas such as complicated local motion. Finally, although the full ICM or BP algorithms are not suited to GPU implementation, certain parts of them could certainly benefit from GPU acceleration. For example generating the DFD for each candidate vector is an operation that has been shown to be very efficient on the GPU.

On-line Global Motion Estimation

The size of the parameter space associated with affine motion estimation means that the GMEPF algorithm in its current form is not well suited to this problem. The use of partitioned sampling as a means of reducing the complexity of the problem is one option. However it is not known how estimating each parameter separately will affect the final global motion estimate. This is an area that requires further research. Future work might also include investigating other means of incorporating history in global motion estimation. Finally the GPU can also be used here to accelerate the particle filter solution. Each particle requires the estimation of a likelihood function, which essentially involves image compensation and a difference calculation. This could be done very efficiently on the GPU. In this way a large number of particles could be used to yield an accurate estimate, or else a smaller number of particles could be used for a fast, less accurate solution.

GPU Video Processing

The GPU has two main advantages over dedicated hardware for video processing. These are the low cost and the high performance growth, both driven by the demand of the computer games industry. Currently the main disadvantages are the difficult programming environment and the limitations due to the GPU architecture. The former should become less of an issue with the development of high-level languages which hide the graphics specific nature of the underlying hardware from the programmer. The latter problem however is likely to remain for the immediate future at least as the GPU maintains certain architectural designs necessary for high performance graphics processing. However if the application can be efficiently implemented on this architecture, as in the case of the video stabilisation and flicker algorithms, then the GPU can offer very significant improvements in speed. In fact, it can out-perform expensive commercial video processing units.

The ongoing evolution of the GPU architecture means that GPGPU continues to be a broad area of research. In the particular domain of this thesis several immediate issues require further investigation. Firstly the accuracy issue which arose in the block summation methods needs to be resolved. Also better use of the bandwidth available with PCI Express needs to be incorporated into future algorithms. The fact that the GPU is a vector processor capable of operating on four elements at once can also be easily exploited to provide further increases in speed for the algorithms outlined in chapter 6. Finally the new shader programs with support for branching and dynamic flow control, and the use of framebuffer objects instead of Pbuffers, should also be explored.

It is hard to predict what direction the GPU will take in the medium to long term. One thing which is quite certain in the short term however, is that GPUs will continue to get more powerful. For example, the newest NVIDIA GPU at the time of writing, the GeForce 7800, released in the summer of 2005, offers double the performance of the GeForce 6800, the fastest GPU used in this work. It is unlikely that the GPU will ever approach the general processing model of a modern CPU. Hence it is not worthwhile suggesting new features which, though they might help real-time video processing on the GPU, would not benefit graphics processing. However, one possible extension which could do both would be to provide automatic mipmapping support for floating point textures. This would enable fast accurate block summation as outlined in chapter 5, one of the key operations for real-time video processing on the GPU identified here.

A

Affine Phase Correlation

The phase-correlation method estimates the relative shift between two images by means of a normalized cross-correlation function computed in the 2D spatial Fourier domain. This appendix outlines the phase correlation method for estimating the affine transformation between two images. This has been used for global motion estimation [73, 111]. Firstly it recalls the case where there is the transformation contains only a translational component.

A.1 Translation Only

The Discrete Fourier Transform (DFT) F of a 2D signal $f(x, y)$ of size $M \times N$ is given by

$$F(u, v) = \sum_{x=0}^{N-1} \sum_{y=0}^{M-1} f(x, y) e^{-j(2\pi/N)ux} e^{-j(2\pi/M)vy} \tag{A.1}$$

Let I_n and I_{n-1} be two frames that differ only by a displacement $[d_x, d_y]$ such that

$$I_n(x, y) = I_{n-1}(x + d_x, y + d_y). \tag{A.2}$$

Their corresponding Fourier transforms, F_n and F_{n-1}, will be related by

$$F_n(u, v) = e^{j2\pi(ud_x + vd_y)} F_{n-1}(u, v). \tag{A.3}$$

The cross correlation function between the two frames is defined as

$$c_{n,n-1}(x, y) = I_n(x, y) * * I_{n-1}(x, y) \tag{A.4}$$

where $**$ denotes the 2-D convolution operation. Taking the Fourier transform of both sides yields the complex-valued CPS expression

$$C_{n,n-1}(u,v) = F_n(u,v)F^*_{n-1}(u,v) \tag{A.5}$$

where F^* is the complex conjugate of F. Normalising $C_{n,n-1}(u,v)$ by its magnitude gives the phase of the CPS

$$\begin{aligned}\tilde{C}_{n,n-1}(u,v) &= \frac{F_n(u,v)F^*_{n-1}(u,v)}{|F_n(u,v)F^*_{n-1}(u,v)|} \\ &= e^{-j2\pi(ud_x+vd_y)}.\end{aligned} \tag{A.6}$$

The inverse Fourier transform of $\tilde{C}_{n,n-1}(u,v)$ yields the phase-correlation function

$$\tilde{c}_{n,n-1}(x,y) = \delta(x-d_x, y-d_y) \tag{A.7}$$

which consists of an impulse whose coordinates are located at $[d_x, d_y]$, the required displacement. Following on from this the next section considers estimation the transformation with both translational and rotational components.

A.2 Translation and Rotation

Let I_n and I_{n-1} be two frames that differ by a rotation θ_n and translation $[d_x, d_y]$ such that

$$I_n(x,y) = I_{n-1}(x\cos\theta_n + y\sin\theta_n + d_x, -x\sin\theta_n + y\cos\theta_n + d_y). \tag{A.8}$$

Using the fundamental properties of the Fourier transform, it can be shown that a rotation in the spatial domain corresponds to a rotation in the frequency domain. Also noting equation (2.29) the Fourier transforms of I_n and I_{n-1} may be written as

$$F_n(u,v) = e^{j2\pi(ud_x+vd_y)}F_{n-1}(u\cos\theta_n + v\sin\theta_n, -u\sin\theta_n + v\cos\theta_n). \tag{A.9}$$

Letting M_n and M_{n-1} be the magnitudes of F_n and F_{n-1}, Equation A.9 becomes

$$M_n(u,v) = M_{n-1}(u\cos\theta_n + v\sin\theta_n, -u\sin\theta_n + v\cos\theta_n). \tag{A.10}$$

Equation A.10 shows a very useful property of the frequency domain: the separability of the estimate of rotation from the estimate of translation, which is not possible in the spatial domain. If a polar coordinate system is used, the rotation can be found as a shift in the frequency domain.

$$M_n(\rho,\theta) = M_{n-1}(\rho, \theta-\theta_n) \tag{A.11}$$

This shift, or θ_n, may be found using the phase correlation technique outlined above.

A.3 Zoom Only

Let z_n be a zoom factor between frames I_n and I_{n-1} such that

$$I_n(x, y) = I_{n-1}(z_n x, z_n y). \tag{A.12}$$

According to the Fourier scale property, their Fourier transforms will be related by

$$F_n(u, v) = \frac{1}{|z_n|} F_{n-1}(u/z_n, v/z_n). \tag{A.13}$$

By converting the axis to a logarithmic scale and ignoring the multiplication factor $1/|z_n|$, the scaling may be reduced to a translational movement

$$F_n(\log u, \log v) = F_{n-1}(\log u - \log z_n, \log v - \log z_n). \tag{A.14}$$

This shift may then be found using standard phase correlation.

A.4 Translation, Rotation, and Zoom

Putting all these together allows the estimation of all three parameters as follows. Assume that the two frames I_n and I_{n-1} are related by

$$I_n(x, y) = I_{n-1}(z_n x \cos\theta_n + z_n y \sin\theta_n + d_x, -z_n x \sin\theta_n + z_n y \cos\theta_n + d_y) \tag{A.15}$$

Their corresponding Fourier transforms will be related by

$$F_n(u, v) = e^{j2\pi(ud_x + vd_y)} \frac{1}{|z_n|} F_{n-1}\left(\frac{u\cos\theta_n + v\sin\theta_n}{z_n}, \frac{-u\sin\theta_n + v\cos\theta_n}{z_n}\right) \tag{A.16}$$

Taking magnitudes and using polar representation of the axes yields

$$M_n(\rho, \theta) = M_{n-1}(\rho/z_n, \theta - \theta_n). \tag{A.17}$$

Using log-polar coordinates the zoom and rotation may be obtained using phase correlation

$$M_n(\log\rho, \theta) = M_{n-1}(\log\rho - \log z_n, \theta - \theta_n). \tag{A.18}$$

Frame I_n is then compensated for rotation and zoom and the translational component is found using phase correlation as outlined earlier.

A.5 Affine Theorem

An alternative way of representing the relationship between two frames undergoing an affine transformation is the affine theorem [19]. The affine theorem for the 2D Fourier transform is defined as follows: If $I_{n-1}(x, y)$ has 2D Fourier transform $F_{n-1}(u, v)$, then $I_n(x, y) = I_{n-1}(ax + by + c, dx + ey + f)$ has 2D Fourier transform

$$F_n(u, v) = \frac{1}{|\Delta|} exp\left\{\frac{i2\pi}{\Delta}[(ec - bf)u + (af - cd)v]\right\} \times F_{n-1}\left(\frac{eu - dv}{\Delta}, \frac{-bu + av}{\Delta}\right) \tag{A.19}$$

where the determinant is given by $\Delta = ae - bd$.

B

Belief Propagation

Belief Propagation (BP) is an exact inference method, proposed by Pearl [155], for estimating the marginal probabilities at the nodes in a belief network without loops, e.g. a singly connected MRF. For example, consider the system illustrated in Figure B.1. The nodes y_i represent the observations and the hidden nodes x_i represent the unknown states of the system, $Y = \{y_1, \ldots, y_N\}$, $X = \{x_1, \ldots, x_N\}$. The aim is to estimate the marginal probability $p(x_i = u|Y)$ for each hidden node in the network. If Figure B.1 represents a singly connected Markov network, then the Markovian property allows $p(x_i = u|Y)$ to be factored into three terms: one depending on the local data at i (L_i), another depending on data to the left of i (α_i) and a third depending on data to the right of i (β_i) [196]. Thus

$$p(x_i = u|Y) = c\alpha_i(u)L_i(u)\beta_i(u) \tag{B.1}$$

where

$$\alpha_i(u) = p(x_i = u|y_{1:i-1}) \tag{B.2}$$

$$L_i(u) = p(x_i = u|y_i) \tag{B.3}$$

$$\beta_i(u) = p(x_i = u|y_{i+1:N}) \tag{B.4}$$

and c denotes a normalization factor independent of u. This can be thought of as a message passing scheme in the network where the probability at the current node depends on messages representing the data to the left and right of the node, and the local probability of the node. How are these messages generated? Consider the term representing the data to the left of node

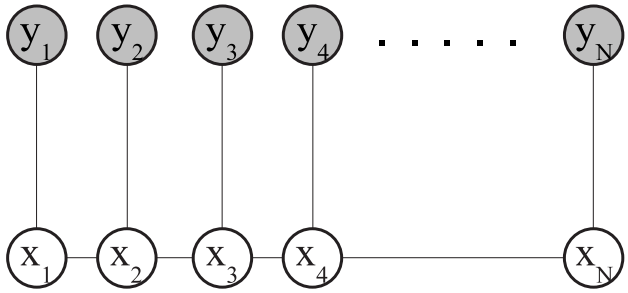


Figure B.1: *Simple Hidden Markov Model. The filled in circles represent the observed nodes y_i. The empty circles, nodes x_i, represent the hidden states of the system.*

i, α_i. This probability, $p(x_i = u|y_{1:i-1})$, may, using marginalization, be expressed as

$$\alpha_i(u) = p(x_i = u|y_{1:i-1}) = \int_v p(x_i = u, x_{i-1} = v|y_{1:i-1})dv. \tag{B.5}$$

The joint probability function in the integral may be factorised as

$$p(x_i = u, x_{i-1} = v|y_{1:i-1}) = \underbrace{p(x_{i-1}|y_{1:i-2})}_{message}\overbrace{p(x_i|x_{i-1})}^{prior}\underbrace{p(x_{i-1}|y_{i-1})}_{likelihood}. \tag{B.6}$$

Denoting the conditional distribution $C_i(u, v) = p(x_i = u|x_{i-1} = v)$, $\alpha_i(u)$ may be written in terms of $\alpha_{i-1}(v)$

$$\alpha_i(u) = c\int_v \alpha_{i-1}(v)C_i(u, v)L_{i-1}(v)dv \tag{B.7}$$

where c is another normalizing constant.

A similar equation can be written for $\beta_i(u)$ with a conditional distribution $B_i(u, v) = p(x_i = u|x_{i+1} = v)$,

$$\beta_i(u) = c\int_v \beta_{i+1}(v)B_i(u, v)L_{i+1}(v)dv. \tag{B.8}$$

This suggests a propagation scheme where nodes represent the probabilities given in the left hand side of equations (B.1) and (B.7), and updates are based on the right hand side, i.e. on the activities of neighbouring nodes.

There are two kinds of BP algorithms with different message passing rules: "max-product" and "sum-product", which maximise the joint posterior of the network, and the marginal distribution of each node respectively. These are outlined next.

B.1 Sum-Product

The sum-product method is also sometimes known as belief update. The sum-product algorithm computes the marginal distribution at each node and hence may be used for generating a MMSE

(Minimum Mean Square Error) estimate. In the sum-product algorithm the message m_{ij} that node i sends to node j is computed as

$$m_{ij}(x_j) \leftarrow \alpha \sum_{x_i} \psi_{ij}(x_i, x_y)\psi_i(x_i) \prod_{k \in \mathcal{N}(i)\backslash j} m_{ki}(x_i) \tag{B.9}$$

where α denotes a normalization constant and $\mathcal{N}(i)\backslash j$ means all nodes neighbouring node i, except j. ψ_{ij} represents the cost function between neighbouring nodes and is given by the prior distribution. ψ_i is the local evidence at a node and is given by the likelihood. The belief b_i (approximate marginal posterior probability) at node i is obtained by multiplying all incoming messages to that node by the local evidence.

$$b_i(x_i) \leftarrow \alpha\psi_i(x_i) \prod_{k \in \mathcal{N}(i)} m_{ki}(x_i) \tag{B.10}$$

B.2 Max-Product

The max-product algorithm is sometimes known as belief revision and is equivalent to the Viterbi algorithm [191, 57, 161]. The max-product version of BP computes a MAP estimate at each node and is used here to compare with the MAP estimate of ICM. In this method new messages are generated using

$$m_{ij}(x_j) \leftarrow \kappa \max_{x_i} \left(\psi_{ij}(x_i, x_j)\psi_i(x_i) \prod_{k \in \mathcal{N}(i)\backslash j} m_{ki}(x_i) \right) \tag{B.11}$$

and the belief is computed as follows

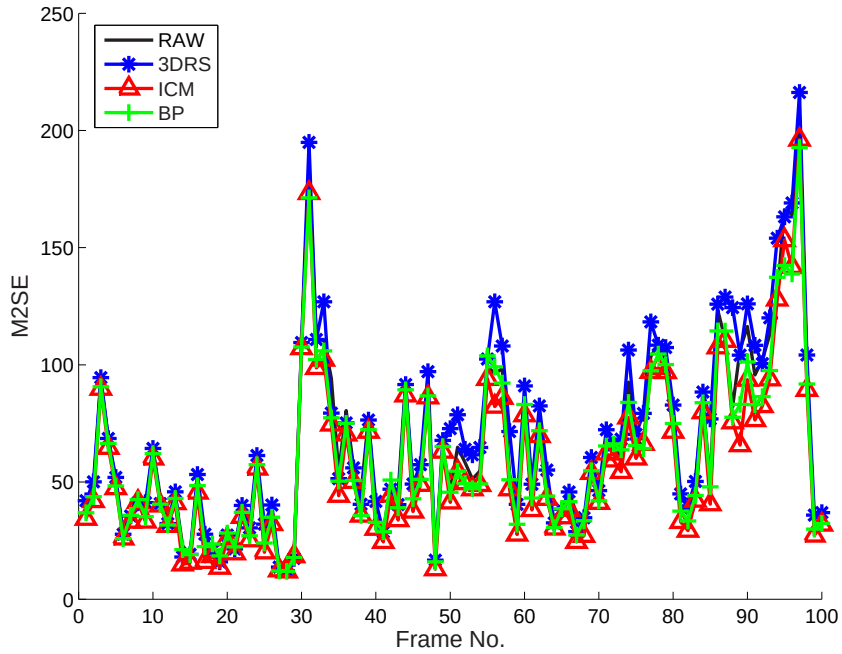
$$b_i(x_i) \leftarrow \kappa\psi_i(x_i) \prod_{k \in \mathcal{N}(i)} m_{ki}(x_i) \tag{B.12}$$

The MAP estimate is then given by

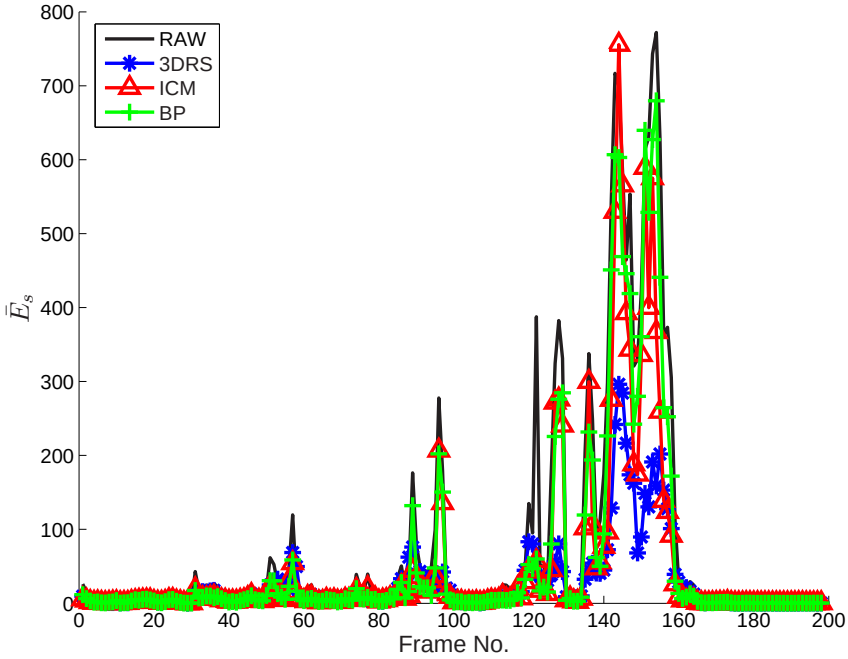
$$x_i^{MAP} = \arg\max_{x_k} b_i(x_k) \tag{B.13}$$

B.3 Additional Candidate Selection Motion Estimation Results

In chapter 3 a new motion estimation algorithm based on candidate selection and Belief Propagation was presented. Some more results from the experiments on this new algorithm are presented here. These include the M2SE error metric and a smoothness measure for vector fields resulting from four different motion estimators. These were a WBME, the 3DRS estimator, and an ICM candidate selection algorithm. Also given are typical samples of the vector fields each method produced to enable a qualitative comparison.

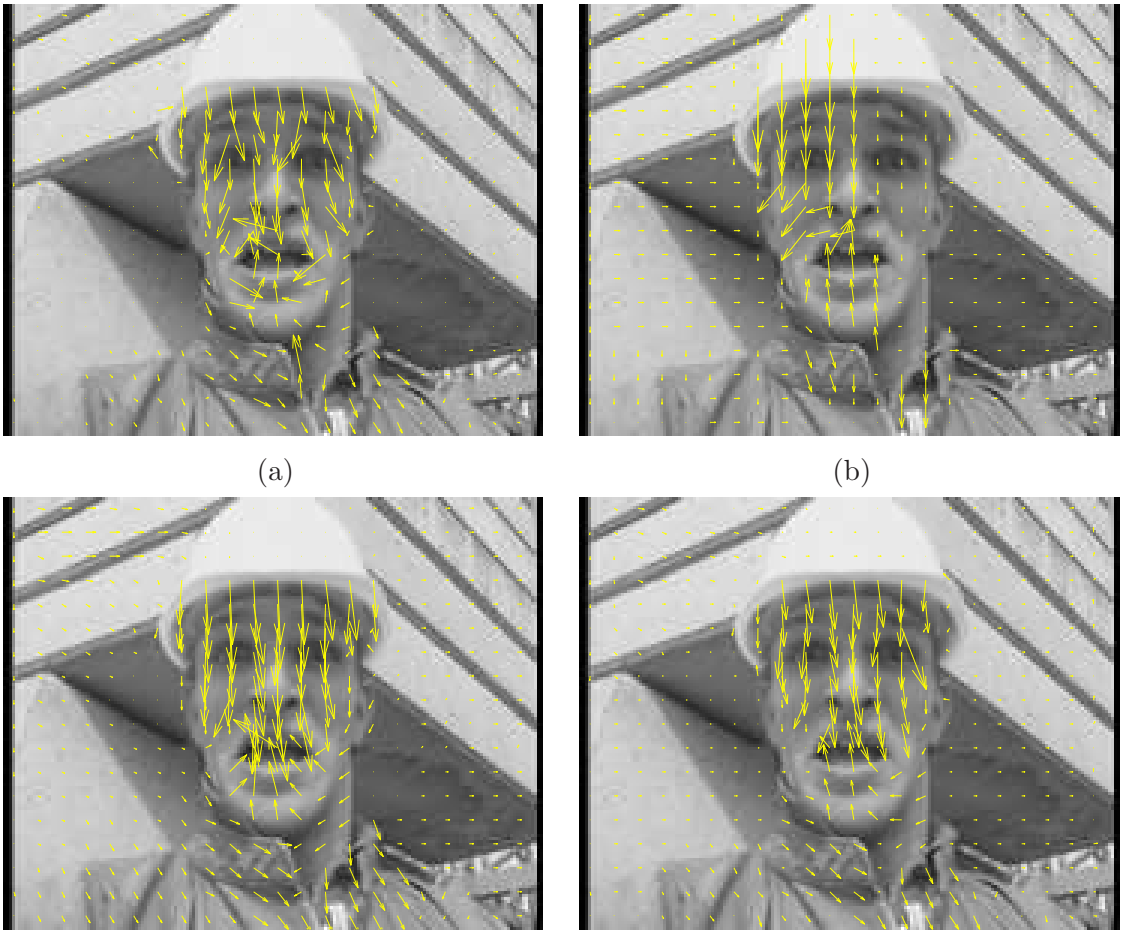


(a)



(b)

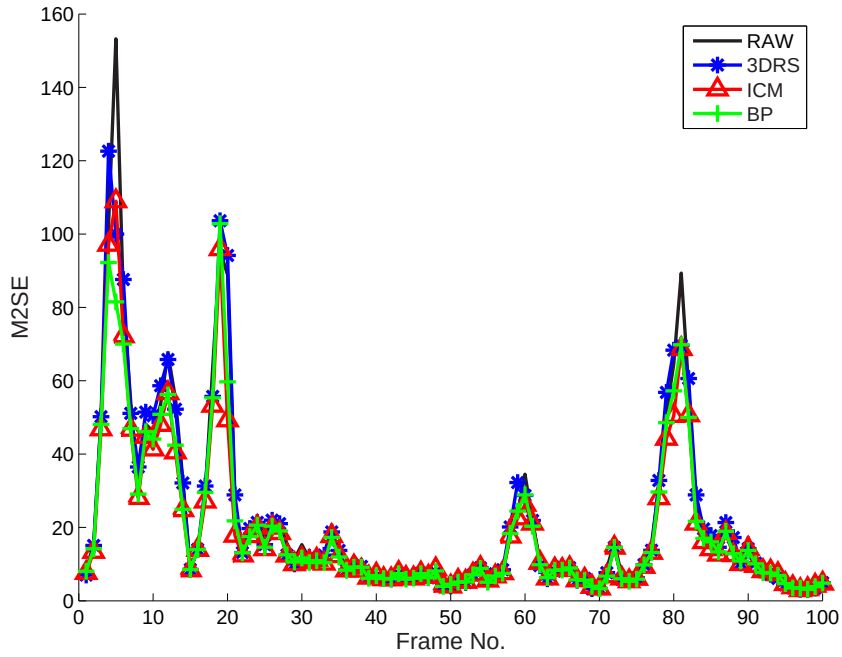
Figure B.2: *Foreman sequence (8x8 Blocks). (a) M2SE for 100 frames of the foreman sequence. The mean value for the zero estimate is 223. (b) Measure of the Smoothness ($\bar{E}_s$) of the motion field.*



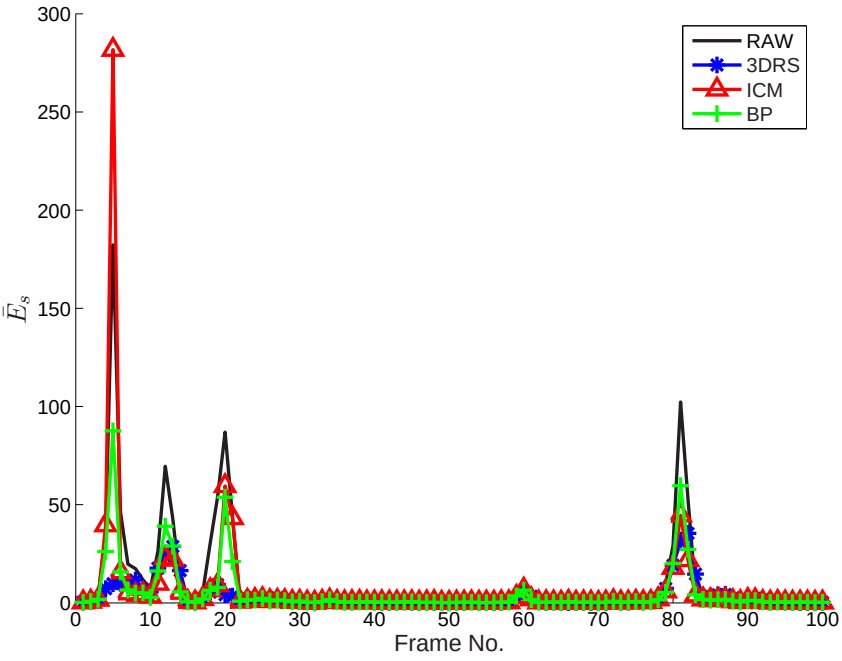
(a) (b)

(c) (d)

Figure B.3: *Foreman sequence (8x8 Blocks). (a) WBME result. (b) 3DRS result. (c) ICM smoothing. (d) BP smoothing.*

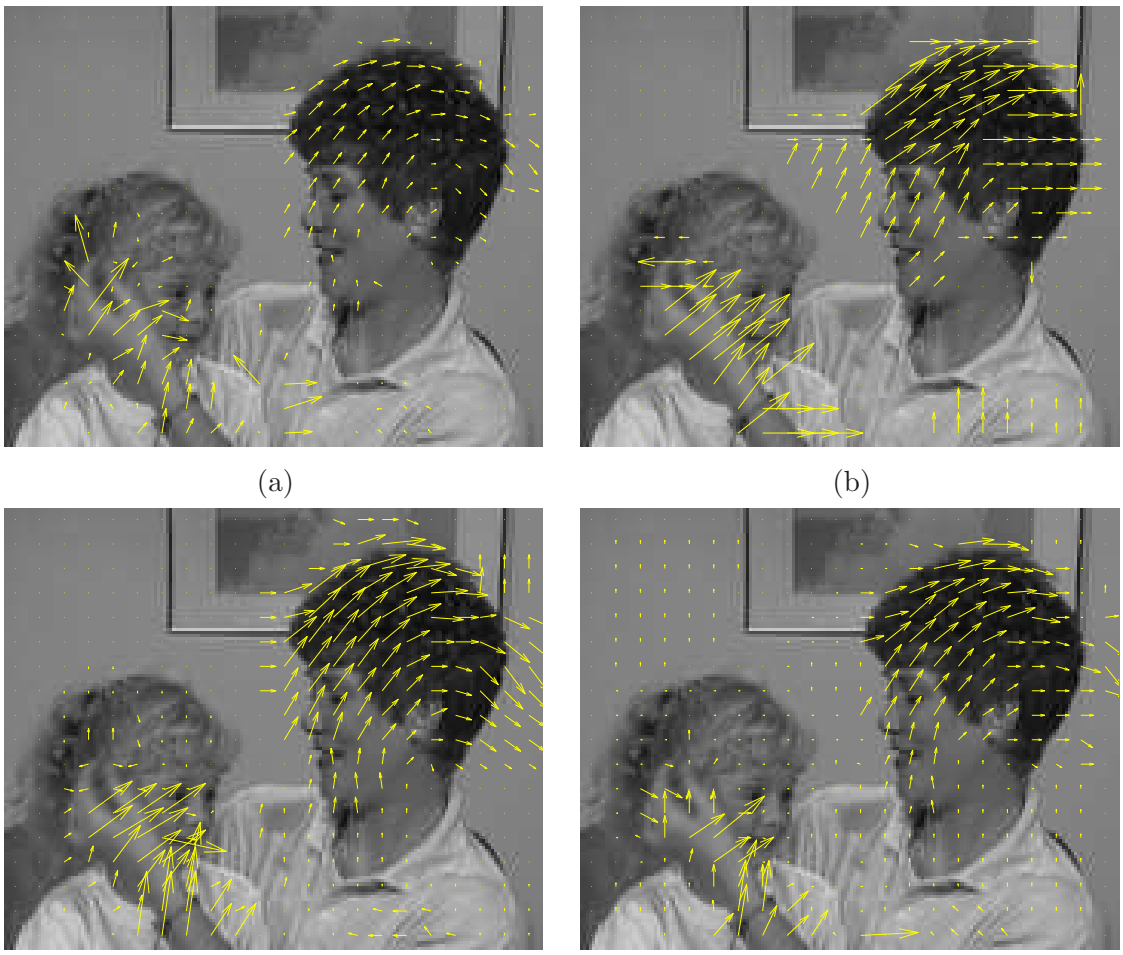


(a)



(b)

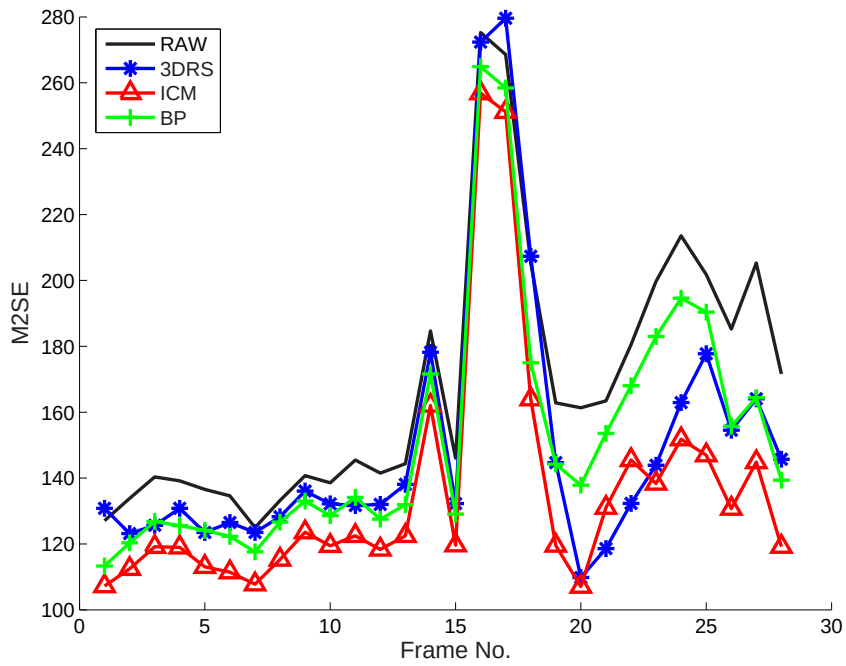
Figure B.4: *Momkid sequence (8x8 Blocks). (a) M2SE for 100 frames of the momkid sequence. The mean value for the zero estimate is 32. (b) Measure of the Smoothness ($\bar{E}_s$) of the motion field.*



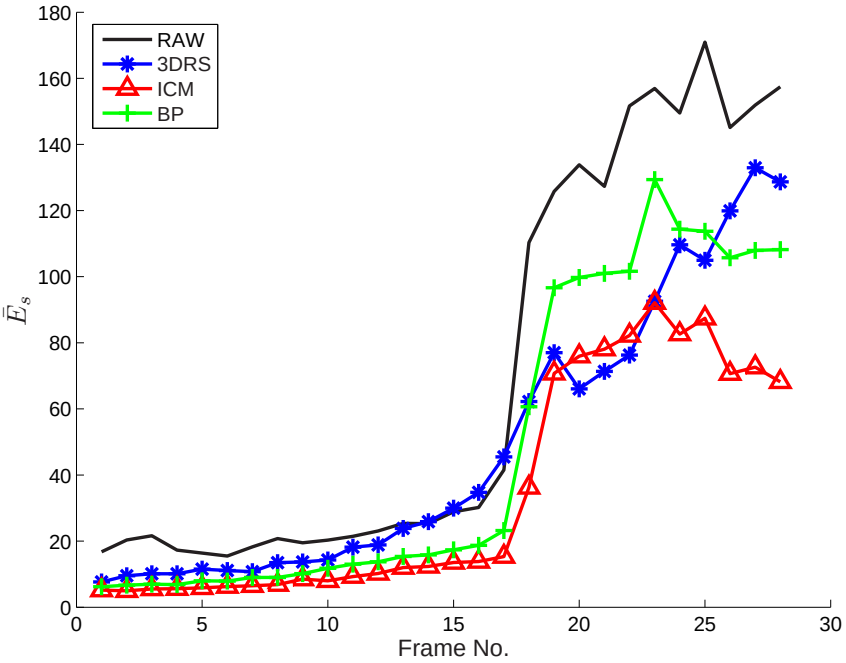
(a) (b)

(c) (d)

Figure B.5: *Momkid sequence (8x8 Blocks). (a) WBME result. (b) 3DRS result. (c) ICM smoothing. (d) BP smoothing.*



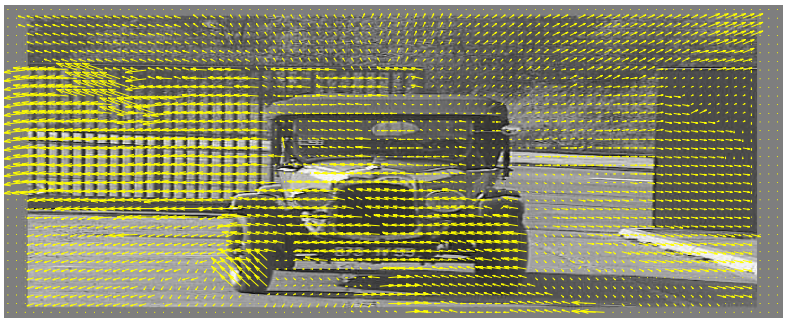
(a)



(b)

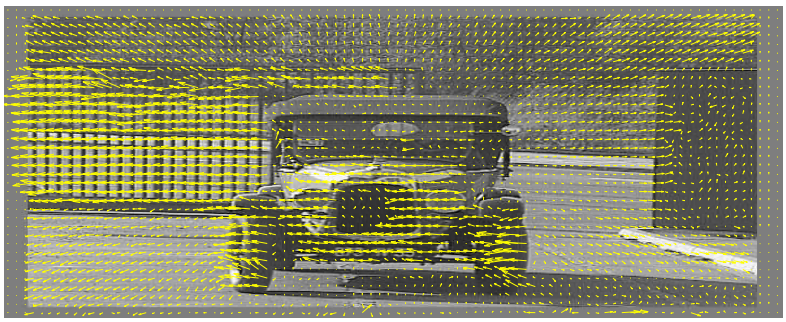
Figure B.6: *Voiture sequence (8x8 Blocks). This was interlaced footage and so for these tests motion estimation was performed only on the odd fields. (a) M2SE for 30 frames of the voiture sequence. The mean value for the zero estimate is 531. (b) Measure of the Smoothness ($\bar{E}_s$) of the motion field.*

(a)



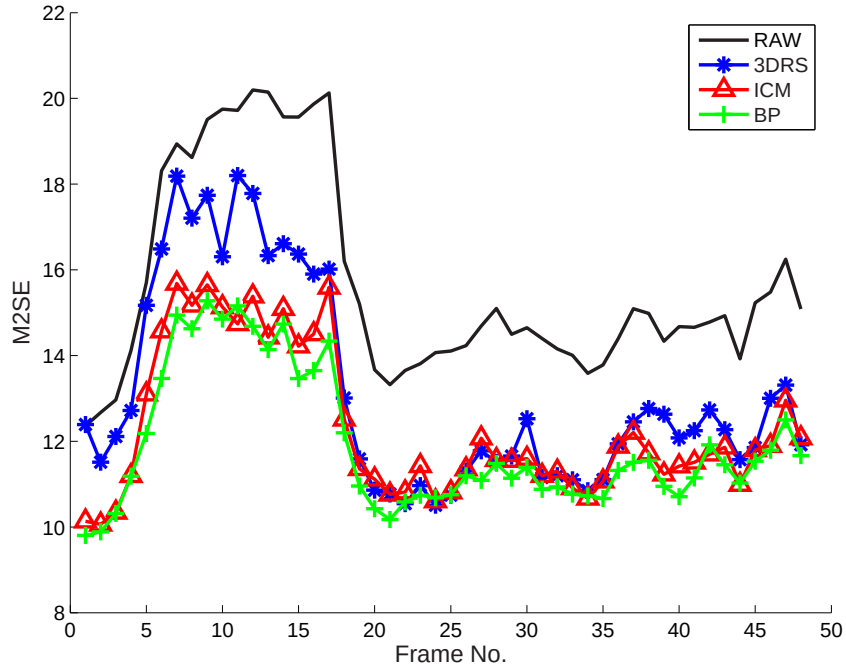
(b)

(c)

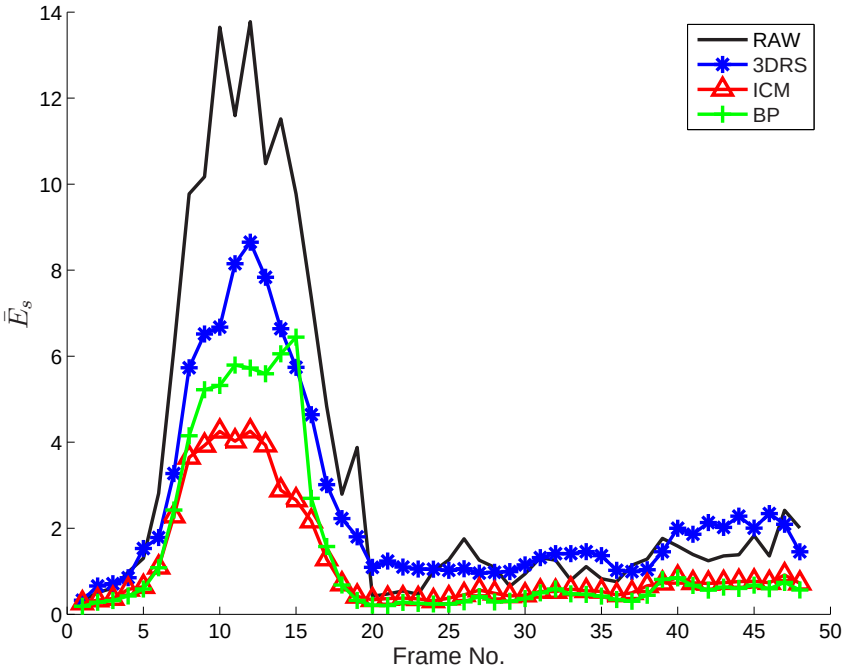


(d)

Figure B.7: *Voiture sequence (8x8 Blocks). This was interlaced footage and so for these tests motion estimation was performed only on the odd fields. (a) WBME result. (b) 3DRS result. (c) ICM smoothing. (d) BP smoothing.*

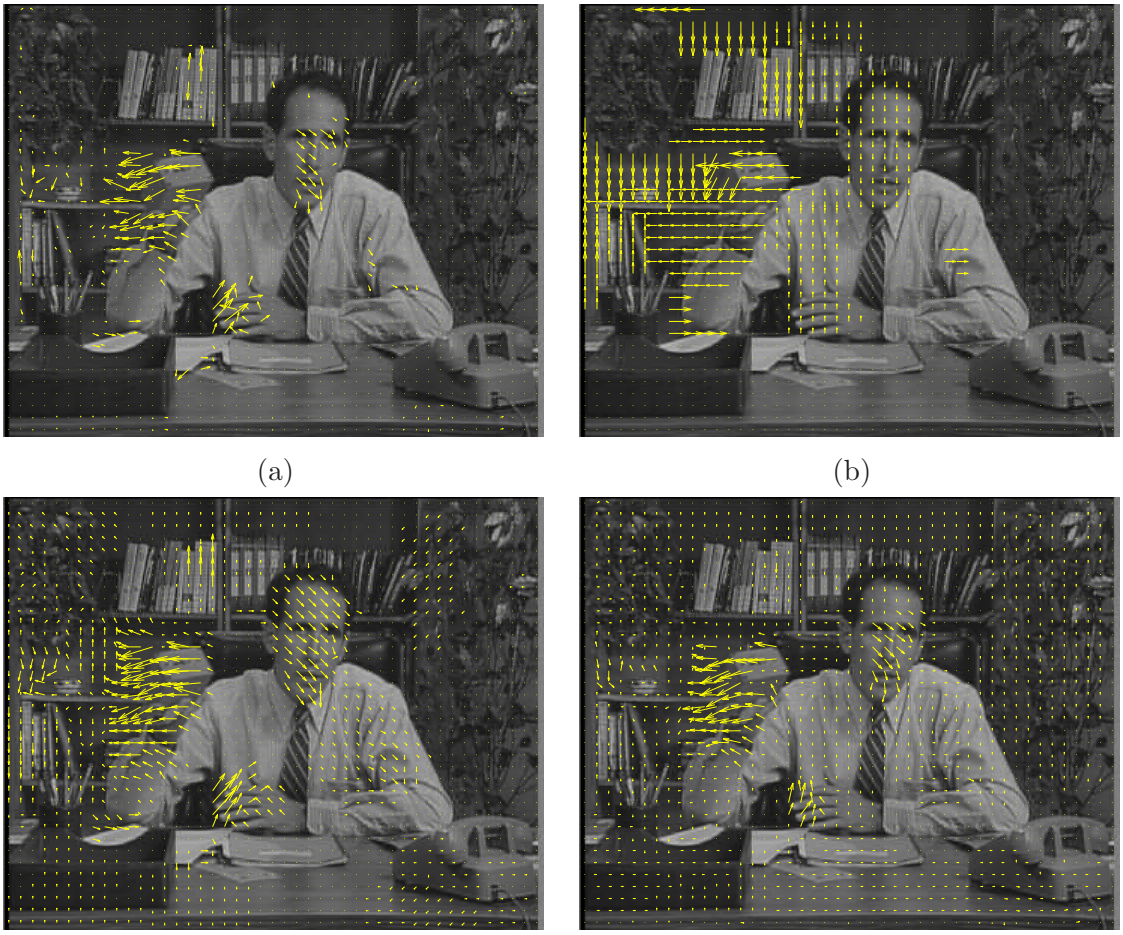


(a)



(b)

Figure B.8: *Salesman sequence (8x8 Blocks). (a) M2SE for 50 frames of the salesman sequence. The mean value for the zero estimate is 22. (b) Measure of the Smoothness ($\bar{E}_s$) of the motion field.*



(a) (b)

(c) (d)

Figure B.9: *Salesman sequence (8x8 Blocks). (a) WBME result. (b) 3DRS result. (c) ICM smoothing. (d) BP smoothing.*

C

GPU Programmable Pipeline

Figures C.1 and C.3 illustrate the operation of the fixed function vertex and fragment stages respectively of the old GPU pipeline. This pipeline was in effect like a state machine. Commands were issued to enable or disable certain elements of each stage. This gave limited control over how vertices could be transformed and lit, and how fragments got coloured and converted into pixels. These were replaced with programmable stages, Figures C.2 and C.4, to allow for much more control of vertex and fragment processing.

Table C.1 lists the instruction set for the ARB fragment program standard. The instruction set for the vertex stage is quite similar. This has since been extended to include support for program flow control such as branching and looping in both the vertex and fragment programs. Documentation on all these standards is available on-line at the SGI extension registry `http://oss.sgi.com/projects/ogl-sample/registry/index.html` [150].

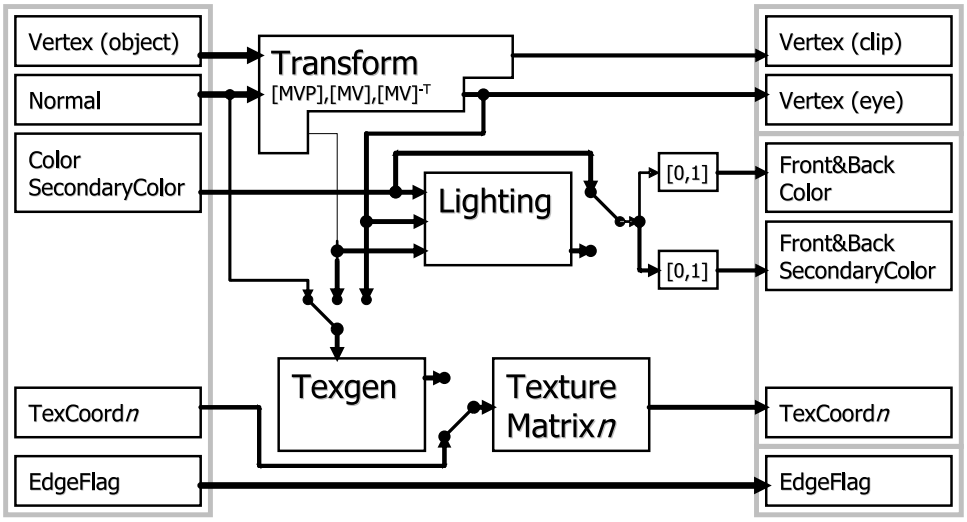


Figure C.1: *Fixed function vertex stage*

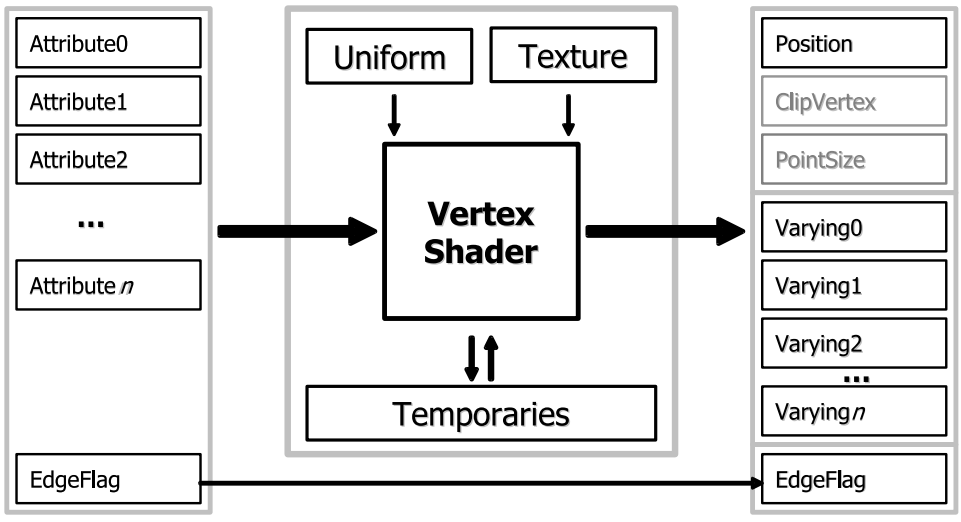


Figure C.2: *Programmable vertex stage*

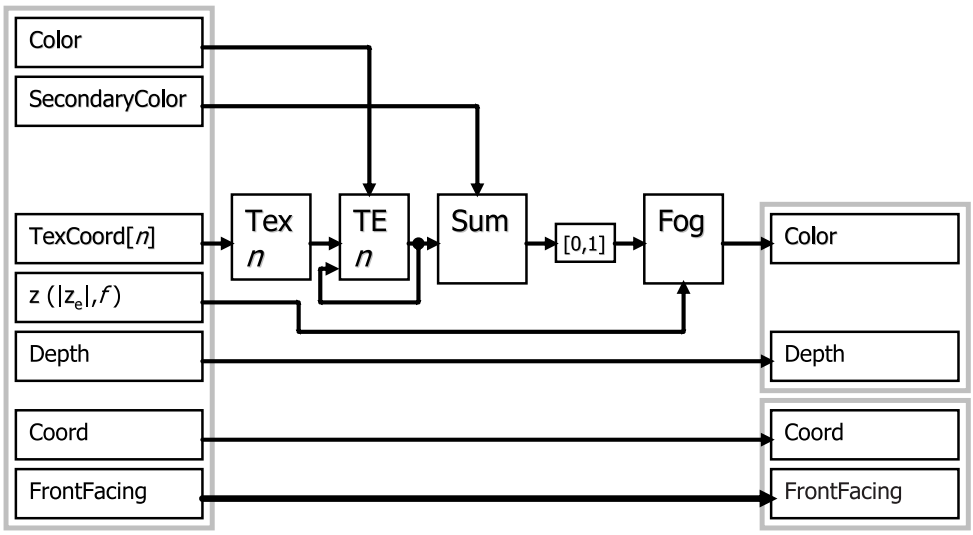


Figure C.3: *Fixed function fragment stage*

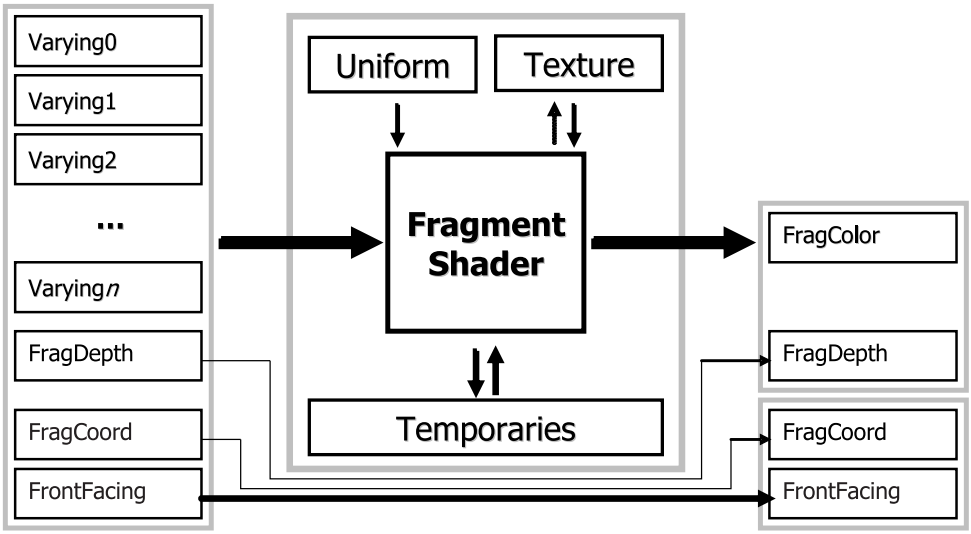


Figure C.4: *Programmable fragment stage*

Instruction	Meaning	Instruction	Meaning
ABS	Absolute Value	MIN	Minimum
ADD	Add	MOV	Move
CMP	Compare	MUL	Multiply
COS	Cosine with reduction to [-pi pi]	POW	Exponentiate
DP3	3-Component Dot Product	RCP	Reciprocal
DP4	4-Component Dot Product	RSQ	Reciprocal Square Root
DPH	Homogeneous Dot Product	SCS	Sine/Cosine without reduction
DST	Distance Vector	SGE	Set on Greater Than or Equal
EX2	Exponential	SIN	Sine with reduction to [-pi pi]
FLR	Floor	SLT	Set on Less Than
FRC	Fraction	SUB	Subtract
KIL	Kill Fragment	SWZ	Extended Swizzle
LG2	Logarithm Base 2	TEX	Texture Sample
LIT	Compute Light Coefficients	TXB	Texture Sample with bias
LRP	Linear Interpolation	TXP	Texture Sample projection
MAD	Multiply and Add	XPD	Cross Product
MAX	Maximum		

Table C.1: *ARB Fragment Program Instruction Set (Version 1.0)*

Bibliography

[1] 2d3. SteadyMove: Automatic stabilization software. `http://www.2d3.com/`.

[2] AGP. Accelerated graphics port technology specification. `http://developer.intel.com/technology/agp/`, 1998.

[3] Apple Computer Inc. Core Image. `http://www.apple.com/macosx/features/coreimage.html`, 2005.

[4] S. Arulampalam, S. Maskell, N. Gordon, and T. Clapp. A tutorial on particle filters for on-line non-linear/non-gaussian bayesian tracking. *IEEE Transactions on Signal Processing*, 50(2):174–188, February 2002.

[5] ATI Technologies Inc. ATI software developer site. `http://www.ati.com/developer/`.

[6] A. Beric, G. de Haan, R. Sethuraman, and J. van Meerbergen. A technique for reducing complexity of recursive motion estimation algorithm. In *Proceedings of SIPS, IEEE Workshop on Signal Processing Systems, Seoul, South Korea*, pages 195–200, August 2003.

[7] C. Berrou, A. Glavieux, and P. Thitimajshima. Near shannon limit error-correcting coding and decoding: Turbo-codes. In *Proceedings IEEE International Conference on Communications*, pages 1064–1070, Geneva, Switzerland, 1993.

[8] M. Bertalmio, L. Vese, G. Sapiro, and S. Osher. Simultaneous structure and texture image inpainting. *IEEE Transactions on Image Processing*, 12(8):882–889, August 2003.

[9] J. Besag. On the statistical analysis of dirty pictures. *Journal of the Royal Statistical Society B*, 48:259–302, 1986.

[10] J. Biemond, L. Looijenga, D. E. Boekee, and R. Plompen. A pel–recursive Wiener based displacement estimation algorithm. *Signal Processing*, 13:399–412, 1987.

[11] M. Bierling. Displacement estimation by hierarchical block-matching. In *Proceedings of Visual Communications and Image Processing*, volume SPIE vol. 1001, pages 942–951, 1988.

[12] G. L. Bilbro, W. E. Snyder, S. J. Garnier, and J. W. Gault. Mean field annealing: A formalism for constructing GNC-like algorithms. *IEEE Transactions on Neural Networks*, 3:131–138, January 1992.

[13] BionicFX. Audio Video Exchange (AVEX): GPU audio processing software. `http://www.bionicfx.com`, 2005.

[14] M. J. Black and P. Anandan. A framework for the robust estimation of optical flow. In *Proceedings of Fourth International Conference on Computer Vision*, pages 231–236, May 1993.

[15] A. Bleiweiss and A. Preetham. Ashli–advanced shading language interface. In *ACM SIGGRAPH Course Notes*, 2003.

[16] J. Bolz, I. Farmer, E. Grinspun, and P. Schroder. Sparse matrix solvers on the gpu: Conjugate gradients and multigrid. *ACM Transactions on Graphics*, 22(3):917–924, July 2003.

[17] L. Böröczky, J. N. Driessen, and J. Biemond. Adaptive algorithms for pel–recursive displacement estimation. In *Proceedings SPIE VCIP*, pages 1210–1221, 1990.

[18] Y. Boykov, O. Veksler, and R. Zabih. Fast approximate energy minimization via graph cuts. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 23(11):1222–1239, November 2001.

[19] R. N. Bracewell, K.-Y. Chang, A. K. Jha, and Y.-H. Wang. Affine theorem for two-dimensional fourier transform. *Electronics Letters*, 29(3), February 1993.

[20] R. Braspenning and G. de Haan. True-motion estimation using feature correspondences. In *SPIE Conference on Visual Communications and Image Processing*, volume 5308, pages 396–407, January 2004.

[21] M. Brunig and W. Niehsen. Fast full-search block matching. *IEEE Transactions on Circuits and Systems for Video Technology*, 11(2):241–247, February 2001.

[22] I. Buck. GPGPU: General-purpose computation on graphics hardware–GPU computation strategies & tricks. In *ACM SIGGRAPH Course Notes*, August 2004.

[23] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for gpus: Stream computing on graphics hardware. *ACM Transactions on Graphics*, 23(3):777–786, August 2004.

[24] P. Burt and E. Adelson. The Laplacian pyramid as a compact image code. *IEEE Transactions on Communications*, 31:532–540, April 1983.

[25] C. Cafforio and F. Rocca. Methods for measuring small displacements of television images. *IEEE transactions on Information Theory*, 22:573–579, 1976.

[26] P. Campisi and G. Scarano. A multiresolution approach for texture synthesis using the circular harmonic functions. *IEEE Transactions on Image Processing*, 11(1):37–51, January 2002.

[27] P. Chou and C. Brown. The theory and practice of bayesian image labelling. *International Journal on Computer Vision*, 4(3):185–210, 1990.

[28] M. F. Cohen, J. Shade, S. Hiller, and O. Deussen. Wang tiles for image and texture generation. *ACM Transaction on Graphics*, 22(3):287–294, July 2003.

[29] B. Collis and A. Kokaram. Filling in the gaps. *IEE Electronics Systems and Software*, pages 22–28, August/September 2004.

[30] B. Collis, S. Robinson, and P. White. Wire removal. In *Proceedings of 1st IEE European Conference on Visual Media Production*, pages 133–138, March 2004.

[31] R. Coudray and B. Besserer. Global motion estimation for mpeg-encoded streams. In *Proceedings of IEEE International Conference on Image Processing*, October 2004.

[32] A. Crawford, H. Denman, F. Kelly, F. Pitie, and A. Kokaram. Gradient based dominant motion estimation with integral projections for real time video stabilisation. In *Proceedings of IEEE International Conference on Image Processing, ICIP'04*, October 2004.

[33] J. Davis. Mosaics of scenes with moving objects. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 1998.

[34] R. Dayhot and A. Kokaram. Comparison of two algorithms for robust M-estimation of global motion parameters. In *Proceedings of Irish Machine Vision and Image Processing*, September 2004.

[35] G. de Haan. Progress in motion estimation for video format conversion. *IEEE Transactions on Consumer Electronics*, 45(3):449–459, August 2000.

[36] G. de Haan, P. Biezen, H. Huijgen, and O. A. Ojo. True-motion estimation with 3-d recursive search block matching. *IEEE Transactions on Circuits and Systems for Video Technology*, 3(5):368–379, October 1993.

[37] G. de Haan and P. W. A. C. Biezen. Sub-pixel motion estimation with 3-d recursive search block-matching. *Signal Processing: Image Communciation*, 6:229–239, 1994.

[38] G. de Haan and P. W. A. C. Biezen. An efficient true-motion estimator using candidate vectors from a parametric motion model. *IEEE Transactions on Circuits and Systems for Video Technology*, 8(1):85–91, February 1998.

[39] G. de Haan and J. Kettenis. System-on-Silicon for high quality display format conversion and video enhancement. In *Proceedings of ISCE'02*, pages E1–E6, September 2002.

[40] P. Delacourt, A. Kokaram, and R. Dayhot. Comparison of global motion estimators. In *Proceedings of Irish Signals and Systems Conference*, June 2002.

[41] A. Doucet. On sequential simulation-based methods for bayesian filtering. Technical Report CUED/F-INFENG/TR. 310, Cambridge University Department of Engineering, 1998.

[42] A. Doucet, N. de Freitas, and N. Gordon, editors. *Sequential Monte Carlo Methods in Practice*. Springer, ISBN: 0387951466, 2001.

[43] J. Driessen, J. Biemond, and D. Boekee. A pel–recursive segmentation and estimation algorithm for motion compensated image sequence coding. In *IEEE ICASSP*, pages 1901–1904, 1989.

[44] J. Driessen, L. Boroczky, and J. Biemond. Pel–recursive motion field estimation from image sequences. *Visual Communication and Image Representation*, 2:259–280, 1991.

[45] F. Dufaux and J. Konrad. Efficient, robust, and fast global motion estimation for video coding. *IEEE Transaction on Image Processing*, 9(3):497–501, March 2000.

[46] F. Dufaux and F. Moscheni. Motion estimation techniques for digital TV: a review and a new contribution. *Proceedings of the IEEE*, 83(6):858–876, June 1995.

[47] A. A. Efros and T. K. Leung. Texture synthesis by non-parametric sampling. In *Proceedings of IEEE International Conference on Computer Vision*, pages 1033–1038, September 1999.

[48] S. Efstratiadis and A. Katsagellos. A model based, pel–recursive motion estimation algorithm. In *Proceedings IEEE ICASSP*, pages 1973–1976, 1990.

[49] M. Ekman, F. Warg, and J. Nilsson. An in-depth look at computer performance growth. *ACM SIGARCH Computer Architecture News*, 33(1):144–147, March 2005.

[50] W. Enkelmann. Investigations of multigrid algorithms for the estimation of optical flow fields in image sequences. *Computer Vision Graphics and Image Processing*, 43:150–177, 1988.

[51] Y. M. Erkam, M. I. Sezan, and A. T. Erdem. A hierarchical phase-correlation method for motion estimation. In *Proceedings Conference on Information Science and Systems*, pages 419–424, 1991.

[52] J. Eyles, S. Molnar, J. Poulton, T. Greer, A. Lastra, N. England, and L. Westover. Pixelflow: The realization. In *1997 SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 57–68, August 1997.

[53] P. F. Felzenszwalb and D. P. Huttenlocher. Efficient belief propagation for early vision. In *Proceedings of IEEE Computer Vision and Pattern Recognition (CVPR)*, volume 1, pages 261–268, 2004.

[54] C. Fermuller, Y. Aloimonos, and H. Malm. Bias in visual motion processes: A theory predicting illusions. In *Workshop on Statistical Methods in Video Processing in conjunction with ECCV 2002*, pages 79–84, June 2002.

[55] A. W. Fitzgibbon. Stochastic rigidity: Image registration for nowhere-static scenes. In *Proceedings International Conference on Computer Vision*, volume 1, pages 662–670, 2001.

[56] For–A. IVS–300A Video Stabiliser. `http://www.for-a.com/sub_html/p_product/ivs300a.html`.

[57] G. Forney. The viterbi algorithm. *Proceedings of the IEEE*, 61(3):268–278, March 1973.

[58] A. Fournier and D. Fussell. On the power of the frame buffer. *ACM Transactions on Graphics*, 7(2):103–128, 1988.

[59] W. T. Freeman, E. C. Pasztor, and O. T. Carmichael. Learning low-level vision. *International Journal of Computer Vision*, 40(1):25–47, 2000.

[60] C. Gallagher, F. Kelly, and A. Kokaram. Fast scale invariant texture synthesis and gpu accelerated block searching. In *IEE Conference on Visual Media Production*, November 2005.

[61] C. Gallagher and A. Kokaram. Nonparametric wavelet based texture synthesis. In *Proceedings of IEEE International Conference on Image Processing*, volume II, pages 462–465, September 2005.

[62] S. Geman and D. Geman. Stochastic relaxation, gibbs distributions, and the bayesian restoration of images. *IEEE Transactions in Pattern Analysis and Machine Intelligence*, 6:721–741, 1984.

[63] I. Geys, T. P. Konichkx, and L. V. Gool. Fast interpolated cameras by combining a gpu based plane sweep with a max-flow regularisation algorithm. In *Proceedings of the 2nd International Symposium on 3D Data Processing, Visualization, and Transmission (3DPVT'04)*, 2004.

[64] M. Ghanbari. The cross-search algorithm for motion estimation. *IEEE Transactions on Communications*, 38:950–953, July 1990.

[65] N. Goodnight, C. Woolley, G. Lewin, D. Luebke, and G. Humphreys. A multigrid solver for boundary value problems using programmable graphics hardware. In *Graphics Hardware 2003*, pages 102–111, July 2003.

[66] GPGPU. General Purpose Computation Using Graphics Hardware. `http://www.gpgpu.org`.

[67] H.261. *Video Codec for audiovisual services at p x 64 kbits/s.* ITU–T Recommendation H.261, 1990.

[68] H.263. *Video coding for low bit rate communication.* ITU–T Recommendation H.263, 1995.

[69] H.264. *Advanced video coding for generic audiovisual services.* ITU–T Recommendation H.264, 2003.

[70] M. Harris. GPGPU: General-purpose computation on GPUs. In *Game Developers Conference*, 2005.

[71] M. Harris, W. B. III, T. Scheuermann, and A. Lastra. Simulation of cloud dynamics on graphics hardware. In *Graphics Hardware*, pages 92–101, July 2003.

[72] R. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision.* Cambridge University Press, ISBN: 0521540518, second edition, 2004.

[73] L. Hill and T. Vlachos. On the estimation of global motion using phase correlation for broadcast applications. In *Proceedings IEE International Conference on Image Processing and its Applications*, volume 2, pages 721–725, July 1999.

[74] P. W. Holland and R. E. Welsch. Robust regression using iteratively reweighted least squares. *Communications on Statistics and Theoretical Methods*, A6:813–828, 1977.

[75] J. Y. Hong and M. D. Wang. High speed processing of biomedical images using programmable gpu. In *ICIP*, pages 2455–2458, October 2004.

[76] B. Horn and B. Schunck. Determining optical flow. *Artificial Intelligence*, 17:185–203, 1981.

[77] P. J. Huber. *Robust Statistics.* John Wiley and Sons, ISBN: 0471418056, 1981.

[78] K. E. H. III, T. Culver, J. Keyser, M. Lin, and D. Manocha. Fast computation of generalized voronoi diagrams using graphics hardware. *ACM Transactions on Graphics*, 18:277–286, August 1999.

[79] K. E. H. III, A. Zaferakis, M. Lin, and D. Manocha. Fast and simple 2d geometric proximity queries using graphics hardware. In *Proceedings of ACM Symposium on Interactive 3D Graphics*, 2001.

[80] K. E. H. III, A. Zaferakis, M. Lin, and D. Manocha. Fast 3d geometric proximity queries between rigid and deformable models using graphics hardware acceleration. In *UNC-CS Technical Report*, 2002.

[81] Intel Corporation. Intel integrated performance primitives version 4.0 (Software libraries). `http://www.intel.com/cd/software/products/asmo-na/eng/perflib/ipp/index%.htm`.

[82] Intel Corporation. PCI Express specification. `http://developer.intel.com/technology/pciexpress/devnet/`, 2004.

[83] M. Isard and A. Blake. CONDENSATION – conditional density propagation for visual tracking. *International Journal of Computer Vision*, 29(1):5–28, 1998.

[84] A. K. Jain and J. R. Jain. Radar image modelling and processing for real-time RF simulation. Technical report, Department of Electronic Engineering, State University of New York at Buffalo, 1978.

[85] J. Jain and A. Jain. Displacement measurement and its application in interframe image coding. *IEEE Transactions on Communications*, 29:1799–1981, 1981.

[86] T. Jansen, B. V. Rymon-Lipinski, N. Hanssen, and E. Keeve. Fourier volume rendering on the gpu using a split-stream fft. In *Proceedings of Vision, Modeling, and Visualization*, pages 395–403, November 2004.

[87] F. Jargstorff. GPU image processing. In *Tutorial Course 5, EuroGraphics04*, 2004.

[88] M. Jedrzejewski. Computation of room acoustics on programmable video hardware. Master's thesis, Polish–Japanese Institute of Information Technology, Warsaw, Poland, 2004.

[89] H. Jozawa, K. Kamikura, A. Sagata, H. Kotera, and H. Watanabe. Two stage motion compensation using adaptive global MC and local affine MC. *IEEE Transactions on Circuits, Systems, and Video Technology*, 7:75–85, February 1997.

[90] R. E. Kalman. A new approach to linear filtering and prediction problems. *Transactions of the ASME–Journal of Basic Engineering*, 82(Series D):35–45, 1960.

[91] J. Kearney, W. Thompson, and D. L. Boley. Optical flow estimation: An error analysis of gradient based methods with local optimisation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 229–243, March 1987.

[92] G. Kedem and Y. Ishihara. Brute force attack on UNIX passwords with SIMD computer. In *Proceedings of the eight USENIX Security Symposium*, pages 93–98, August 1999.

[93] F. Kelly and A. Kokaram. General purpose graphics hardware for accelerating motion estimation. In *Irish Machine Vision and Image Processing (IMVIP) Conference*, September 2003.

[94] F. Kelly and A. Kokaram. Fast image interpolation for motion estimation using graphics hardware. In *SPIE Conference on Real Time Imaging VIII*, volume 5297, January 2004.

[95] F. Kelly and A. Kokaram. Graphics hardware for gradient based motion estimation. In *SPIE Conference on Embedded Processors for Multimedia and Communications*, volume 5309, January 2004.

[96] F. Kelly and A. Kokaram. Online global motion estimation. In *IEE Conference on Visual Media Production*, November 2005.

[97] F. Kelly and A. Kokaram. Candidates in motion. In *Submitted to* 9^{th} *European Conference on Computer Vision*, May 2006.

[98] J. Kessenich, D. Baldwin, and R. Rost. The OpenGL shading language (Online documentation). `http://www.opengl.org/documentation/oglsl.html`, 2004.

[99] J. Kim and R. H. Park. A fast feature-based block matching algorithm using integral projections. *IEEE Journal on Selected areas in Communications*, 10(5), June 1992.

[100] T. Koga, K. Iinmua, A. Hirano, Y. Iijima, and T. Ishiguro. Motion-compensated interframe coding for video conferencing. In *Proceedings NTC'81 (IEEE)*, pages G.5.3.1–G.5.3.4, 1981.

[101] A. Kokaram. *Motion Picture Restoration.* Springer-Verlag, ISBN: 3540760407, May 1998.

[102] A. Kokaram. Parametric texture synthesis for filling holes in pictures. In *IEEE International Conference on Image Processing*, pages 325–328, Rochester, New York, USA, September 2002.

[103] A. Kokaram, B. Collis, and S. Robinson. A bayesian framework for recursive object removal in movie post-production. In *IEEE International Conference on Image Processing*, Barcelona, September 2003.

[104] A. Kokaram, R. Dayhot, F. Pitie, and H. Denman. Simultaneous luminance and position stabilisation for film and video. In *In Proceedings SPIE Visual Communications and Image Processing*, January 2003.

[105] A. Kokaram and P. Delacourt. A new global motion estimation algorithm and its application to retrieval in sports events. In *IEEE International Workshop on Multimedia Signal Processing, MMSP'01*, October 2001.

[106] A. Kokaram, F. Pitie, R. Dayhot, N. Rea, and S. Yeterian. Content controlled image representation for sports streaming. In *Proceedings of the IEEE workshop on Content Based Multimedia Indexing*, June 2005.

[107] A. C. Kokaram. *Motion Picture Restoration.* PhD thesis, Cambridge University, England, May 1993.

[108] J. Konrad and E. Dubois. Bayesian estimation of motion vector fields. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 14(9), September 1992.

[109] J. Kruger and R. Westermann. Linear algebra operators for gpu implementation of numerical algorithms. *ACM Transactions on Graphics*, 22(3):908–916, July 2003.

[110] C. D. Kuglin and D. C. Hines. The phase correlation image alignment method. In *Proceedings of IEEE International Conference on Cybernetics and Society*, pages 163–165, 1975.

[111] S. Kumar, M. Biswas, and T. Q. Nquyen. Global motion estimation in frequency and spatial domain. In *Proceedings of IEEE International Conference Acoustics, Speech, and Signal Processing (ICASSP '04)*, volume 3, pages 333–336, May 2004.

[112] E. S. Larsen and D. McAllister. Fast matrix multiplies using graphics hardware. *Supercomputing 2001*, November 2001.

[113] I. H. Lee and R. H. Park. A fast block matching algorithm using integral projections. In *Proceedings of IEEE TENCON '87 Conference*, pages 18.3.1–18.3.5, August 1987.

[114] J. H. Lee and J. B. Ra. Block motion estimation based on selective integral projections. In *IEEE ICIP*, volume I, pages 689–693, 2002.

[115] A. E. Lefohn, J. M. Kniss, C. D. Hansen, and R. T. Whitaker. A streaming narrow-band algorithm: Interactive computation and visualization of level-set surfaces. *IEEE Transactions on Visualization and Computer Graphics*, 10(4):422–433, July/August 2004.

[116] A. E. Lefohn and R. T. Whitaker. A gpu-based, three-dimensional level set solver with curvature flow. Technical Report UUCS-02-017, University of Utah, 2002.

[117] A. Levinthal and T. Porter. Chap – a SIMD graphics processor. In *Computer Graphics (Proceedings of SIGGRAPH 84)*, volume 18, pages 77–82, July 1984.

[118] W. Li and E. Salari. Successive elimination algorithm for motion estimation. *IEEE Transactions on Image Processing*, 4(1):105–107, January 1995.

[119] M. C. Lin and D. Manocha. Interactive geometric computations using graphics hardware. In *ACM SIGGRAPH Tutorial Course 31*, 2002.

[120] E. Lindholm, M. Kilgard, and H. Moreton. A user-programmable vertex engine. In *Proceedings of ACM SIGGRAPH*, pages 149–158, August 2001.

[121] L. Lucchese. Estimating affine transformation in the frequency domain. In *Proceedings IEEE International Conference on Image Proceedings (ICIP)*, volume 2, pages 7–10, October 2001.

[122] J. MacCormick and A. Blake. A probabilistic exclusion principle for tracking multiple objects. *International Journal of Computer Vision*, 39(1):57–71, 1999.

[123] M. Macedonia. The gpu enters computing's mainstream. *IEEE Computer*, 36(10):106–108, October 2003.

[124] D. J. C. MacKay. *Introduction to Monte Carlo Methods*, pages 175–204. MIT Press, ISBN: 0262600323, 1999.

[125] J. F. A. Magarey. *Motion estimation using complex wavelets.* PhD thesis, Cambridge University, England, 1997.

[126] S. Mallat. Multifrequency channel decompositions of images and wavelet models. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 37(12):2091–2110, December 1989.

[127] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard. Cg: A system for programming graphics hardware in a c-like language. *ACM Transactions on Graphics*, 22(3):896–907, July 2003.

[128] D. M. Martinez. *Model-based motion estimation and its application to restoration and interpolation of motion pictures.* PhD thesis, Massachusetts Institute of Technology, 1986.

[129] M. McCool, S. D. Toit, T. Popa, B. Chan, and K. Moule. Shader algebra. *ACM Transactions on Graphics*, 23(3):787–795, August 2004.

[130] R. J. McEliece, D. J. C. MacKay, and J. F. Cheng. Turbo decoding as an instance of Pearl's "Belief Propagation" algorithm. *IEEE Journal on Selected Areas in Communications*, 16(2):140–152, February 1998.

[131] Microsoft Corporation. DirectX 9.0 software development kit. `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/direct%x9_c/directx/directx9cpp.asp`, 2005.

[132] Microsoft Corporation. HLSL: High-level shader language (Online documentation). `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/direct%x9_c/directx/graphics/programmingguide/hlslshaders/programmablehlslshaders.asp%`, 2005.

[133] P. Milanfar. A model of the effect of image motion in the radon transform domain. *IEEE Transactions on Image Processing*, 8(9):1276–1281, 1999.

[134] A. Mittal and D. Huttenlocher. Scene modelling for wide area surveillance and image synthesis. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, volume 2, pages 160–167, June 2000.

[135] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[136] K. Moreland and E. Angel. The FFT on a GPU. In *Graphics Hardware 2003*, July 2003.

[137] C. Morimoto and R. Chellappa. Evaluation of image stabilization algorithms. In *Proceedings of IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 5, pages 2789–2792, May 1998.

[138] MPEG-1. *Coding of moving pictures and associated audio for digital storage media up to about 1.5 Mbit/s*. ISO/IEC JTC1/SC29/WG11 11172-2:Video, 1993.

[139] MPEG-2. *Generic coding of moving pictures and associated audio information*. ISO/IEC JTC1/SC29/WG11 13818-2:Video, 1994.

[140] MPEG-4. *Coding of moving pictures and audio*. ISO/IEC JTC1/SC29/WG11 14496-2:Visual, 1998.

[141] H. Nagel. Image sequences – ten (octal) years – from phenomenology towards a theoretical foundation. In *IEEE ICASSP*, pages 1174–1185, 1986.

[142] H. Nagel. On a constraint equation for the estimation of displacement rates in image sequences. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11:13–30, January 1989.

[143] H. Nagel and W. Enkelmann. An investigation of smoothness constraints for the estimation of displacement vector field from image sequences. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 8:565–592, September 1986.

[144] NeHe Productions. Online OpenGL programming tutorials. `http://nehe.gamedev.net`.

[145] A. Netravali and J. Robbins. Motion-compensated television coding: Part 1. *The Bell System Technical Journal*, pages 59:1735–1745, November 1980.

[146] NVIDIA Corporation. Nvidia software developer site. `http://developer.nvidia.com`.

[147] J.-M. Odobez and P. Bouthemy. Robust multiresolution estimation of parametric motion models. *Journal of Visual Communication and Image Representation*, 6(1):348–365, 1995.

[148] K. Okuma, A. Taleghani, N. de Freitas, J. J. Little, and D. G. Lowe. A boosted particle filter: Multitarget detection and tracking. In *European Conference on Computer Vision (ECCV)*, 2004.

[149] OpenGL. Official OpenGL web page. `http://www.opengl.org`.

[150] OpenGL Extensions. SGI Online OpenGL extensions registry. `http://oss.sgi.com/projects/ogl-sample/registry/index.html`.

[151] OpenVIDIA. Open source parallel GPU computer vision project. `http://openvidia.sourceforge.net/`.

[152] A. V. Oppenheim, R. W. Schafer, and J. R. Buck. *Discrete-Time Signal Processing.* Signal Processing Series. Prentice Hall, ISBN: 0137549202, 2^{nd}edition, 1999.

[153] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. In *Eurographics 2005, State of the Art Reports*, pages 21–51, August 2005.

[154] PCI SIG. PCI special interest group website. `http://www.pcisig.com`.

[155] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference.* Morgan Kaufmann, ISBN: 1558604790, 1988.

[156] Philips Corporation. *Philips SAA4991WP MELZONIC image processing chip*, 1997.

[157] M. Pilu. Video stabilization as a variational problem and numerical solution with the viterbi method. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, volume 1, pages I–625–I–630, June 2004.

[158] F. Pitie, R. Dahyot, F. Kelly, and A. Kokaram. A new robust technique for stabilizing brightness fluctuations in image sequences. In 2^{nd} *Workshop on Statistical Methods in Video Processing in conjunction with ECCV*, May 2004.

[159] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C, The Art of Scientific Computing.* Cambridge University Press, ISBN: 0521431085, 2nd edition, 1992.

[160] T. J. Purcell, I. Buck, W. R. Mark, and P. Hanrahan. Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics*, 21(3):703–712, July 2002.

[161] L. R. Rabiner and B. H. Juang. An introduction to Hidden Markov Models. *IEEE Acoustics, Speech & Signal Processing Magazine*, 3(1):4–16, January 1986.

[162] B. S. Reddy and B. N. Chatterji. An fft-based technique for translation, rotation, and scale-invariant image registration. *IEEE Transactions on Image Processing*, 5(8), August 1996.

[163] J. Rhoades, G. Turk, A. Bell, A. State, U. Neumann, and A. Varshney. Real-time procedural textures. In *In 1992 Symposium on Interactive 3D Graphics*, volume 25, pages 95–100, March 1992.

[164] J. Riveros and K. Jabbour. Review of motion analysis techniques. *IEE Proceedings*, 136:397–404, December 1989.

[165] J. J. O. Ruanaidh and W. J. Fitzgerald. *Numerical Bayesian Methods Applied to Signal Processing*. Springer Series in Statistics and Computing. Springer–Verlag, ISBN: 0387946292, 1996.

[166] M. Rumpf and R. Strzodka. Level set segmentation in graphics hardware. In *Proceedings of the IEEE International Conference on Image Processing*, volume 3, pages 1103–1106, 2001.

[167] G. Shen, L. Zhu, S. Li, H.-Y. Shum, and Y.-Q. Zhang. Accelerating video decoding using gpu. In *IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP '03)*, volume 4, pages IV– 772–775, April 2003.

[168] Snell & Wilcox. Archangel Ph.C Image Restoration System. `http://www.snellwilcox.com/`.

[169] Snell & Wilcox. Steadfast Ph.C Video Stabilization System. `http://www.snellwilcox.com/`.

[170] Sony Corporation. *Sony CXD19222Q Real Time MPEG-2 Video Encoder Chip*, 1997.

[171] C. Stiller. Motion-estimation for coding of moving video at 8kbit/sec with gibbs modelled vectorfield smoothing. In *SPIE Visual Communications and Image Processing*, volume 1360, pages 468–576, 1990.

[172] C. Stiller. Object-based motion computation. In *Proceedings of IEEE International Conference on Image Processing*, pages 913–916, 1996.

[173] C. Stiller. Object-based estimation of dense motion fields. *IEEE Transactions on Image Processing*, 6(2):234–250, February 1997.

[174] C. Stiller and J. Konrad. Estimating motion in image sequences. *IEEE Signal Processing Magazine*, 16(4):70–91, July 1999.

[175] R. Strzodka. Virtual 16-bit precise operations on RGBA8 textures. In *Proceedings of Vision, Modeling, and Visualization*, pages 171–178, 2002.

[176] R. Strzodka, M. Droske, and M. Rumpf. Image registration by regularized gradient flow - a streaming implementation in DX9 graphics hardware. *Computing*, 73(4):373–389, 2004.

[177] R. Strzodka and C. S. Garbe. Real-time motion estimation and visualization on graphics cards. In *Proceedings of IEEE Visualization 2004*, volume 545–552, 2004.

[178] R. Strzodka, I. Ihrke, and M. Magnor. A graphics hardware implementation of the generalized hough transform for fast object recognition, scale, and 3d pose detection. In *International Conference on Image Analysis and Processing (ICIAP 2003)*, pages 188–193, 2003.

[179] J. Sun, H. Y. Shum, and N. N. Zheng. Stereo matching using belief propagation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 25(7):787–800, July 2003.

[180] R. Szeliski. Video mosaics for virtual environments. *IEEE Computer Graphics and Applications*, 16(2):22–30, March 1996.

[181] M. A. Tanner. *Tools for Statistical Inference*. Springer–Verlag, ISBN: 0387946888, 1996.

[182] M. F. Tappen and W. T. Freeman. Comparison of graph cuts with belief propagation for stereo, using identical MRF parameters. In *Proceedings of the Ninth IEEE International Conference on Computer Vision*, 2003.

[183] A. M. Tekalp. *Digital Video Processing*. Signal Processing Series. Prentice-Hall, ISBN: 0131900757, 1995.

[184] G. A. Thomas. Television motion measurement for DATV and other applications. Technical Report BBC–RD–1987–11, BBC Research Department, 1987.

[185] G. A. Thomas. TV Picture motion measurement. *UK Patent Specification No. GB2 188510A*, Sept. 1987.

[186] C. J. Thompson, S. Hahn, and M. Oskin. Using modern graphics architectures for general-purpose computing: A framework and analysis. *International Symposium on Microarchitecture (MICRO)*, November 2002.

[187] TimeSlice Films. Post–Production/Visual Effects Company. `http://www.timeslicefilms.com`.

[188] P. H. S. Torr, R. Szeliski, and P. Anandan. An integrated bayesian approach to layer extraction from image sequences. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 23(3):297–303, March 2001.

[189] S. Upstill. *The RenderMan Companion: A Programmers guide to Realistic Computer Graphics*. Addison–Wesley, ISBN: 0201508680, 1990.

[190] S. Venkatasubramanian. The graphics card as a stream computer. In *SIGMODDIMACS Workshop on Management and Processing of Data Streams*, 2003.

[191] A. J. Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory*, 13(2):260–267, April 1967.

[192] D. Walker and K. Rao. Improved pel–recursive motion compensation. *IEEE Transactions on Communications*, 32:1128–1134, October 1984.

[193] J. Wang, T. T. Wong, P. A. Heng, and C. S. Leung. Discrete wavelet transorm on GPU. `http://www.cse.cuhk.edu.hk/~ttwong/software/dwtgpu/dwtgpu.html`, 2005.

[194] J. Y. A. Wang and E. H. Adelson. Representing moving images with layers. *IEEE Transactions on Image Processing*, 5(3):625–638, September 1994.

[195] L.-Y. Wei and M. Levoy. Fast texture synthesis using tree-structured vector quantization. *Proceedings of ACM SIGGRAPH 2000*, pages 479–488, 2000.

[196] Y. Weiss. Interpreting images by propagating bayesian beliefs. In *Advances in Neural Information Processing Systems 2*, volume 9, pages 908–915, 1997.

[197] Y. Weiss and E. H. Adelson. A unified mixture framework for motion segmentation: Incorporating spatial coherence and estimating the number of models. In *Proceedings of the 1996 Conference on Computer Vision and Pattern Recognition (CVPR '96)*, page 321, 1996.

[198] G. Welch and G. Bishop. An introduction to the kalman filter. Technical Report TR 95-041, Department of Computer Science, University of North Carolina, 2004.

[199] S. Witt. Real-time video effects on a PlayStation2. In *ACM SIGGRAPH Sketches*, 2002.

[200] M. Woo, J. Neider, T. Davis, and D. Shreiner. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.2*. Addison-Wesley, 3rd edition, 1999.

[201] R. Yang and M. Pollefeys. Multi-resolution real-time stereo on commodity graphics hardware. In *CVPR'03*, 2003.

[202] R. Yang and G. Welch. Fast image segmentation and smoothing using commodity graphics hardware. *Journal of Graphics Tools*, 7(4):91–100, 2002.

[203] A. Zaccarin and B. Liu. Fast algorithms for block motion estimation. In *Proceedings of IEEE ICASSP*, volume 3, pages 449–452, 1992.

[204] Y.-Q. Zhang and S. Zafar. Motion–compensated wavelet transform coding for color video compression. In *Proceedings SPIE Visual Communications and Image Processing*, volume 1605, pages 301–316, January 1991.