Real Time Sub-Pixel Space-Time Stereo on the GPU

Jean–Daniel Nahmias

A dissertation submitted in partial fulfilment

of the requirements for the degree of

Doctor of Philosophy

of the

University of London.

Department of Computer Science

University College London

September 14, 2009

Table of Contents

List of Figures

List of Tables

Abstract

Recent advances in virtual reality, 3d computer generated graphics and computer vision are making the goal of producing a compelling interactive 3d face to face communication system more tractable. The problem with producing such a system is reconstructing the 3d geometry of the users in real-time.

There are many ways of tackling this problem however many of them require prior knowledge (i.e model fitting methods). These add unnecessary constraints and limit the usability of the system to reconstructing known entities. Other high quality methods using laser triangulation require too many samples and therefore cannot handle dynamic and deformable shapes such as the human face. A more suited approach is to use stereo based algorithm that function using two of more views and augmenting their capabilities using structured light.

The work presented in this thesis will examine and evaluate various stereo vision algorithms and hybrids with the goal of producing accurate 3d representations of human faces in real time. Various dynamic programming algorithms will be presented and hybrid variations. These will be extended into the space-time domain and the impact of using different structured light patterns with various algorithms and cost functions will be examined.

Most real-time correspondence algorithms are limited to producing pixel value disparities; these can be augmented into producing sub-pixel disparities by smoothing functions. Applying such smoothing functions tends to remove detail. Another approach is to use non-linear optimization on a spatial-temporal warp function. These algorithms tend to be very computationally expensive and therefore not feasible for real time applications. With recent development of GPUs (Graphics Processing Units) driven by the consumer demand for complex real time 3d graphics, these cards are capable of processing large amounts of data in parallel. This makes them very amenable to solving large linear algebra problems. .

The result being a tuneable stereo reconstruction framework that has been reformulated into streaming problems in order to be processed on the GPU to produce real time sub-pixel depth maps of human faces that can be triangulated to produce accurate 3d models.

Acknowledgments

I would like to thank my supervisor Prof Anthony Steed for his guidance and extreme patience as well as giving me the freedom to explore and research the work presented in this thesis.

I would also like to thank the members EPSRC funded Interdisciplinary Research Collaboration (IRC) project EQUATOR for their support and funding.

A special mention goes out to Simon Evans who helped and advised me in the development of the circuit board responsible for synchronizing the stereo cameras to a projector.

Dr Bernard Buxton for his feedback and advice along the way

All of this would not have been possible without the support of my parents, who encouraged and motivated me in times of need. I would like to dedicate this work to my father.

Chapter 1

Introduction

The main goal of virtual reality research is to immerse human beings in a perception of a digitally created reality. This is achieved by presenting the user with alternative sensory information using a variety of technologies. As virtual reality has evolved and its technologies become more mainstream (e.g. through special FX, video games, simulation and training) one of the main pursuits is to make these alternative perceptions or virtual environments more believable.

As adult human beings we seem to have expectations concerning the way in which the world around us functions. In virtual environments users project these expectations into their new digital world. When these expectations are broken the users become all too aware of the artificial environment they are experiencing. With this awareness the sensation of actually being in the environment, disappears. In the search to make virtual environments more believable, and to make the user feel more immersed the tendency has been to focus on realism. As these technologies mature and virtual reality becomes more realistic, a new application known as tele-immersion [28] has emerged. Tele-immersion is a form of tele-communication between users in virtual reality or mixed reality environments. Similar to video conferencing, the main objective of tele-presence is for face to face communication that can take place in a shared virtual world.

The greatest difficulty facing the development of a tele-immersive system is accurately capturing the user's behaviour in 3D. More importantly with regard to face to face communication, one needs to capture the user's facial animations as well as appearances. Capturing 3D facial animation data for the application of tele-immersion has the added constraint of the system having to run in real-time for it to be useable.

Recreating the physical reality in virtual environments has the advantage of adding realism and facilitating communication. This can be achieved in various ways

using a variety of instruments. Since what one is interested in is the visual properties of the world, the instrument of choice seems to be video cameras which are relatively cheap and can capture a large volume of data quickly. However video cameras only capture 2D data and virtual environments work with 3D data. One could take the 2D video and project it onto a plane, but one would lose most of the advantage of realism due to a loss of parallax. Fortunately it is possible to perform a 3D reconstruction for a sequence of images in certain heavily constrained environments, this falls into the research field of 3D computer vision or photogrammetry [42]. This thesis will focus on a constrained situation, where the intention is to sample close range dynamically deformable objects in real-time. Although the immediate goal is to focus on human beings, and more specifically their faces, care has been taken not to make many assumptions about the scene and its reconstructed objects. Not making any assumptions makes the system more flexible, and therefore enables it to potentially reconstruct the entire human body and or various other objects presented to it.

The interests of the research presented in this text lie in reconstructing and more specifically in 3D reconstruction of a scene with the target application being face to face communication. This can be achieved, using a variety of techniques that are covered in greater depth in the literature review. Some of these techniques have limitations such as only functioning offline. The common approach is to reconstruct the geometry of the head in a neutral pose and then apply motion capture techniques in order to track the facial animations. In these circumstances one refers to visual realism as a measure of how well the geometry matches the head and one refers to behavioural realism as a measure of how well the animation is portrayed. The work presented here will not focus on these model fitting approaches as they lack the flexibility to reconstruct more than just the human head.

Building on research in stereo vision, this report proposes to borrow research into 3D reconstruction from the field of computer vision and improve the visual appearance and performance of the results by leveraging the capabilities of modern graphics processing units (GPUs). Using GPUs enables one to run certain types of algorithms that would normally only be reserved to offline reconstruction in the context of real-time applications. These offline algorithms given less computational constraints tend to be optimized in the pursuit of visual quality as opposed to computational performance. The work presented here will demonstrate that GPUs have opened up the

possibility of using certain types of algorithms previously reserved to offline use in the context of real-time applications. It therefore hopes to blur the distinction between behavioural realism and visual realism by performing the 3D reconstruction in real-time and thereby implicitly modelling the behavioural elements.

1.1 Problem Statement

In the pursuit of presenting users with a more believable and realistic environment for both virtual reality and augmented reality, the intention is to build and develop a system capable of capturing the dynamic shape of other users, using off the shelf components such as digital cameras and projectors. The initial target application was to reproduce photorealistic avatars heads for virtual environments and tele-immersive communication. The system should provide visual realism as well as behavioural realism. Traditionally emphasis [9] has been on visual realism which involves spending more computation time offline, building highly accurate visual models of human faces or heads, and then dedicating a fraction of these resources on behavioural realism in the form of motion capture. The emphasis in this body of work is on capturing the geometric properties of the human face in real-time and thereby implicitly capturing behavioural animation thereby negating the need to perform motion capture. The aim is wish to capture the dynamically deformable nature of the human face, more specifically its deforming geometry. Although the initial target application is tele-immersion for face to face communication, it was felt that by explicitly not relying on a model-based approach often found in facial capturing systems, that the overall system would be more flexible and would be employed to capture more general purpose objects.

This thesis proposes ways of using digital cameras to sample the geometric properties of physical objects, such as humans, and examine the compromises between accuracy and realism as well as computational time. By using two or more cameras one can obtain the depth of each overlapping pixels. These depth images can either be triangulated to obtain a 3D mesh, or used in image-based rendering algorithms in order to obtain images from different viewpoints, thereby creating the illusion of 3D, which in turn creates a more realistic representation. The contributions of this thesis focus on retrieving this depth information from a stereo camera setup using structured light for

the potential use of 3D reconstruction or image-based re-rendering from differing viewpoints. With these goals the following research questions were posed.

Can a real-time sub-pixel scalable stereo reconstruction system be developed?
What are the potential benefits of using structured light?
What are the benefits of extending the support region into the time domain?
What is the optimum cost function for performance and quality?
What benefits can be achieved by a parallel system and can it leverage GPUs?
What compromises have to be made between computational performance and visual quality?
Can the correspondence problem be solved with alternative techniques that use less computational resources?

Precedence will be given to the performance of the 3D geometry acquisition system (i.e. whether it can it run in real-time) in order for it to be used in real-time applications such as tele-immersion. The quality of the results are of course important but the priority lies with real-time or interactive applications. Although the initial target application is face to face communication, the assumption of using a prior generic face model that would then be subsequently deformed to fit to the video data was avoided. Another goal was to keep the system as flexible as possible so that it could be utilized in order to capture other deformable dynamic objects and could easily be extended to the acquisition of the full human body, as well as various other applications.

1.2 Scope

Performing 3D reconstructions of heads was carried out by building and developing a real-time geometry acquisition system using off the shelf equipment. The only hardware that was developed was a simple circuit for synchronising the cameras and projector. Excellent performance gains could have been achieved with custom hardware design, which could have been prototyped using modern day field programmable gate arrays. However, this would have required significant development time and is beyond the scope of this thesis. Hardware acceleration was leveraged by using modern day GPUs that have not only been evolving faster than CPUs but have been designed to be fully

programmable with the use of vertex and pixel shaders. This has created a recent trend known as GPGPU, general purpose GPU. Recently GPUs have been used to solve a variety of problems outside the realm of 3D graphics. They have been used to solve PDEs [70], option pricing, vision, medical imaging [66] and in a variety of other applications.

1.3 Outline of Thesis

This thesis is structured in the following manner. A comprehensive literature review of methods for acquiring 3D shape information using various types of sensors, as well as a more detailed description of stereo methods will be covered in Chapter 2. Chapter 3 will describe a capture rig developed using stereo cameras and a projector. It will also describe experiments using dynamic programming algorithms with different cost functions and structured light patterns. The impact of different cost functions and light patterns as well as the extension of these algorithms into the time domain, will be discussed and analysed in terms of qualitative and computational cost. Chapter 4 will examine how the data captured in the previous chapter can be improved upon using non-linear optimization methods. The convergence of different solvers will be examined. All these solvers will require an initialization produced by the previous chapter's dynamic programming algorithms. The impact of this initialization will also be examined in Chapter 4 as well as quality of the newly produced sub-pixel disparity maps and some of the potential weakness of these methods and how they can be overcome using regularization. Chapter 5 will focus on how these algorithms can leverage graphic processing units by being parallelised and made to run in real-time. Chapter 6 will bring together all the work carried out in the previous chapters and present a scalable frame work for solving the stereo correspondence problem. This framework will be able to scale with resolution increase and performance increase either to produce superior results or reduce the acquisition time depending on the target applications. The conclusion drawn from this body of work as well as future directions for extending this research, will be discussed in Chapter 7.

1.4 Summary of Contributions

This thesis presents a wide range of techniques to solve the stereo correspondence problem. The following contributions were made.

- Analysis of modern dynamic programming algorithms and their ability to solve the correspondence problem under various structured light patterns using a variety of different cost functions as well as extending their support region into the time domain.
- Analysis of non-linear optimization methods for solving the space time stereo problem.
- Development of a multi scale, non-linear optimization algorithm for solving the sub-pixel stereo correspondence problem in real-time using graphics processing units.
- Development of a scalable frame work for acquiring 3D deformable objects depth in real-time.

`

1.5 Publications

Nahmias, J.D., Steed, A., Buxton, B.
Evaluation of Modern Dynamic Programming Algorithms for realtime Active Stereo Systems, *13th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision'2005, WSCG 2005*, University of West Bohemia, Campus Bory, Plzen-Bory, Czech Republic, January 31 - February 4, 2005, 113-116

Nahmias, J.D., Steed, A., Buxton, B.
Analysis of Cost Functions and Structured Light Patterns for Modern Dynamic Programming Stereo Algorithms, *IEE International Conference on Visual Information Engineering 2005*, University of Glasgow, Scotland, April 4 - 6, 2005

Chapter 2

Literature Review

This section of the thesis will focus on exploring the research space in which this work belongs. Figure 2.1 illustrates a taxonomy from [90] which is used to distinguish the various approaches to solving the acquisition of dynamic shapes. This chapter will focus on the space that is highlighted as it represents the use of digital cameras as sensors. This satisfies the criteria of using off the shelf equipment. As the research space is more in line with the 3D computer vision field, an alternative taxonomy is presented in Section 2.1. 3D computer vision is a very active field of research that has produced very extensive and promising results over the years. To fully cover the whole field in any useful depth is beyond the scope of this thesis.

Following the taxonomy presented in Section 2.1, a more in depth look at one specific area of 3D vision, namely stereo vision will be examined in greater depth. Section 2.2 will demonstrate that the stereo problem can be reformulated into a 1D correspondence problem. Section 2.3 will present a taxonomy for solving this 1D correspondence problem, while Section 2.4 will focus on some of the more modern techniques presenting the ones that achieve good computational performance with others performing better in terms of visual accuracy.

2.1 3D Computer Vision

3D computer vision is based on extracting 3D information from a dataset consisting of images acquired from digital cameras. It is a specialization of the general field of machine vision also known as computer vision. This area of research is very broad and diverse. Example applications include and are not limited to robotics, augmented reality, virtual reality, mixed reality, computer games, military, architectural and production line inspection, to name a few. Many diverse areas of research have tackled these

problems, and producing a comprehensive review of all the literature in these fields would be challenging to say the least. This section of the report will therefore firstly present a taxonomy of 3D vision research, followed by a study of stereo capturing systems, structured light capturing, modern advances in their implementations on GPUs as well as tele-immersive systems. For an introduction and overview of the field of 3D computer vision please refer to [22], [42], [82], [32].

2.1.1 Taxonomy

3D computer vision techniques that infer shape or structure can be categorised in various ways. The following are key distinctions employed in the subject matter literature.

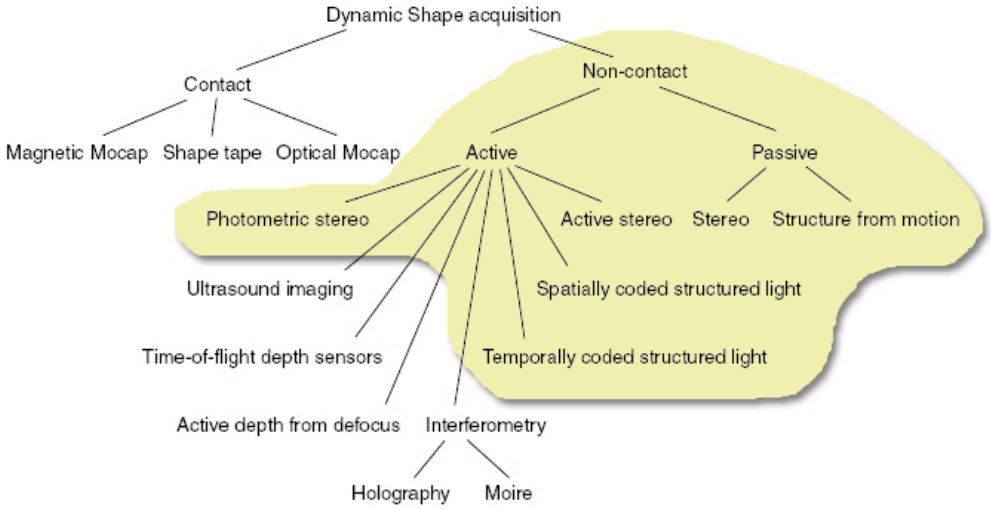

Figure 2.1 Taken from [90] illustrating Dynamic Shape Acquisition Taxonomy

Active versus Passive

All image based modelling techniques can fall into one of two groups, active and passive. Active techniques change the environment in some way (i.e. illuminate the environment) while passive techniques capture the environment without changing it. Active techniques are usually more accurate but expensive and not always viable. Passive techniques on the other hand are cheap and viable but at the cost of accuracy. Nevertheless active systems greatly improve correlation algorithms.

Autonomous versus Semi-Autonomous

Again all the following techniques can fall into one of these two groups. Autonomous systems require no user interaction, while semi-autonomous systems require varying degrees of user interaction. This field has been researched with different goals in mind. One example is robotics researchers who have pursued this area with robot navigation and exploration as their aim, and have therefore focused on trying to produce autonomous systems of very high accuracy. By contrast 3D graphics and special effects communities have prioritised aesthetically pleasing results at the cost of needing user interaction in order to rapidly produce compelling virtual environments.

Shape from Single View

These are techniques that rely solely on one image for their input. Because these techniques require such little input data they usually necessitate user interaction. One possible technique is presented by [18] and makes use of the following assumptions: 3 orthogonal sets of parallel lines, 4 known points on ground plane, 1 height in the scene. An alternative technique for single view that also scales to multiple views is presented by [20] and used in the Façade application (Section 2.4.2). Another interesting approach is presented in [61], where users specify lines that are then extruded into planes. These planes are subsequently used to create a coarse depth-map that is then manually refined. This multi-layered depth-map is then used to reconstruct geometry. For review of commercially available products please refer to [80].

The proposal in this report focuses more on reconstruction of avatar heads in a controlled environment. With the advances that have been made in digital cameras and computational power why not take advantage of using multiple images?

Shape from Stereo

Once a point in 3D space has been projected onto an image plane it loses its depth information. This point can lie anywhere along the ray passing through the centre of projection and the pixel the point was projected onto. However one can recover the depth information of that pixel if it is projected onto the image plane of another camera by using triangulation. Shape from stereo can be decomposed into two problems. The first problem is correspondence, given a pixel in image A representing the projection of

a 3D point, where is the projection of that point in image B? This can be reduced to a 1d search by using the epipolar constraint. The second problem is one of triangulation. Please refer to [72] and [74] for good surveys on stereo. Section 2.4.3 of this report will present a more detailed look at stereo along with some of the more recent work produced by Li Zhang et al.in Space-time Stereo [91]. Stereo vision has been extensively researched and has produced excellent results in certain constrained environments; it has therefore been used in avatar head reconstruction algorithms. Section 2.5 of this report will review some techniques employed for tele-immersion in the context of avatar reconstruction that rely on stereo algorithms. Lastly good candidates for real-time applications will be presented.

Shape from n-View

This refers to obtaining the 3D reconstruction from multiple images taken from different viewpoints. There are many algorithms that rely on different visual cues that can be used to achieve this task. In [54] an algorithm is presented that computes visual hulls from a set of images by extracting the silhouette of the object being reconstructed. Other interesting algorithms include Space Carving [47] and Voxel Colouring [73]. Zisserman [3] introduces a novel algorithm that uses edge features extracted from multiple images to calculate half planes and reconstruct geometry. Although these algorithms can produce good results in certain situations, they also pose greater constraints on the environment. Work using some of these space carving type algorithms for head reconstruction has been carried out by [89]. However, the results seem less compelling than stereo alternatives.

Structure from Motion

By using multiple images taken from a single camera over time, and finding corresponding features over the set of images Kruppa [44] proved that if 5 features could be detected from 2 images, the rotation and translation of the camera between the images and the 3D location of the 5 features could be determined up to an arbitrary scale. Ullman [83] and Longuet-Higgins [48] were early pioneers of this work. These algorithms rely on finding features such as corners [31], SIFT [50], or SURF [6] and their correlation across multiple frames using [59] in order to solve the system. More recently the theoretical foundations of these works are used to track camera motion in

commercially available software such as Boujou [1] and PFTrack [81]. These algorithms produce very good results for camera tracking but because they are based on features, the 3D information inferred by the techniques is very sparse and lacking in detail as opposed to dense stereo techniques that provide depth for each pixel. These algorithms are also very computationally expensive making them much less suitable for real-time applications.

Shape from Shading

These algorithms use surface properties and lighting models to determine the surface normals. One simple algorithm developed by [35] demonstrates this very simply as follows. If one were to take a picture of two objects with the same surface properties (e.g. the same surface material) under the same lighting conditions, using a perfect sphere and arbitrary second object, one can easily determine the surface normals of the second object. This is achieved, firstly by generating normal values for each projected pixel of the sphere (this is done simply by registering 3D sphere with the picture), after which the normal values of the arbitrary second object are determined simply by looking up each pixel value against a matching pixel value of the projected sphere and assigning its corresponding normal value. This is a very simple example; many more algorithms have been developed that usually rely on extracting the BRDF of materials see [42]. Shape from shading algorithms tend to work well on synthesized pictures but seem to fail with real data. Failure can be caused by specular highlights, image noise or ambiguities (given a single light source and viewing direction many different geometric shapes can be rendered to produce the same image). For a more detailed overview of shape from shading please refer to [67].

2.2 Stereo Formulation

As mentioned previously, an image pixel's 3D location can lie anywhere along the ray passing from the centre of projection through that point. If one has another image taken from a different point of view containing the projection of the same point, one can infer the 3D location of that point through triangulation. Triangulation describes the intersection of the ray passing through the centre of projection and the pixel of the first image with its equivalent in the second image. However in practice these rays will never

intersect due to sampling and correspondence errors. The stereo problem can therefore be broken down into two sub-problems, the first being one of correspondence and the second being triangulation. One possible way of solving triangulation is to find the vector representing the shortest distance between the two rays and choosing its mid-point. The stereo problem can be solved for two cases; the calibrated (where the camera's intrinsic parameters are known) and un-calibrated. The un-calibrated case allows the determination of a pixel's depth relative to an arbitrary scale value. The most difficult problem facing stereo vision is the one of correspondence. Pixels represent a quantized sampling of light, they do not represent actual points in the scene, and therefore one pixel can contain a region producing a modulated value of the edge transition while this same edge could lie at a pixel boundary in the other image and therefore produce two different pixel values in the second image. This highlights just one potential problem, another is occlusion; some pixels in one image are simply occluded in the other. The correspondence problem can also be extremely hard to solve for images of bland textureless and featureless scenes.

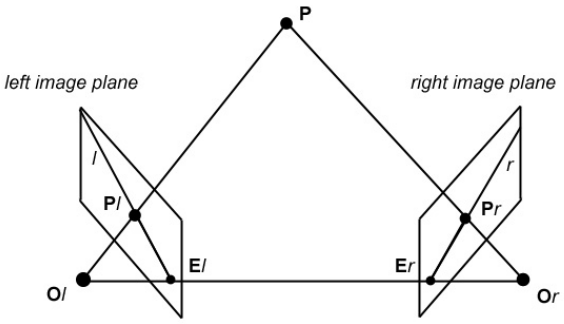

Figure 2.2 Epipolar Constraint

This type of scene as well can be tackled by using an active system. An active system will project light onto the environment and by carefully choosing the type of projected pattern, it is possible to significantly increase the algorithm's ability to find corresponding matches between images. To make the correspondence problem more manageable many algorithms use constraints. One such constraint which enables the correspondence problem to be reduced from a 2D to a 1D search is the epipolar constraint.

2.2.1 Epipolar Constraint

Given a point P projected on the left image plane as P_l with centre of projection O_l, the projection on the right image plane P_r with centre of projection O_r will lie on the epipolar line r. The epipolar plane is defined by the points P, O_r, and O_l. One can therefore calculate the essential matrix E that maps the point P_l onto the epipolar line r. This matrix can be calculated as follows.

The transformation from the left coordinate frame to the right coordinate frame is a translation followed by a rotation. Therefore

$$\begin{aligned} &\overline{P}_r = R\left(\overline{P}_l - T\right) where \\ &T = \left(O_r - O_l\right), \overline{P}_r = \left(P - O_r\right), \overline{P}_l = \left(P - O_l\right) \end{aligned}$$

2.1

The equation of the epipolar plane is then:

$$\begin{aligned} &\left(\overline{P}_l - T\right)^T T \times \overline{P}_l = 0 \\ &\left(R^T \overline{P}_r\right)^T T \times \overline{P}_l = 0 \\ &T \times \overline{P}_l = S.\overline{P}_l \end{aligned}$$

2.2

Where

$$S = \begin{bmatrix} 0 & -T_z & T_y \\ T_z & 0 & -T_x \\ -T_y & T_x & 0 \end{bmatrix}$$

2.3

Given

$$\begin{aligned} &\overline{P}_r^T \cdot E \cdot \overline{P}_l = 0 \\ &E = R \cdot S \\ &\overline{P}_r^T \cdot E \cdot \overline{P}_l = 0 \therefore \\ &P_r^T \cdot E \cdot P_l = 0 \therefore \\ &r = E \cdot P_l \end{aligned}$$

2.4

However r is in normalized image plane coordinates. In order to achieve a similar mapping in pixel coordinates one has to calculate the fundamental matrix F as follows:

$$F = M_r^T \cdot E \cdot M_l^{-1}$$

2.5

Where M_r and M_l are the matrices representing the right and left cameras intrinsic parameters.

Once the stereo system has been calibrated (i.e. F found) the correspondence search can be reduced to 1D. One can go one step further and rectify both images so that the actual scan-lines of the images are matched with the epipolar lines. For further details please refer to [42],[82],[22].

2.2.2 Correspondence Algorithms and Taxonomy

The problem of stereo correspondence has been widely studied and is still a very active area of research. In [72] a taxonomy and evaluation of twenty dense two frame stereo algorithms are presented. Most stereo algorithms produce disparity values (i.e. differences between corresponding points along the epipolar line) that, are inversely proportional to the depth values. This definition can be generalized for multiple viewpoints but we will focus on the simple case of two viewpoints. This section will focus on algorithms that produce a dense disparity map, which are the algorithms that calculate disparity values for each input pixel. These algorithms work by selecting a cost function for correspondence, which is then minimized. They function by performing all or a subset of the following procedures.

- Cost Computation
- Cost Aggregation (Support Region)
- Disparity Computation / Optimization
- Disparity Refinement

These algorithms fall into one of the following two categories; local or global. Local algorithms minimize the cost function independently over a window while global algorithms make explicit assumptions such as a smoothness constraint and then solve an optimization problem over a scan-line or complete image.

Cost Computation

Cost computation is the method used to calculate the error of a particular disparity value. This represents the error of matching a pixel in the left image with a pixel in the right image, alternatively if a support region is used, the error of matching a window of pixels in the left image against another window in the right image. This region can also be extended into the time domain and used as a support region that spans multiple images. Example cost functions can be AD (Absolute Difference), SD (Square Difference), SSD (Sum of Squared Difference), and normalized cross correlation, see [72] for further examples of cost functions used in correspondence algorithms. Global algorithms tend to use an error function that combines the cost functions with another term. This term usually adds constraints such as occlusions (e.g. dynamic programming where a fixed penalty is associated with an occlusion) and or smoothness.

Cost Aggregation

Stereo algorithms usually aggregate cost, such as sum or average over a support region. These support regions usually span the spatial domain however, more recently better results have been obtained by using support regions that also span the temporal domain. Examples of such algorithms will be presented in Chapter 3**.** The problem with using square support regions (e.g. 5x5 window of pixels) is that they make the assumption that the surface is fronto-parallel. Although this maybe the case in certain circumstances in reality it is rarely so. To compensate, these algorithms usually use large support regions that tend to average the surface out and produce blurred disparity maps lacking any fine detail. To overcome this limitation, some global algorithms also tend to optimize a warp function for the support region. As one shall see in Chapter 4 this can produce substantially superior results.

Disparity Computation / Optimization

With locally optimizing algorithms finding the disparity is trivial, the correlation with the lowest cost is selected. These algorithms tend to focus on the cost function and aggregation steps. These Winner Take All type algorithms, although computationally inexpensive also tend to perform badly. While with global algorithms offers a disparity function $E(d)$ that minimizes the energy is solved.

$$E(d) = E_{data}(d) + \lambda E_{smooth}(d)$$
$$E_{data}(d) = \sum_{x,y} C\big(x, y, d(x,y)\big)$$
$$E_{smooth}(d) = \sum_{x,y} \rho(d(x,y) - d(x+1,y)) + \rho(d(x,y) - d(x,y+1))$$

2.6

E_{data}(d) is the data term measuring how well the disparity function matches the input image pair, using the cost function *C*. E_{smooth}(d) is the smoothness constraint making the minimization computationally tractable and is often restricted to neighbouring pixel as shown in the above example. *p* is some monotonically increasing function of disparity. In order to solve the above optimization, one can use a variety of different algorithms including simulated annealing [64], graph cuts [43] [78], dynamic programming [8] [77], Lucas and Kanade (i.e. Gauss-Newton) [51], cooperative [96], max flow [11]. With computational performance of modern day PCs the focus of research in this field has focused on global type algorithms which produce superior results. However DP (Dynamic Programming) has the distinct advantage that it can find a solution for independent scan-lines in polynomial time and is therefore a candidate for real-time applications. An overview of DP algorithms as well Lucas and Kanade [51] and some modern variations will be presented in Chapter 3, 4 and 5.

Disparity Refinement

Most disparity computation algorithms, with the exception of some of the global optimization variants, tend to produce disparity maps with outliers and other types of artefacts. Disparity refinement is a final post-processing stage aimed at correcting some artefacts or smoothing the results. One example of this is the Birchfield et al.[8] algorithm, which has a final post processing stage that uses a region growing type algorithm to try and eliminate certain disparity outliers. One could also apply parabola fitting to the disparity values in order to approximate a higher sampling rate and estimate sub-pixel disparity values. This could be incorporated into the smoothness term of a global optimization algorithm.

The following section will describe some stereo correspondence algorithms, namely dynamic programming and its more recent variation developed by Criminsi et al.[19] as well the Lucas and Kanade [51] algorithm that has substantially evolved over

the years. Li Zhang et al.[91] work on space-time stereo will be discussed and finally I will present some of the work carried out by [56], with regard to stereo for the practical application of tele-immersion.

2.2.3 DP Stereo Formulation

This section will cover the standard DP stereo formulation [42] as well as the variations due to Birchfield et al.[8] and Criminisi et al.[19].

2.2.3.1 Traditional DP Stereo

Given a pair of rectified images $I_l(x)$ and $I_r(y)$ representing the left and right images at the x[th] and y[th] pixels respectively for a given scan-line, it can be shown from [7] that the depth of a given pixel is inversely proportional to its disparity $(x - y)$. The problem is therefore one of correspondence. Using the *uniqueness* and *monotonic ordering* constraint, DP algorithms will solve the disparity by minimizing a cumulative cost function *C(l, r)* defined as follows:

$$C(l,r) = \min \begin{cases} C(l-1,r) + OccCost \\ C(l-1,r-1) + M(l,r) \\ C(l,r-1) + OccCost \end{cases}$$

2.7

.

Here, *OccCost* is a parameter of the system and defines a penalty for occlusions and *M(l,r)* is a cost function that defines the dissimilarity between two pixels *l* and *r* of the left and right scanline respectively. It is quite common for *M(l,r)* to be the sum of squared difference (SSD) defined as:

$$M(x,y) = \sum (I_l(x) - I_r(y))^2$$

2.8

.

The recurrence in *C(l, r)* defines the possible moves in the forward pass of the DP algorithm, namely: one horizontal occluded move, one diagonal matched move, and one

vertical occluded move. After initialisation of the cost matrix the DP algorithm iterates through each cell within the constraint network calculating *C(l, r)* and storing a backwards link to the previous cell containing the minimum cost. Once the cost matrix has been calculated, the second stage of the algorithm is a backwards pass that follows all the stored links to produce the minimum cost path and therefore the disparity for that scan-line. This is repeated for each scan-line in the pair of images and a disparity map is produced. Figure 2.3 illustrates the possible moves of the DP algorithm.

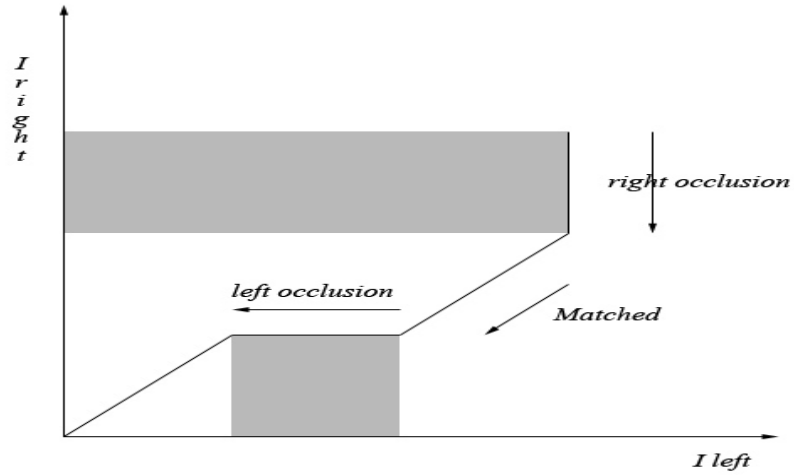

Figure 2.3 Allowed Dynamic Programming Moves

2.2.3.2 Birchfield DP Algorithm

The Birchfield et al. [8] algorithm differs from the traditional DP algorithm in a few key ways. Firstly, the cumulative cost function and the dissimilarity measure are defined as follows:

$$C(l,r) = \min\begin{cases} C(l-1,r) + OccCost \\ C(l-1,r-1) + M(l,r) + MatchR \\ C(l,r-1) + OccCost \end{cases} \quad 2.9$$

where:

$$M(x_i, y_i) = \max\{0, I_L(x_i) - I_{\max}, I_{\min} - I_L(x_i)\}$$
$$I_{\min} = \min(I_R^-, I_R^+, I_R(y_i)), I_{\max} = \max(I_R^-, I_R^+, I_R(y_i))$$
$$I_R^- = \frac{1}{2}(I_R(y_i) + I_R(y_i - 1)), I_R^+ = \frac{1}{2}(I_R(y_i) + I_R(y_i + 1)) \quad 2.10$$

Again, *OccCost* and *MatchR* are parameters of the system that define an occlusion cost and match reward respectively. The dissimilarity function *M(x,y)* measures how well the intensity at *x* fits the linearly interpolated region around *y*.

Another change is the addition of a constraint that intensity variation accompanies depth discontinuities. An intensity variation is defined as any set of three pixels whose min and max levels vary more than four grey scale values. This threshold is very low and is intended to prevent the algorithm from making poor choices in regions of the image that do not contain much information. It also specifies on which side the depth discontinuity must lie with respect to the intensity variation and also requires occlusions to be accompanied by the intensity variation on the appropriate side. This is illustrated nicely in [8].

The cost matrix is also computed in a different manner. Instead of iterating through each cell and computing the minimal cumulative cost of reaching a particular cell, the algorithm computes the cumulative cost of reaching the neighbouring cells through the particular cell currently being evaluated. If this is lower than the neighbouring cell's current cost, that neighbouring cell is updated. Intuitively this can be thought of as looking forwards instead of backwards while evaluating the cumulative cost matrix. The computational cost is equivalent to that of the traditional DP algorithm [42]. However, it permits the algorithm to prune the cost matrix and further reduce the number of cells that need to be evaluated. This can speed up the running time quite considerably; instead of taking $O(n\Delta^2)$, where n is the number of pixels of the left and right scan-line and Δ is the maximum disparity, the computational cost approximates $O(n\Delta \log \Delta)$. Readers are referred to [8] for a comprehensive explanation of the pruning technique.

Once the cost matrix has been calculated, the initial estimates of the disparities are further refined by post processing steps. Firstly outliers are removed. Outliers are disparity values that are surrounded by different disparity values in agreement with each other. Then, the disparities are classified into three types; slightly reliable, moderately reliable and highly reliable, based on how many continuous disparities are in agreement along the y-axis. Moderately and highly reliable disparities are then propagated along the y-axis until they reach a slightly reliable disparity with a lower disparity value or a position that represents an intensity variation in the Left image. Moderately reliable disparities differ from highly reliable disparities in that they will not override their

neighbours if the disparity variation is just one. This helps overcome some of the artefacts usually present in DP algorithms that cause separate scan-lines of disparities to be out of alignment. This process is then repeated in a second pass along the x-axis. In Section 4 I will show that these post-processing steps can cause serious problems when used in conjunction with structured light.

2.2.3.3 Criminisi DP Algorithm

In Criminisi et al. [19] a new DP algorithm is proposed with the motivation of creating a depth map in order to be used in conjunction with an image based rendering technique that morphs two images to create a new image from a different viewpoint. I evaluate this algorithm from the point of view of 3D reconstruction. The algorithm uses a three-plane graph, a left occluded plane L, a matched plane M and a right occluded plane R (see Figure 2.4). This model allows a total of thirteen moves in the DP and has the advantage of allowing a much finer grain control of penalty costs.

$C_L(l,r), C_M(l,r), C_R(l,r)$ for each plane L, M and R respectively:

$$C_L(l,r) = \min\begin{cases} C_L(l,r-1)+\alpha \\ C_M(l,r-1)+\beta \end{cases} \tag{2.11}$$

$$C_R(l,r) = \min\begin{cases} C_R(l-1,r)+\alpha \\ C_M(l-1,r)+\beta \end{cases} \tag{2.12}$$

$$C_M(l,r) = M(l,r) + \min\begin{cases} C_M(l-1,r) \\ C_L(l-1,r)+\beta \\ C_R(l-1,r)+\beta \\ C_M(l,r-1) \\ C_L(l,r-1)+\beta \\ C_R(l,r-1)+\beta \\ C_M(l-1,r-1) \\ C_L(l-1,r-1)+\beta \\ C_R(l-1,r-1)+\beta \end{cases} \tag{2.13}$$

Here, α is the cost of moving within an occluded plane and β the cost of making a transition between planes. *M(l,r)* is a windowed normalized cross-correlation:

$$M(l,r) = (1 - M'(l,r))/2$$

2.14

Where

$$M'(l,r) = \frac{\sum (I_L - \bar{I}_L)(I_R - \bar{I}_R)}{\sqrt{\sum (I_L - \bar{I}_L)^2 \sum (I_R - \bar{I}_R)^2}}$$

2.15

In [6] the dissimilarity matrices are stacked across all the scan-lines and Gaussian smoothed with a kernel orthogonal to both left and right scan-lines.

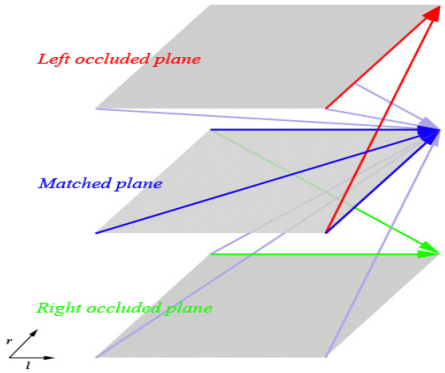

Figure 2.4 Taken from Criminisi et al. showing the 13 possible moves of the algorithm [19] with each plane representing the Cl, Cr and Cm cumulative cost matrices

2.2.4 Lucas and Kanade Stereo Algorithm

The Lucas and Kanade [51] algorithm originally developed for stereo correspondence is basically an image alignment algorithm. The goal of the algorithm is to align a template T(x**)** to an image I(x**)** where x is a column vector (y,x,t) representing pixel coordinates. The template T(x) can be a sub-region of the left image (5x5 window) while I(x) can represent the right image or vice versa. Let W(x:p) denote the parameterized set of allowed warps, where p = (p1, p2 ….pn) is a vector of parameters. The warp W(x:p) maps the pixels x in the coordinate frame of the template T into the sub-pixel location in the coordinate frame of the image I. An example warp could be a simple translation in the x, y axis defined as follows:

$$W(x:p) = \begin{pmatrix} x + p_1 \\ y + p_2 \end{pmatrix}$$

2.16

For example this warp could be used to determine optical flow. However generally if one were to track a larger patch moving in 3D a more general affine warp would be a better choice.

$$W(x:p) = \begin{pmatrix} (1+p_1)\cdot x + p_3 \cdot y + p_5 \\ p_2 \cdot y + (1+p_4) + p_6 \end{pmatrix} = \begin{pmatrix} 1+p_1 & p_3 & p_5 \\ p_2 & 1+p_4 & p_6 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

2.17

Equation 2.17 defines an affine warp with 6 parameters $p = (p_1, p_2, p_3, p_4, p_5, p_6)^T$. In the original implementation of the algorithm described in [51] a simple translation was used, and given an input stereo pair of rectified images one could just use a translation in the x axis. However, this would be based on the assumption that the surfaces are fronto-parallel, a much more suitable warp for the application of stereo will be defined in the next section and extended for a 3D support region. The algorithm can also be extended to support an arbitrary complex set of affine warps.

The goal of the algorithm is to minimize the sum of the squared error between the template T and the image I warped back onto the coordinate frame of the template.

$$\sum_x [I(W(x:p)) - T(x)]^2$$

2.18

Equation 2.18 is minimized with respect to the parameter p of the warp over all pixels in the template support region. To produce a dense disparity map this would be done per pixel. The Lucas and Kanade [51] algorithm assumes an initial estimate of the parameters p and solves iteratively for increments Δp i.e. the following expression is minimized:

$$\sum \left[I\left(W\left(x : p + \Delta p\right)\right) - T\left(x\right)\right]^2$$

2.19

With respect to Δp, and the parameters are updated:

$$p \leftarrow p + \Delta p$$

2.20

These two steps are performed iteratively until p converges, Δp is computed using equation 2.25.

2.2.4.1 Derivation of the Lucas and Kanade Algorithm

Equation 2.19 is minimized using a non-linear Gauss-Newton gradient descent non-linear optimization algorithm. This is achieved by linearizing Equation 2.19 using a first order Taylor series approximation on $I\left(W\left(x : p + \Delta p\right)\right)$ to give:

$$\sum \left[I\left(W\left(x : p\right)\right) + \nabla I \frac{\partial W}{\partial p} \Delta p - T\left(x\right)\right]^2$$

2.21

Where $\nabla I = \left(\frac{\partial I}{\partial x}, \frac{\partial I}{\partial y} \right)$ is the gradient of the image I evaluated at W(x:p). $\frac{\partial W}{\partial p}$ is the Jacobian of the warp, if $\mathrm{W(x : p)} = \left(\mathrm{W_x}(x : p), W_y(x : p)\right)^T$ then:

$$\frac{\partial W}{\partial p} = \begin{pmatrix} \frac{\partial W_x}{\partial p_1} & \frac{\partial W_x}{\partial p_2} & \cdots & \frac{\partial W_x}{\partial p_n} \\ \frac{\partial W_y}{\partial p_1} & \frac{\partial W_y}{\partial p_2} & \cdots & \frac{\partial W_y}{\partial p_n} \end{pmatrix}$$

2.22

For example the affine warp in Equation 2.18 has the Jacobian:

$$\frac{\partial W}{\partial p} = \begin{pmatrix} x & 0 & y & 0 & 1 & 0 \\ 0 & x & 0 & y & 0 & 1 \end{pmatrix}$$

2.23

The partial derivative of Equation 2.21 with respect to Δp is:

$$\sum_x \left[\nabla I \frac{\partial W}{\partial p}\right]^T \left[I(W(x:p)) + \nabla I \frac{\partial W}{\partial p}\Delta p - T(x)\right]$$

2.24

Setting Equation 2.24 to zero gives us the closed form solution to Equation 2.21 as:

$$\Delta p = H^{-1} \sum_x \left[\nabla I \frac{\partial W}{\partial p}\right]^T \left[T(x) - I(W(x:p))\right]$$

2.25

Where H is the Gauss-Newton approximation to the Hessian matrix:

$$H = \sum_x \left[\nabla I \frac{\partial W}{\partial p}\right]^T \left[\nabla I \frac{\partial W}{\partial p}\right]$$

2.26

A summary of the Lucas and Kanade algorithm is as follows:

Iterate:

1. Warp I with W(x:p) to compute I(W(x:p))
2. Compute the error image T(x) – I(W(x:p))
3. Warp the gradient ∇I with W(x:p)
4. Evaluate the Jacobian $\frac{\partial W}{\partial p}$ at (x:p)
5. Compute steepest decent images $\nabla I \frac{\partial W}{\partial p}$
6. Compute the Hessian matrix using Equation 2.26
7. Compute $\sum_x \left[\nabla I \frac{\partial W}{\partial p}\right]^T \left[T(x) - I(W(x:p))\right]$
8. Compute Δp using Equation (10)
9. Update the parameters $p \leftarrow p + \Delta p$

Until $\|\Delta p\| \leq \varepsilon$

This summarises the Lucas and Kanade algorithm as presented in [51]. However, it has been generalised for any warp, where the original formulation was specific to a simple translation along the x-axis. For simple warps such as translations and affine the Jacobian can be constant. However, in general all 9 steps have to be repeated for each iteration because the parameters *p* change from iteration to iteration. In [5] a general framework for the Lucas and Kanade algorithm [51] as well as all the variations that have been developed over the last 20 years, is presented. This algorithm is referred to as the forward additive. Figure 2.5 is a graphical representation of the algorithm taken from [5].

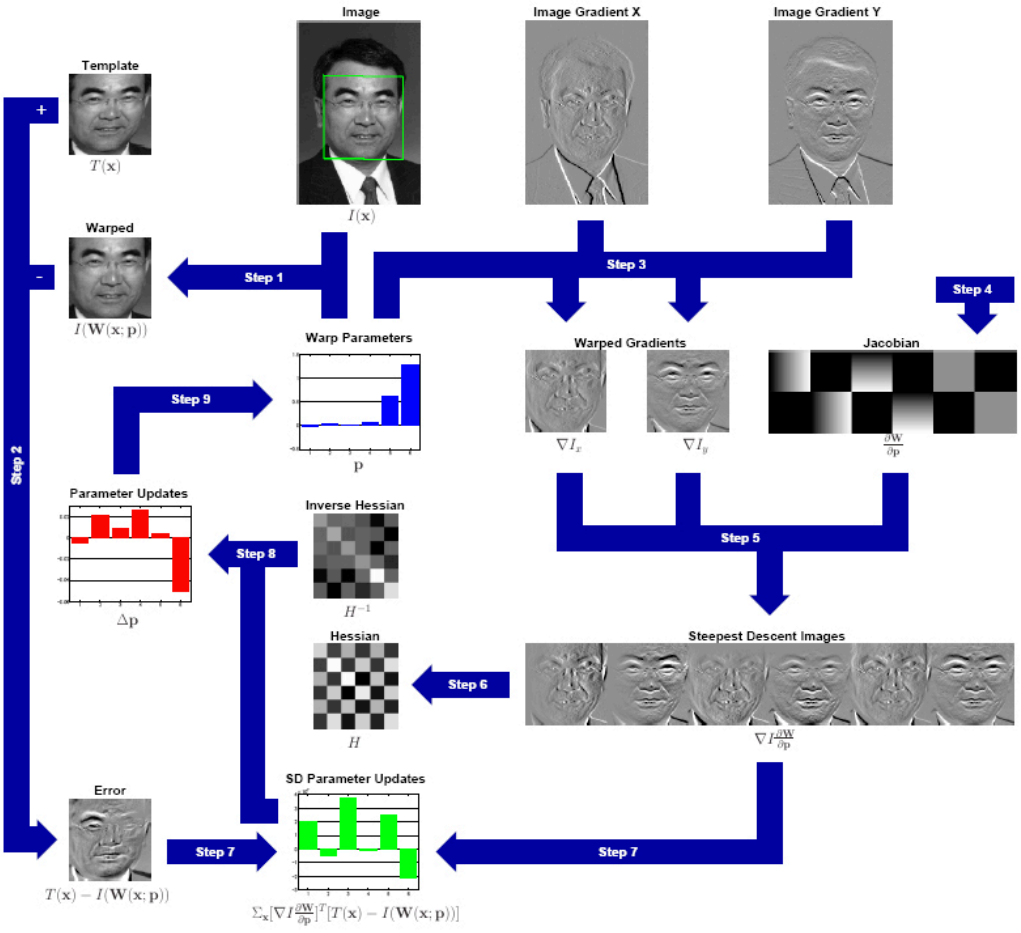

Figure 2.5 Representing a the 9 steps involved for the forward additive algorithm taken from [5]

2.2.4.2 Computational Cost

This section will briefly discuss the computational cost of the forward additive Lucas and Kanade algorithm [51] Table 2.1 illustrates the computational cost given *n* is the number of warp parameters and N is the number of pixels in the template.

Step 1	Step 2	Step 3	Step 4	Step 5	Step 6	Step 7	Step 8	Step 9	Total
$O(nN)$	$O(N)$	$O(nN)$	$O(nN)$	$O(nN)$	$O(n^2N)$	$O(nN)$	$O(n^3)$	$O(n)$	$O(n^2N + n^3)$

Table 2.1 Computational Cost of the Forward additive Lucas and Kanade [51]

Table 2.1 shows the total computational cost, as well as the cost of each step. As one can see step 6 is the most expensive. This table represents the cost of matching one template to an image. However for the purposes of stereo reconstruction the aim is to produce a dense disparity map for each pixel in the images. This could be achieved by creating a template for each pixel in the left image. The template would define the support region this could be a 5x5 pixel window. In the simple case of using a square window on a pair of rectified images only one warp parameter would be necessary i.e. a translation along the x-axis. However, given a pair of images of 640x480 pixels in resolution and given that this cost arises per iteration and that one would typically require at least 5 iterations (see Section 4) for the algorithm to produce acceptable results, this algorithm becomes quickly prohibitively expensive.

The following section will describe work carried out by Li Zhang et al.[91] that takes this algorithm and extends it into the space-time domain used in conjunction with structured light to produce very promising results.

2.2.5 Space-time Stereo

In [91] a novel stereo framework is presented, and is implemented using a combination of Dynamic Programming as well the Lucas and Kanade [51] algorithm. This section of the report will describe the work carried out in [91]. The framework is designed for local based stereo algorithms using a spatial-temporal window. The framework assumes for input two time varying video streams I_{left} & I_{right} that have been rectified with the help of camera calibration. In order to perform 3D reconstruction one needs to estimate the disparity function *d(x,y,t)* for each pixel *(x,y)* at time *t.* Traditional stereo algorithms achieve this by minimizing the following cost function.

$$E(d(x_0, y_0, t_0) = \sum_{(x,y)\in W_0} e(I_l(x, y, t_0), I_r(x - d(x_0, y_0, t_0), y, t_0))$$

2.27

Where *e* is a dissimilarity measure and usually defined as follows (i.e. SSD).

$$e(a,b) = (a-b)^2$$

2.28

However due to radiometric differences between cameras Li Zhang et al.[91] used the following metric:

$$e(a,b) = (a \cdot s + o - b)^2$$

2.29

Where s is a scale and o is an offset value dependent on the support region size.

When the above dissimilarity measure is applied to the above cost function, the cost function is referred to SSD. In order to incorporate temporal variation, one can use the following cost function.

$$E(d_0) = \sum_{t\in I_0} \sum_{(x,y)\in W_0} e(I_l(x, y, t), I_r(x - d_0, y, t))$$

2.30

This cost function reduces ambiguity by simultaneously matching in multiple frames. One distinct advantage is that the spatial window can be shrunk while at the same time extending the temporal window.

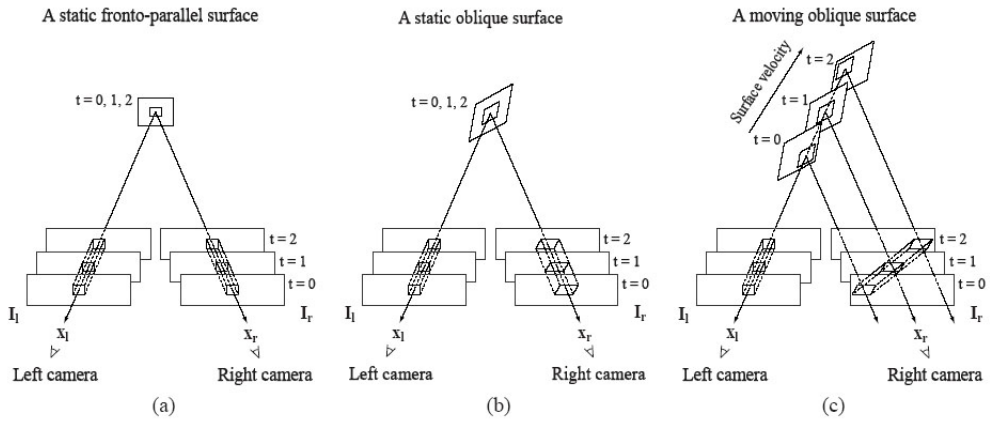

Figure 2.6 Taken from [91] depicting the space time window for static fronto-parallel, static oblique, and time varying oblique surfaces

The previously mentioned dissimilarity measure assumes constant disparity across the spatial window (i.e. static fronto-parallel surfaces). One can use a dissimilarity measure that better approximates oblique static surfaces as follows.

$$d(x,y,t) \approx \widehat{d}_0(x,y,t) \equiv d_0 + d_{x0} \cdot (x - x_0) + d_{y0} \cdot (y - y_0)$$

2.31

Where d_{x0} and d_{y0} are the partial derivatives of the disparity function with respect to the spatial coordinates *x* and y, at (x_0, y_0, t_0). This results in the following SSSD cost function to be minimized.

$$E(d_0, d_{x0}, d_{y0}) = \sum_{t \in I_0} \sum_{(x,y) \in W_0} e(I_l(x,y,t), I_r(x - \widehat{d}_0, y, t))$$

2.32

This will stretch and shear the window as shown in Figure 2.6 (b).

The above dissimilarity measure assumes oblique static surfaces. However one can devise another dissimilarity measure that will skew the window temporally as shown in Figure 2.6 (c). This measure will use a first order linear approximation of disparity variation across the temporal coordinates as well the spatial coordinates and is presented in Equation 2.33. This will have the benefit of better modelling moving object across the temporal domain.

$$d(x,y,t) \approx \widehat{d}_0(x,y,t) \equiv d_0 + d_{x0} \cdot (x - x_0) + d_{y0} \cdot (y - y_0) + d_{t0} \cdot (t - t_0)$$

2.33

Where d_{t0} is the partial derivative of the disparity with respect to time at (x_0, y_0, t_0). This can form the following cost function.

$$E(d_0, d_{x0}, d_{y0}, d_{t0}) = \sum_{t \in I_0} \sum_{(x,y) \in W_0} e(I_l(x,y,t), I_r(x - \widehat{d}_0, y, t))$$

2.34

When the above cost function is used with a camera of high frame rate, thereby minimizing the errors induced by linear approximations to motion, the results are very encouraging, as shown in Figure 2.8.

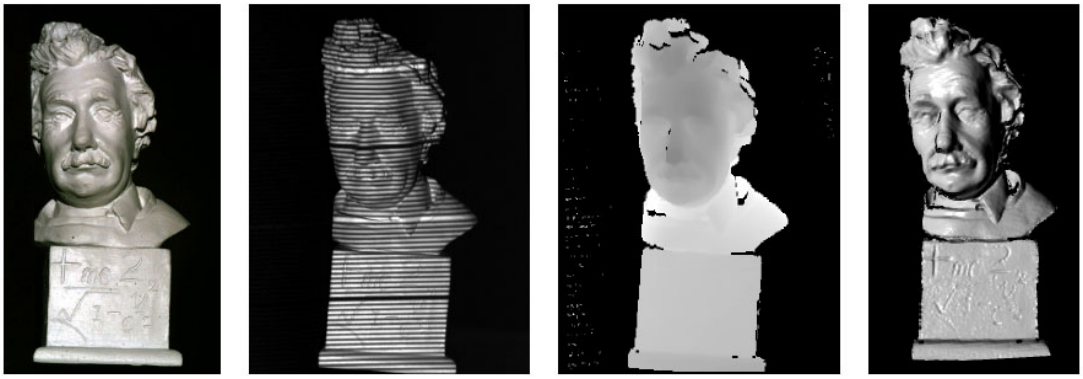

Figure 2.7 Screen shots taken from [91] showing the results of the above algorithm

Li Zhang et al.[91] solved Equation 2.34 by firstly using a simple DP algorithm that was extended into the temporal domain. This was simply done by using a 3D support region across multiple frames from the left and right camera. The results of this were then used as an initial estimate for the following stage, which used the previously mentioned forward additive Lucas and Kanade [51] algorithm. The forward additive Lucas and Kanade [51] algorithm was extended into the time domain and the warp was modified to coincide with the one specified in Equation 2.34. To derive this extension it is easier to revert back to the original Gauss-Newton optimization algorithm as follows:

Given m functions f_1 ..... f_m of n parameters p_1..... p_n one wants to minimize the sum:

$$S(p) = \sum_{i=1}^{m} \left(f_i(p)\right)^2$$

2.35

Where p stands for the vector $(p_1 \ldots p_n)$

The Gauss-Newton converges by starting off with an initial guess for p then iteratively updating it with following recurrence relation:

$$p^{k+1} = p^k - \left(J_f\left(p^k\right) J_f\left(p^k\right)^T\right)^{-1} J_f\left(p^k\right) f\left(p^k\right)$$

2.36

Where $f = (f_1 \ldots f_m)$ and $Jf(p)$ denotes the Jacobian of f with respect to p

To apply this algorithm to space-time stereo as demonstrated in [91] *f* would be replaced by Equation 2.32 the vector $p = (d, d_x, d_y, d_z)$ and the *J* would be the Jacobian of *f* with respect to *p*. Further details of this implementation will be discussed in Chapter 4 of this report. This local algorithm produces very good results, however, there are certain banding artefacts as shown in Figure 2.8. These are produced by the fact that the parameters d_x, d_y and d_z are not constrained by the disparity gradient with respect to the relevant axis.

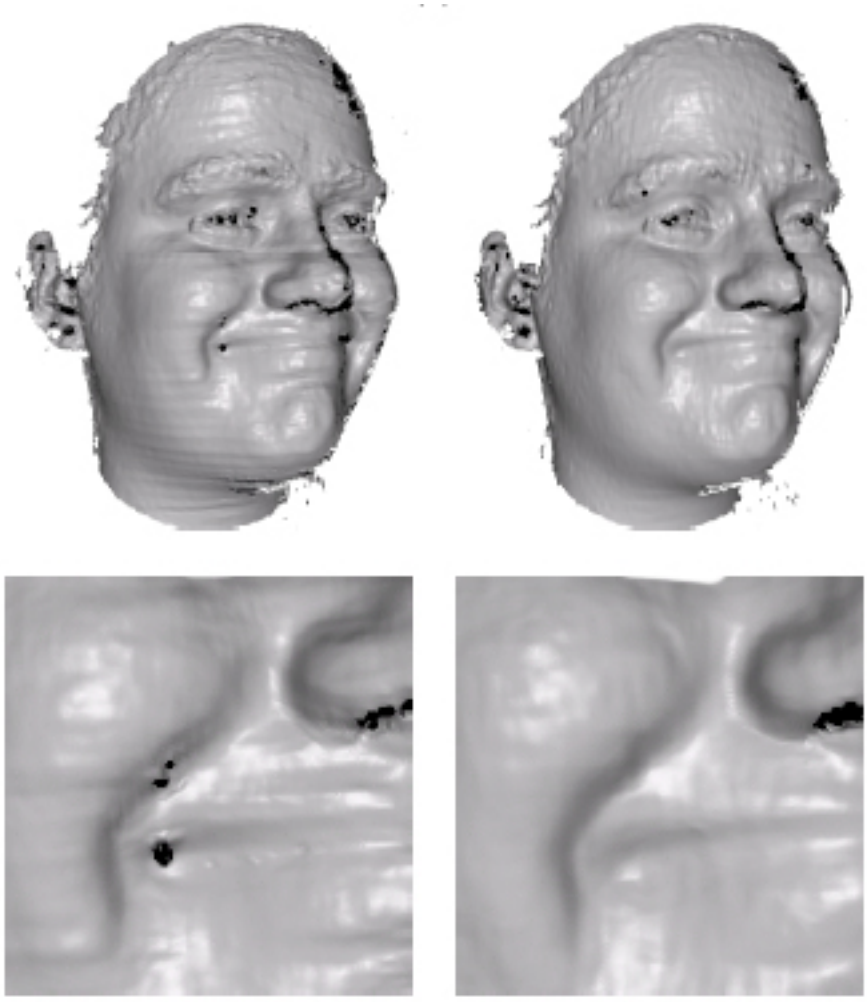

Figure 2.8 Taken from [91] illustrating the difference between the local (left images) space-time algorithm and global (right images)

To overcome this banding artefact Li Zhang et al.[92] reformulated the error function to be global, as follows:

$$\Gamma\left(\{d(x,y,t)\}\right) = \sum_{x,y,t} E\left(d, d_x, d_y, d_t\right) \qquad 2.37$$

Subject to the following constraints:

$$d_x(x,y,t) = \frac{1}{2}\left(d(x+1,y,t) - d(x-1,y,t)\right)$$
$$d_y(x,y,t) = \frac{1}{2}\left(d(x,y+1,t) - d(x,y-1,t)\right)$$
$$d_t(x,y,t) = \frac{1}{2}\left(d(x,y,t+1) = d(x,y,t-1)\right)$$

2.38

This is solved as follows:

The optimal updates $\delta D, \delta D_x, \delta D_y, \delta D_t$ are given by

$$H\begin{bmatrix}\delta D\\ \delta D_x\\ \delta D_y\\ \delta D_t\end{bmatrix} = -b$$

2.39

Where H is the Gauss-Newton approximation to the Hessian, $-b$ is the gradient and D is the concatenation of d(x,y,t) for every x,y, t into a column vector and similarly so for D_x, D_y, D_t. The linear constraints specified in Equation 2.38 are specified by matrix multiplications as follows:

$$D_x = G_x D \qquad D_y = G_y D \qquad D_t = G_t D$$

2.40

Where G_x, G_y, G_t are the sparse matrix encoding of the finite difference operator. Substituting Equation 2.38 into Equation 2.39 produces the following:

$$\begin{bmatrix}I\\ G_x\\ G_y\\ G_t\end{bmatrix}^T H \begin{bmatrix}I\\ G_x\\ G_y\\ G_t\end{bmatrix} \delta D = -\begin{bmatrix}I\\ G_x\\ G_y\\ G_t\end{bmatrix}^T b$$

2.41

Equation 2.41 is then solved iteratively using the conjugate gradient [75] method. This algorithm is very powerful and produces excellent results that can almost rival laser scanning, while maintaining the benefit of being able to sample deformable objects. The drawback of this algorithm is the intensive computational burden it places on today's available hardware.

2.2.6 Stereo Reconstruction on GPUs

This thesis will demonstrate how powerful GPUs can be in the context of stereo reconstruction. For this reason it was important to summarise some of the advances made in the field with regards to solving the reconstruction problem on GPUs. Gong and Yang [27], Wang et al.[85], Moslah et al.[55] and Sin et al.[77] all use dynamic programming to some extent to solve the correspondence problem. All these instances use a hybrid-approach where the cost function is implemented on the GPU while the optimization step is performed on the CPU. Gong and Yang [27] however, implemented two versions, one that solely runs on the GPU while the other uses both CPU and GPU, they found that even though their hybrid approach suffered from the severe limitations of the AGP bus, they achieve superior performance using it. Other examples of reconstructions performed on GPUs include Chang et al.[14] which uses a multi view approach combined with graph cuts to reconstruct surfels, as well as Yang et al.[87] and Zach et al.[88].

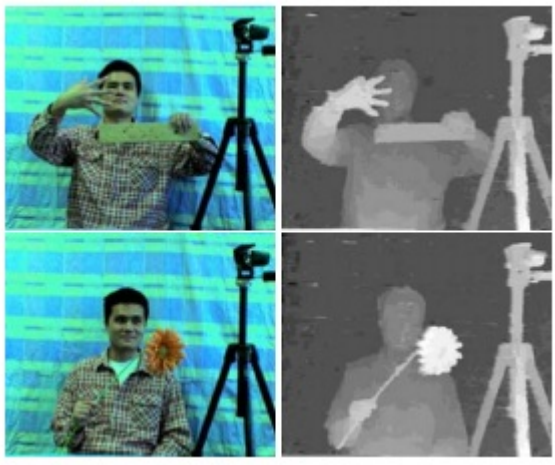

Figure 2.9 Taken from Wang et al.[85] illustrating their stereo results

2.3 Structured Light Surface Capturing

Another method of acquiring the 3D structure of surfaces is with the use of structured light. This method illuminates its environment and therefore falls in the category of active systems. By projecting a pattern of light onto a surface it becomes distorted. Measuring this distortion enables these systems to infer the geometric properties of the underlying surface as illustrated in Figure 2.10. A projector can be conceptually interpreted as the inverse of a camera. Cameras function by taking a ray of light and projecting it into a pixel, whereas projectors to do the opposite of projecting a pixel into a ray of light. This fact enables a projector to be modelled with the same parameters as a camera. By doing so, some of the same fundamental concepts behind stereo photogrammetry presented in the previous section can be applied, and one can infer the depth of imaged pixels. Where stereo systems use two cameras, structured light systems replace the second camera with a projector. The section will briefly describe two types of structured light systems, the first being systems based on coded structured light while the second will focus on systems that use phase shifting, it is also worth noting that these two methods can also be combined to create hybrid systems.

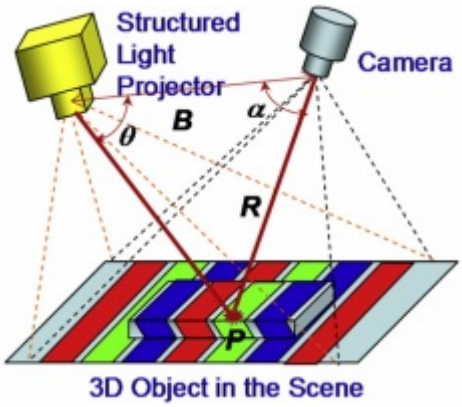

Figure 2.10 Taken from [26] illustrating the distortion created from surfaces on structured light patterns

The projected patterns used by such systems all have a specific structure so that a set of pixels is easily distinguished by means of a local coding strategy. Therefore locating such points in the captured image solves the correspondence problem. These systems trade the ease of solving the correspondence problem inherently difficult in stereo systems against the added difficulty of calibration, as well as requiring the use of

multiple patterns. Such calibration techniques are proposed by Chen and Kak [15], Zhang and Huang [94] and Park et al.[65].

2.3.1 Coded Structure Light

Coded structured light systems function by projecting a series of patterns that allow a set of pixels to be easily identified by means of local coding strategy. The 3D shape is then calculated from the decoded pixels by means of triangulation. The most commonly used patterns contain stripes as they are easily distinguished. The key to designing a good system lies in using a coding strategy that enables accurate localisation of these stripes. The patterns can be combined over time and projected onto a surface sequentially creating a unique code word for each imaged pixel, therefore making the localisation trivial. One example of such a pattern presented by Inokuchi et al.[37] uses a binary Gray code. These Gray codes are resilient to errors since only one bit changes at a time however using Gray codes requires $\log_2(n)$ patterns to localise n points. This can be a substantial number if a high special resolution is required, and doing so makes the system only useful for capturing static scenes. More recent studies of binary codes have been published by Rocchini et al.[69], Skocaj and Leonardis [79] and Furukawa and Kawasaki [23].

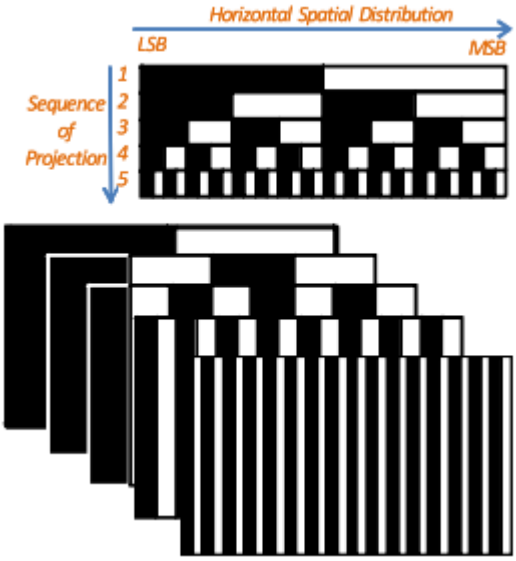

Figure 2.11 Example of Gray code pattern taken from [26]

Using a binary code requires a large number of images. By taking advantage of the grey scale resolution of projectors, Horn and Kiryati [36] present a method for grey level code selection by combining Gray code and intensity ratio techniques. Alternatives for

using coded light while reducing the number of patterns, is to use colour information (Wang et al.[84], Caspi et al.[13]). Hall-Holt and Rusinkiewicz [30] use space and temporal coherence. They define a binary coded pattern capable of being used to capture moving scenes. A complete survey of coded structured light techniques is published by Salvi et al.[71].

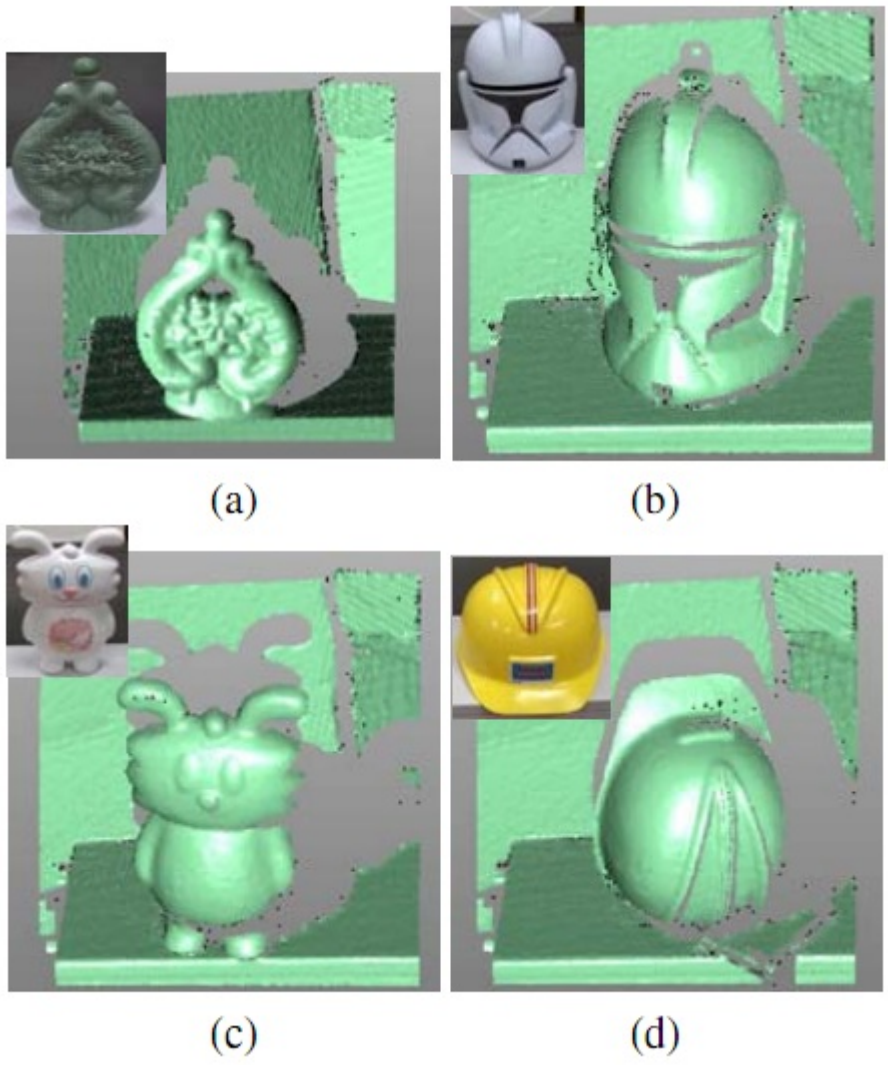

Figure 2.12 Example taken from Furukawa and Kawasaki [23] depicting reconstruction from coded structure light

2.3.2 Phase Shifting

Phase shifting methods work by projecting periodic patterns onto the surface. The surface geometry will warp and distort these patterns. One well known method relies on

projecting three sinusoidal patterns. The intensity of each pixel (x,y) of the three patterns are described as:

$$I_1(x,y) = I_0(x,y) + I_{mod}(x,y)\cos(\phi(x,y) - \theta),$$

2.42

Where I_1 (x,y), I_2 (x,y) and I_3 (x,y) are the intensities of each fringe pattern, I_0 (x,y) is the DC component (i.e. background), I_{mod} (x,y) is the modulation signal amplitude, ϕ (x,y) is the phase, θ is the constant phase shift angle.

Phase unwrapping is the process that converts the wrapped phase to the absolute phase. The phase information ϕ (x,y) can be retrieved (i.e., unwrapped) from the intensities in the three fringe patterns:

$$\phi' = \tan^{-1}\left(\sqrt{3}\frac{I_1(x,y) - I_3(x,y)}{2I_2(x,y) - I_1(x,y) - I_3(x,y)}\right)$$

2.43

The discontinuity of the arc tangent function at 2π can be removed by adding or subtracting multiples of 2π on the ϕ' (x,y):

$$\phi(x,y) = \phi'(x,y) + 2\pi k$$

2.44

Where k is an integer representing projection period. The 3D (x,y,z) coordinates are calculated based on the difference between measured phase $\phi(x,y)$ and the phase value from a reference plane.

$$Z = \frac{L - Z}{B}d$$

2.45

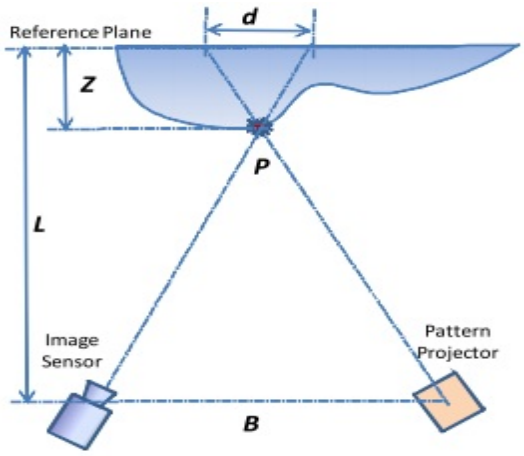

Figure 2.13 Taken from [26] shows relationship between phase difference and surface depth

Zhang and Huang [93] used a variation of this technique to produce a real time face scanner with quite compelling results, as show in Figure 2.14. The advantage of phase shift methods are that they are fast and require fewer structured patterns than the coded methods to produce accurate results. The down side of these methods is that they assume a continuous surface, and do not handle great changes in depth due to phase wrapping, they also perform badly on surfaces that have a steep gradient.

Figure 2.14 Taken from Zhang and Huang [93]

2.4 Tele-Immersion

With advances in networking, virtual reality and computer vision, researchers are looking into new innovative ways of increasing social presence in tele-immersive environments. Tele-immersion seen as the natural evolution to video conferencing, is based on displaying stereo displays of remote participant users in a shared virtual space. The eventual ideal goal is to make the physical presence of individuals irrelevant by

creating a space that is part virtual part physical that all the users can interact in with incredible fidelity.

Excellent work has been carried out by various institutions, notably by the University of North Carolina and the University of Pennsylvania, with their collaborative work on “The Office of The Future project” [28]. Other notable efforts include [46]. They point out the various limitations of tele-conferencing systems, where users lack perception of depth, as well as offer limitations such as eye gaze. In tele-conferencing systems users are rarely looking directly at each other. This is due to the fact that the cameras are often offset with regard to the display. This problem can be overcome with the use of a half-silvered mirror, which is a screen made out of glass that has the property of reflecting as well as transmitting light. These screens can be arranged in such a way that allows the camera to be placed directly behind the centre of projection, thereby creating the appearance of direct eye contact between the users. There are still further limitations to consider, one of them being motion parallax. Users of these systems do not experience motion parallax (i.e. when a user moves their heads they do not see the other user from a different point of view). One possible solution is to build a 3D model of the users from video feeds as well as track their head movements in order to correctly display different viewpoints.

Most real-time 3D capturing systems used for tele-immersion fall into three categories (1) silhouette base methods [29] [4], (2) voxel based methods [33], and (3) dense stereo reconstruction methods [40] [41].

In [56] a tele-immersive system is described. This system uses 7 digital firewire cameras. These cameras are used to create different combinations of trifocal stereo pairs that are used to perform dense depth estimations. However, the computational burden placed on such a system is great, and therefore certain constraints were imposed on the choice of stereo algorithm, as well as error metric used for the correlations. As suggested in [56], using optical flow to further constrain the problem and improve performance of their algorithms. Please refer to Section 2.4.3 for an overview of stereo algorithms.

Figure 2.15 Results achieved from system described in [56]

2.4.1 Hybrid Stereo-Image Based Rendering

3D reconstruction can be very expensive and as a result, in [45] Kurashima et al.opted for taking a hybrid approach inspired by image based rendering techniques. The approach mentioned can be broken down into two main steps. In the first step they find a sparse proxy geometry of the user, which is then combined in the second step with view dependent texturing.

Step 1 Build proxy geometry

Initially the foreground object is segmented from the background, the best plane that fits the foreground is then found. This is achieved by finding the approximate silhouette of the foreground object. Having calibrated and rectified a stereo pair, the left most foreground pixel in the left image is matched to the left most foreground pixel in the right image. This action is performed for the right most foreground pixel as well, and is repeated for each scan-line. This produces an approximation to the silhouette. Problems do occur however with foreground boundaries that are almost parallel to the epipolar lines, to remove outliers a plane is fitted using a least squares method, then the mean

and standard deviation of the distances to the plane are found. Any points whose distance to the plane is greater than 3σ are removed, and the plane is refit.

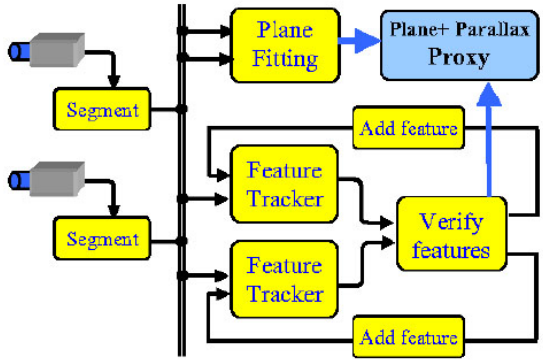

Figure 2.16 Block diagram taken from [45] outlining step 1

This is performed until convergence, or until a pre-determined number of iterations have been performed. This process is then followed by a feature tracker such as the KLT [7] on the foreground, constrained by the epipolar geometry. These features are then used to find the offsets from the fitted plane. The result is then triangulated, producing a very approximate geometry used in the view dependent texturing stage.

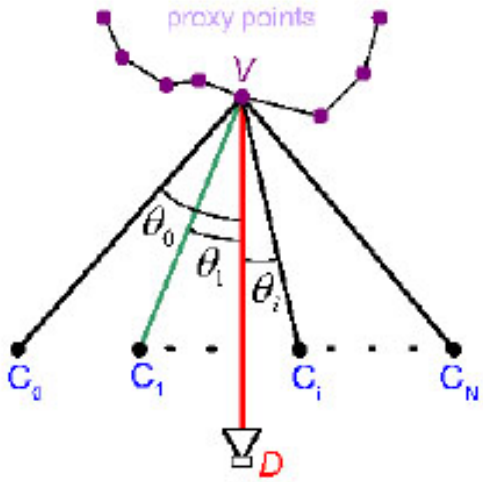

Figure 2.17 From [45] showing angles between Cameras and the desired synthesised view

Step 2 View dependent texture map

In this stage the proxy geometry is texture mapped by all the images from each camera. Each vertex in the proxy geometry is assigned a weight for each camera. This weight is

then used to blend pixels from different cameras into the resulting texture. The weights are calculated as follows.

The angle θ_i between the camera C_i and the desired view D is calculated. The computation of the blending weight w_i is given by the following equations.

$$\widehat{w}_i = \exp\left(\frac{-\theta_i^2}{2 \cdot \sigma^2}\right)$$

$$w_i = \frac{\widehat{w}_i}{\sum_{j=0}^{N-1} \widehat{w}_j}$$

2.46

Where σ is a constant value for the limit maximum angle, if the angle exceeds this value it is assigned a weight of zero.

2.5 Conclusion of Literature Review

As the literature review has demonstrated using stereo cameras with structured light is a very powerful tool for producing 3D models. However, there still seems to be a wide gap between algorithms that can be performed in real-time and offline algorithms, when it comes to output quality. The advantages stereo algorithms have over purely structured light systems, are as follows: They can handle greater depth variations, they tend to produce superior results with oblique surfaces, they can be made to work with or without structured light, they solve the correspondence at every frame even when using a space time window and therefore can potentially capture at higher frame rates. All of these advantages are a compelling reason to use the stereo approach. However these advantages come at the cost of having to solve a more difficult correspondence problem. The real-time solutions of the past tend to produce less compelling results when compared to the much more computationally expensive iterative methods.

The motivation behind the research carried out in this report is to try and bridge this gap. What are the compromises currently being made with regards to real-time stereo systems, and are they the correct ones? Are there alternative compromises possible to achieve the goals of real-time stereo in the context of tele-immersion?

Chapter 3

Dynamic Programming and Structured Light

The objective of this chapter is to determine the suitability of various dynamic programming algorithms for stereo reconstruction with the target application of tele-immersion. These types of algorithms were chosen because of their computational performance, the fact that they treat scan-lines individually and therefore can potentially be made to run in parallel with relative ease, and finally because they also contain mechanisms to handle occlusions. Bearing in mind these motivations the following questions were posed:

- Are these algorithms suitable for active stereo?
- What is the impact of different structured light patterns on these algorithms?
- Are there any benefits in using a space-time support region for the dissimilarity functions?
- Which algorithm performs best in the following scenarios: with and without structured light, with and without a space-time window?
- What is the most suitable cost function for different algorithms and patterns?
- Given that these algorithms implicitly assume fronto-parallel surfaces, how do they perform when reconstructing a head shape which violates these assumptions?

This chapter will describe the contributions made with regard to extending, developing and analysing stereo systems benefiting from structured light while using a variety of different DP algorithms reviewed in Section 2. The reasoning behind choosing DP algorithms was that they are relatively computationally cheap, and have proven to be applicable to real-time systems. They also contain some mechanism for handling occlusions and because they optimize each individual scan-line separately, they can be

easily adapted to run on parallel architectures. However, this comes at a cost of introducing certain inconsistencies and creating stripping artefacts due to the fact they optimize each scan-line individually. This has led to the further development of two pass dynamic programming algorithms [62] that introduce inter scan-line consistency constraints in an attempt to eliminate these artefacts. By using structured light it will be shown that these stripping artefacts are no longer an issue.

However, these algorithms have other limitations. One of these is the assumption of the monoticity constraint as described in Chapter 2. It will be shown for the purpose of reconstructing a human head or generally convex objects, that this is not a concern. Dynamic programming algorithms also have the advantage over the winner-take-all type (WTA) and scan-line optimization (SLO) algorithms by having a mechanism that deals with occlusions. Another one of their limitations is caused by their use of symmetric rectangular support regions or windows when computing the cost function. This implies fronto-parallel surfaces and is indeed violated when reconstructing objects such as the human head.

These DP algorithms were initially developed and tested on passive scenes, not artificially illuminated ones, using structured light. It was therefore necessary to evaluate these algorithms in the context of a capturing system using structured light and that motivated the contributions made in this chapter. The main purpose was to discover the implication of using structured light with the following dynamic programming algorithms Criminisi et al.[19], Birchefield et al.[8] and traditional DP, which are all described in Chapter 2.

In order to evaluate and determine the suitability of these algorithms I pose the following questions:

How well are these algorithms suited to solving the stereo correspondence problem?

Does using structured light improve their performance?

What are the implications of choosing different cost functions with regard to different illumination patterns?

Can these algorithms be improved by extending the support region of their cost function into the time domain?

Are these algorithms suitable for real-time reconstruction of the human face?

To answer these questions this section is broken down into the following sub-sections.

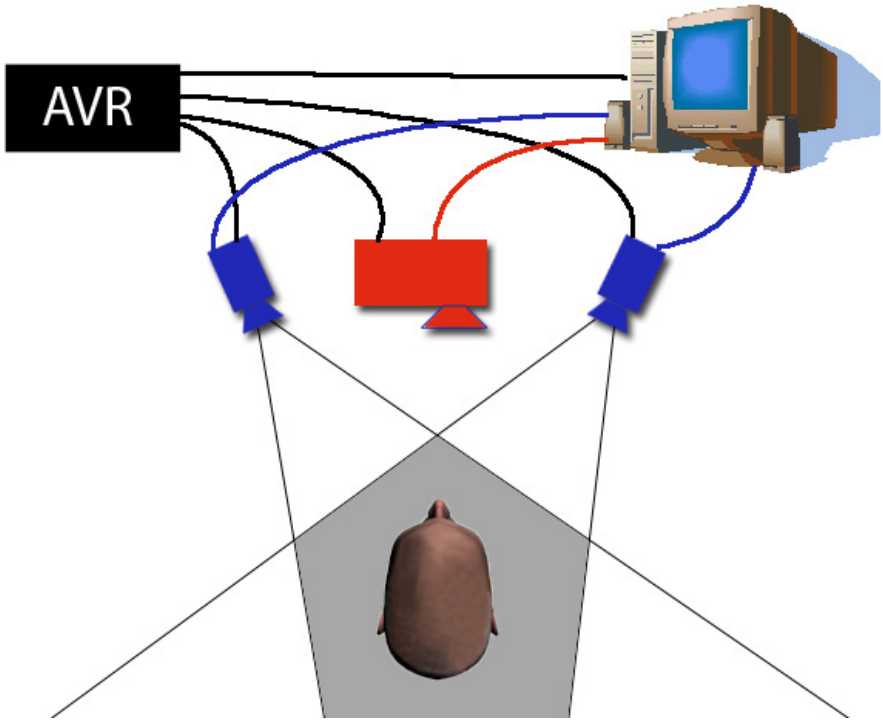

Figure 3.1 Illustration of capturing system

Section 3.1 will describe a capturing system consisting of two stereo cameras and projector capable of projecting various structured light patterns. The light patterns used were temporally varying and it was therefore essential to synchronize the cameras to the projector. This synchronisation was achieved with a circuit using an Amtel AVR microprocessor [2] development board. In order to evaluate these various algorithms and their extensions into the space-time domain as well as various other modifications described, a true disparity map would be necessary for comparison. Due to the difficulties in obtaining such a map, a simulation was developed whose details will be given in Section 3.2. The practical implications of using structured light as well as the implementation details of the various algorithms are described in Section 3.3. The experiments carried out will be reviewed in Section 3.4 as well as their results in Section 3.5. The conclusions will then be summarised in the final section of this chapter. In essence this chapter will explain and discuss the contributions carried out in [57] and [58].

3.1 Experimental Setup and Sample Acquisition

In order to acquire image samples two Balser A602fc cameras were used. These cameras were chosen because they are capable of very high capture speeds (100fps) while their resolution is adequate (640x480). They also use IEEE 1384 interfaces which tends to be more reliable than USB and are capable of high bandwidth (400 mb/s). These cameras also possess the feature of being able to be triggered via their own propriety interface. The structured light patterns were projected using a BenQ DLP projector capable of running at 800x600 @ 60Hz. Ideally one would use infra-red light as it is not perceivable to the human eye and is less obtrusive. One should note that the DLP projector works using a CMOS IC consisting of an array of micro reflective mirrors that can be switched electronically from the projector circuitry. This array represents each pixel and reflects a light source through the projector lens. The intensity of each pixel is determined by the amount of time the micro mirror is reflecting light, while the pixel colour is produced by a rotating disc containing three filters, one for each red green and blue colour component. This disc is placed in between the projectors light source and the array of micro mirrors/ By synchronizing this rotating disc with the switching of the mirror array, the projector is able to project pixels of varying intensity and colour spectrums. This design makes it convenient to modify a DLP projector by replacing the light source with one capable of emitting infrared light and switching the colour filters with filters that only allow infra-red light to pass (such as a piece of exposed film). The outcome would be a fully functional infra-red projector capable of projecting different infra-red light patterns of varying intensity as demonstrated in [17] and [90]. For the purpose of evaluating and extending the algorithm to solve the stereo correspondence problem, as presented in this and following sections, it was deemed unnecessary to make such modifications.

To appropriately use dynamically changing structured light patterns with two cameras for stereo reconstruction, the cameras were synchronized with each other, and in order for the cameras to capture one unique structured light pattern they were synchronised to the projector. The synchronization was achieved with custom electronics specifically developed for this purpose. A circuit was developed that would take as input the VGA horizontal sync signal as well as the two camera trigger ready signals. These were regulated to match the specification of the AVR microprocessor and fed into the AVR input channels. Then a relatively simple program was run on the AVR

that checks the horizontal synch and make sure the cameras were ready to be triggered and if so would send an output trigger to both cameras.

This setup enabled the acquisition of stereo pair images that both contained the same structured light pattern and therefore eliminated the horizontal synch mismatch artefacts produced when the cameras were not synchronised. This is crucial if using a structured light pattern that is changing rapidly. Another option would be to synchronize the cameras in software. However, this has many limitations, and in order to produce results with the lowest possible latency, rendering the hardware options are being the only practical one.

3.2 Simulation of Capturing System

The previously described capturing system was also simulated in order to produce synthetic data with a known true disparity that could then be used to evaluate the algorithms and their respective performance. This simulation was done in Autodesk 3D Studio Max [34]. A scene was created containing a 3D head with the same structured light pattern applied as a projective texture. The scene was then rendered from the two different viewpoints representing the left and right cameras, a number of times with and without a variety of different structured light patterns. A calibration object was also rendered in the synthetic setup, which allowed the use of the same calibration toolkit with the simulated images, than that used with the real sample images. From the synthetic images it was possible to render the exact depth map for each camera view. This precise synthetic depth map was then used for comparisons with the ones produced by the different algorithms.

The synthetic data is only an approximation of the capturing system, as the real system contains acquisition noise and lens distortion that was not simulated. This data did prove useful in providing a benchmark for the various dynamic programming algorithms. The following figures illustrate some of the samples obtained with the real synchronised cameras as well as simulated synthetic images. Figure 3.2 illustrates some of the synthetic images produced. However as shall be described in Section 3.5, these images do not simulate the level of noise commonly found with real images and therefore they are not perfect for evaluating these algorithms.

Figure 3.2 Synthetic scene rendered from left camera using a Gray code and noise pattern.

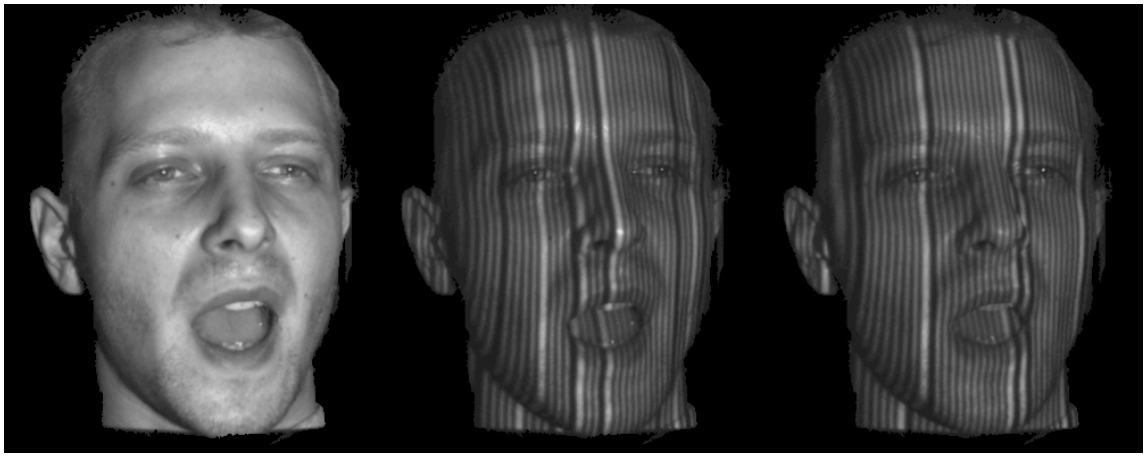

Figure 3.3 Real images acquired from left camera with and without Gray code pattern

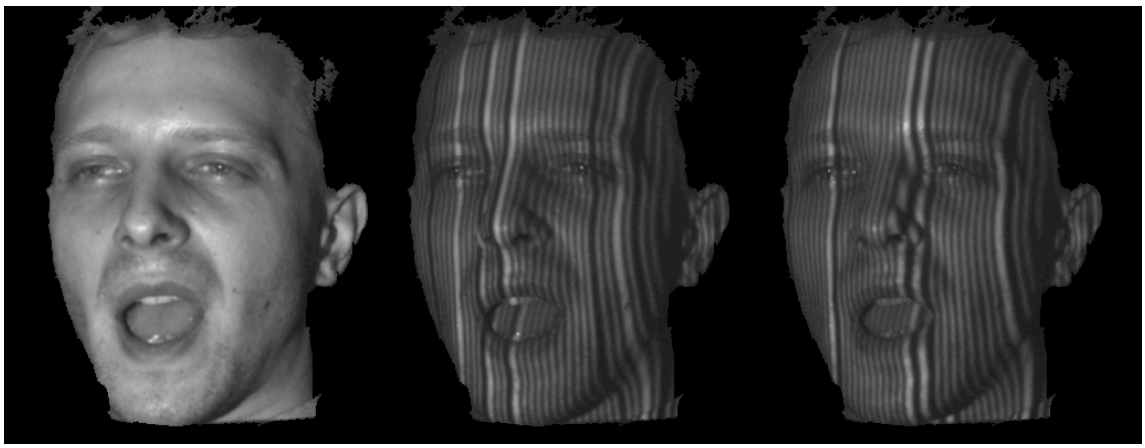

Figure 3.4 Real images acquired from right camera with and without Gray code pattern

3.3 Implementation

This section will cover the implementations of the stated techniques as well as the variations that had to be made in order for the algorithms to work with structured light and be extended into the time domain. A possible framework for space-time stereo using structured light described in [91] motivated the extension of the implemented algorithms into the space-time domain. However the space-time support windows were

not sheared and skewed. The results of these various implementations will be presented in Section 3.5.

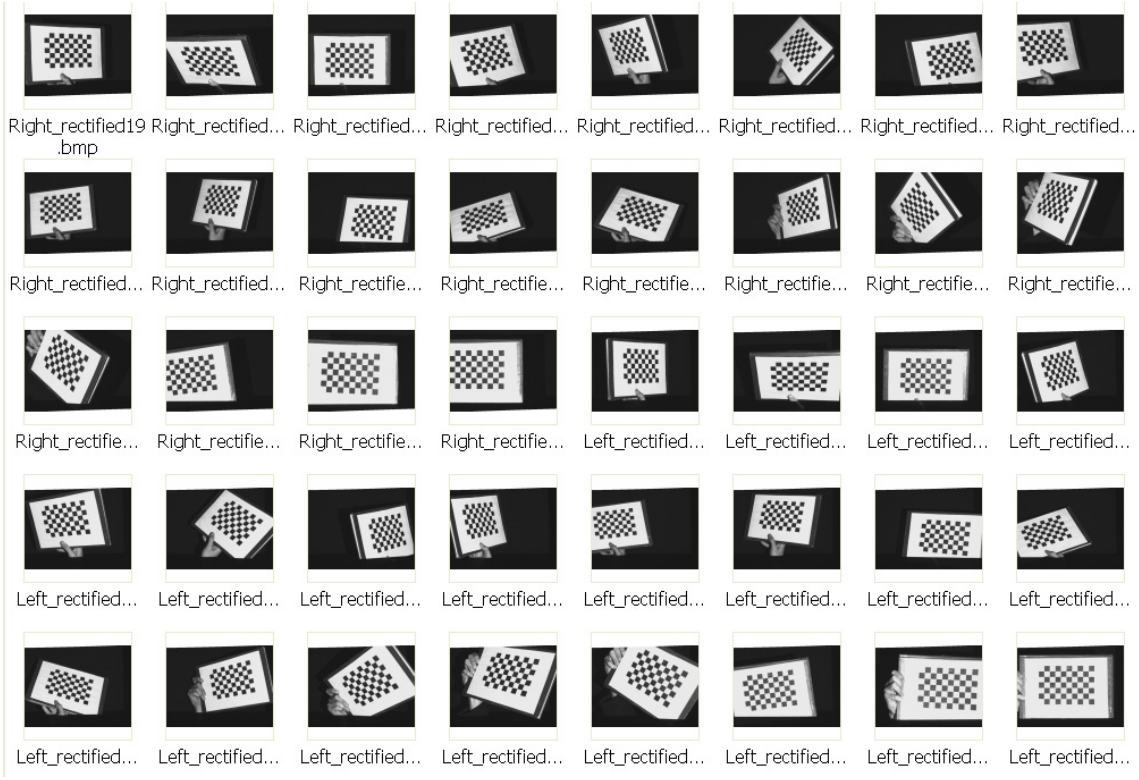

Figure 3.5 Calibration image samples

3.3.1 Camera Calibration

Section 2.2 demonstrated that the stereo correspondence problem can be reduced to a 1D search problem using the epipolar constraint. This was achieved by firstly calibrating the stereo cameras. This calibration process only needs to be completed once, as long as the position of the cameras stay fixed relative to each other. The outcome is to discover the camera's intrinsic and extrinsic parameters. In the case of a simple pinhole camera this transformation from 3D world coordinates to 2D image space coordinates can be described by Equation 3.1 as follows

$$s\cdot\tilde{m} = K[R\ t]\tilde{M} \tag{3.1}$$

Where a 2D image point is denoted by $\tilde{m} = [u, v, 1]^T$. A 3D world point is denoted by $\tilde{M} = [X, Y, Z, 1]^T$, s is an arbitrary scaling value while (R, t) is a rotation and translation

from world coordinates to camera coordinates and usually referred as the extrinsic parameters. The matrix *K* defined in Equation 3.2 and refers to the camera's intrinsic parameters that define focal length, centre of projection and skew.

$$K = \begin{bmatrix} \propto & c & u_0 \\ 0 & \beta & v_0 \\ 0 & 0 & 1 \end{bmatrix}$$

3.2

Solving the parameters from Equations 3.1 and 3.2 was achieved by solving an over determined linear system formed by a known set of corresponding point in the 3D world coordinates and the 2D image space. This was achieved by imaging a calibration object, in this instance, the commonly used chequered board pattern was used as illustrated in Figure 3.5.

The Equation 3.1 is transformed into the linear system (Equation 3.3) by Equation 3.2 and can be solved using SVD.

$$A \cdot \bar{x} = \bar{b}$$

3.3

Where $\bar{x}$ represents the vector of parameters.

This method will usually perform poorly, while at the same time care must be taken to normalize the data (i.e. the coordinates of feature points). Alternative methods that tend to produce superior results often rely on non-linear optimization methods while also augmenting the camera model to support forms of radial distortion. These methods usually start off by solving a linear system and use these results as an initialisation step for more robust maximum likelihood solvers usually based on the Levenberg Marquardt method. For the experiments presented throughout this thesis a Matlab toolkit developed by Bouguet et al.[10] which was loosely based on an algorithm created by Zhang et al. [91] was employed, which uses such non-linear optimization techniques.

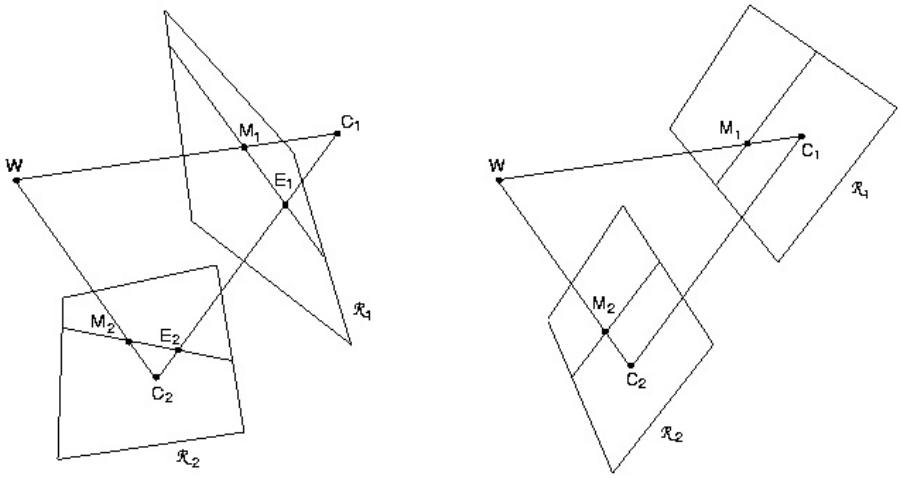

Figure 3.6 Image rectification

Multiple images of the calibration objects positioned at different orientations were taken from both cameras. The resulting two sets of images (Figure 3.5) were then used for calibration, which was performed in two stages, the first being calculating the camera's intrinsic parameters or projection matrix, then followed by computing the extrinsic parameter defined as rotation and translation transformation relating the coordinate frame of one camera to the other.

Once the cameras are calibrated and their parameters are known, two rectifying homographies can be computed. Each homography is a projective transformation that warps the image of each respective camera so that each scanline of the image becomes parallel to the epipolar lines of the stereo rig. This is equivalent to re-projecting the images of both cameras onto the same plane with the added constraint of each set of scan-lines from both cameras being matched to each other. This reduces the correspondence problem to a 1D search problem (i.e. each pixel in one cameras frame of reference will have its corresponding pixel in the other camera's frame on the same scan-line given no occlusions). The geometric interpretation of these two transformations is illustrated in Figure 3.6 and can be represented by a 3x3 transformation matrix. There are many solutions to the rectifying homographies. Ideally one would find a solution that also minimizes the amount of image distortion in both cameras. One such technique is presented by Zhang et al.[95]. Another simpler and also effective technique is presented by Trucco et al.in [24]. For the purpose of the experiments presented in this thesis the Bouguet et al.[10] Matlab toolkit was also used

to compute the rectifying transformations that could then be trivially applied as a projective texture. The difference between the captured and rectified images of both cameras is illustrated in Figure 3.7.

Figure 3.7 Top row captured images, bottom row rectified images

	stripe intensities															
t=0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
t=1	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0
t=2	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0
t=3	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0

(a) Gray code

	stripe intensities															
t=0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
t=1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1
t=2	0	1	0	1	1	0	1	0	0	1	0	1	1	0	1	0
t=3	1	0	1	0	1	0	1	0	0	1	0	1	0	1	0	1

(b) Modified Gray code

Figure 3.8 Illustrates the initial Gray code (a) that was subsequently shuffled in the space to domain to produce the stripe pattern (b) used in the structure light reconstructions

3.3.2 Structured Light

The creation of the structured light patterns was achieved with two sets of patterns containing eight frames each. The first set was generated as described in [91] using a 16bit reflected Gray code [16] that was subsequently shuffled to produce high frequency change in both the spatial and temporal domains as represented in Figure 3.8. This produced eight frames containing stripes four pixels wide that were then smoothed with a Gaussian filter. The properties of this pattern are that it has high frequency changes in both the temporal and space domain and therefore maximizes the entropy across the space time support region. This helps reduce ambiguity when solving the correspondence problem. These properties were also achieved with a second set of patterns, by approximating white noise using an out of tune TV (approximation of white noise) signal followed by a low pass filter. These patterns were pre-computed offline and then stored as OpenGL [76] textures and applied to a textured quad.

These patterns were subsequently projected using an Infocus DLP projector at a resolution of 800x600 and cycled at the camera capture rate. The projector could have been driven by the second VGA output of the capturing machine. A superior alternative was to use a secondary computer to drive the projector, thereby conserving the computational resources of the data acquisition and reconstruction machine. The projected patterns are illustrated in Figure 3.9.

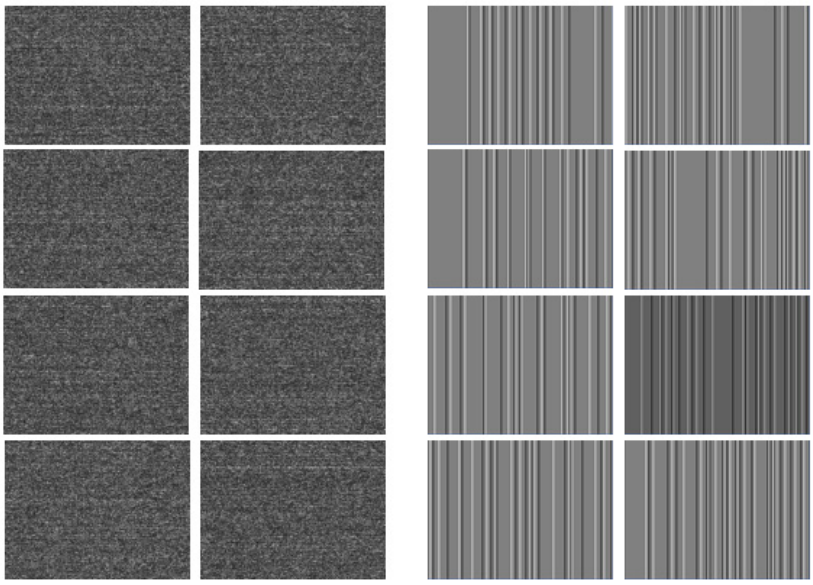

Figure 3.9 Left column 8 noise patterns, right column 8 shuffled stripe patterns

3.3.3 Dynamic Programming CPU

Initially, three core DP algorithms along with their extensions into the time domain and the use of different sets of cost functions were evaluated for their qualitative results. This did not require optimal performance in terms of computational speed and therefore un-optimized CPU implementations were used. A naïve DP algorithm described in Chapter 2 was implemented. The OpenCV [12] implementation was used for the Birchfield et al. [8] DP algorithm and modified to support various cost functions and extended into the time domain. After experimentation with the OpenCV implementation it was found that the structured light patterns broke some of the assumptions made in the post-processing steps of Birchfield et al. [8] described in Section 2.3. The striped structured light pattern removes intensity variations along the y-axis and therefore one of the stopping criteria for the region growing of the disparities along the y-axis is violated. These post processing steps were subsequently removed.

The Criminisi et al. [19] DP algorithm was implemented from scratch without using Gaussian smoothing of the dissimilarity matrix. The equivalent can be achieved in image space by blurring the images with a Gaussian kernel or defocusing the lenses of the cameras slightly. Support for SSD dissimilarity measure and normalized cross correlation over spatial and temporal windows was added to both the Birchfield et al.[8] and Criminisi et al.[19] implementations. The implementation of these algorithms was

not optimised, and a lot of redundant calculations were performed in order to conserve memory. Thus the performance was not real-time in some cases.

It should be noted that all the algorithms presented in this paper can be made to run in real-time on high-end workstations with the use of optimisation techniques, in combination with the SIMD instruction set. One such technique that is presented in Section 6.1 is to leverage the power of GPUs and multi core processors to create a hybrid implementation. This allows the cost function part of the computations to be performed on one or more GPUs while the multiple threads each running on its own processor core to compute the optimization part. Further optimizations to the Criminisi et al.[19] algorithm were explored and are presented in Section 6.2.

These algorithms were selected because of their relatively low computational complexity when compared to other stereo algorithms and the focus was to determine their suitability for use with structured light and real-time applications. Finally, the best disparity map created by the Criminisi et al.algorithm using stripe pattern with the SSD cost function, was used to create 3D surface using two different triangulation algorithms. Figure 3.12 shows a surface reconstructed from the true disparity values of the synthetic data. Figure 3.18 was created using Delaunay et al.[21] triangulation taken from the depth map created using the Criminisi et al.[19] algorithm on the real images dataset.

3.4 Experiments

Given that the dynamic programming algorithms described in Section 2.3, and further developed in this chapter were not originally designed with structured light in mind, it was therefore necessary to evaluate their performance with the capturing system described in Section 3.1 and its simulation in Section 3.2 using the patterns described in Section 3.3.2. It was also necessary to determine if these patterns improved results and whether there was a correlation between the patterns used and the dissimilarity measure or cost function. Although the original designers of these dynamic programming algorithms chose a particular cost function, would alternatives be more suitable, and what impact would extending these functions across into the time domain have on the

results? These algorithms were also designed with real-time applications as a goal, however, extending them into the time domain would alter their computational burden.

The naïve DP algorithm described in Section 2.2 behaved so poorly that it was quickly dismissed and detailed experiments were carried out as described below only on the Criminisi et al.[19] and Birchfield et al.[8] algorithms. A polystyrene head was used for the experiments on real images shown in Figure 3.15 as well as images of a real face as shown in Figure 3.3. The synthetic images were created using 3D Studio Max as shown in Figure 3.2. Both algorithms were tested on real and synthetic images using both the stripe and noise patterns for structured light. The following summarizes the various cost functions tested with each algorithm.

The Birchfield et al. [8] algorithm was tested using the following cost functions:

- Birchfield Cost Space Domain
- Birchfield Cost Extended into Space-Time Domain
- SSD Space Domain
- SSD Space-Time Domain
- Cross Correlation Space Domain
- Cross Correlation Space-Time Domain

The Criminisi et al. [19] algorithm was tested using the following cost functions:

- SSD Space Domain
- SSD Space-Time Domain
- Cross Correlation Space Domain
- Cross Correlation Space-Time Domain

The results for real images were evaluated qualitatively by examining the disparity maps. The results for the synthetic images were evaluated by calculating the RMS (root mean square) error (measured in disparity values) between calculated disparities and true disparities, as well as the percentage of incorrectly (within error threshold) matching pixels against the true disparity values. These qualitative results will be examined and summarised in the following Section 3.5.

3.5 Qualitative Results

Table 3.1 shows the RMS between the synthetic images and the true disparity. Figure 3.13 shows the true disparity map for comparison. In all cases the Criminisi et al.[19] algorithm produces superior disparity values containing fewer errors.

Figure 3.12 shows a graph representing the ratio of pixels whose error is below a certain threshold. Again the results indicate that the algorithm of Criminisi et al.[19] produces more precise disparity estimates. One can also conclude that the choice of algorithm has a greater impact than the choice of cost function or structured light pattern. In most cases using structured light improves these algorithms. Extending the support region into the space-time domain can further enhance these results. Choosing the appropriate cost function is quite strongly dependent on the structured light pattern used in conjunction with the system. When using the stripe pattern the cross correlation cost function is not ideal. Much better results are obtained using a low pass filtered noise pattern in conjunction with cross correlation. However, these results are limited to synthetic scenes. Figure 3.13 shows the 3D reconstruction results based on the synthetic images using Criminisi et al. [19] algorithm while Figure 3.17 demonstrates the performance on real-images. Using these disparity maps, 3D models are produced as shown in Figures 3.15, 3.16, 3.18.

Algorithm Cost Function Light Window	RMS
BirchCrossNoise3x7x16	24.411
BirchCrossNoise3x7	24.3297
BirchCrossNolight3x7	25.9128
BirchCrossStripe3x7x8	25.3448
BirchCrossStripe3x7	25.2588
BirchCrossStripeNoise3x7x16	25.918
BirchInterpNoise8	32.5587
BirchInterpNoise	32.5394
BirchInterpNolight	25.9111
BirchInterpStripe8	26.8512
BirchInterpStripe	28.2637
BirchSSDNoise3x7x8	23.5116
BirchSSDNoise3x7	24.3733
BirchSSDNolight3x7	23.4193
BirchSSDStripe3x7x8	22.8373
BirchSSDStripe3x7	25.4086
BirchSSDStripeNoise3x7x16	23.7173
CrimCrossNoise3x7x8	12.5939
CrimCrossNoise5x7	14.6616
CrimCrossNoiseStripe3x7x8	12.8042
CrimCrossNolight5x7	12.8788
CrimCrossSSDTest3x7x8	11.6188
CrimCrossStripe3x7x8	12.5563
CrimCrossStripe5x7	18.3184
CrimSSDNoise3x7x8	11.6739
CrimSSDNoise3x7	12.2769
CrimSSDNoiseStripe3x7x16	12.3845
CrimSSDNolight3x7	13.207
CrimSSDStripe3x7x8	11.4531
CrimSSDStripe3x7	11.4322

Table 3.1 RMS of errors between all algorithm's disparity values and true disparities

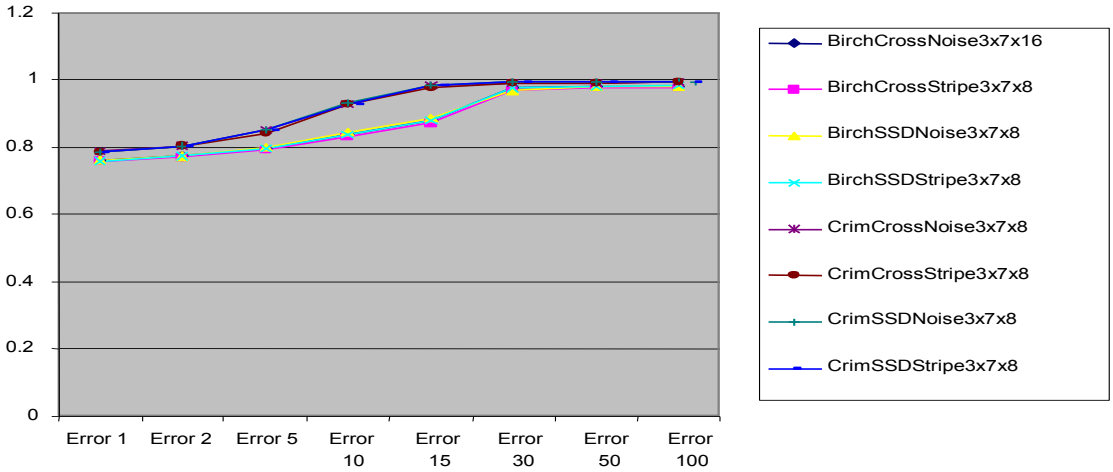

Figure 3.10 Ratio of pixels with error below threshold indicated in columns

Figure 3.12 represents a 3D surface created using the disparity estimates from the Criminisi et al. [19] algorithm with the cross correlation cost function applied to a support region of 3 *by* 7 *by* 8 pixels run on synthetic images containing a projected noise pattern. This can be compared to the equivalent created using the true disparity values. Figures 3.15, 3.16 and 3.18 show analogous results for real images.

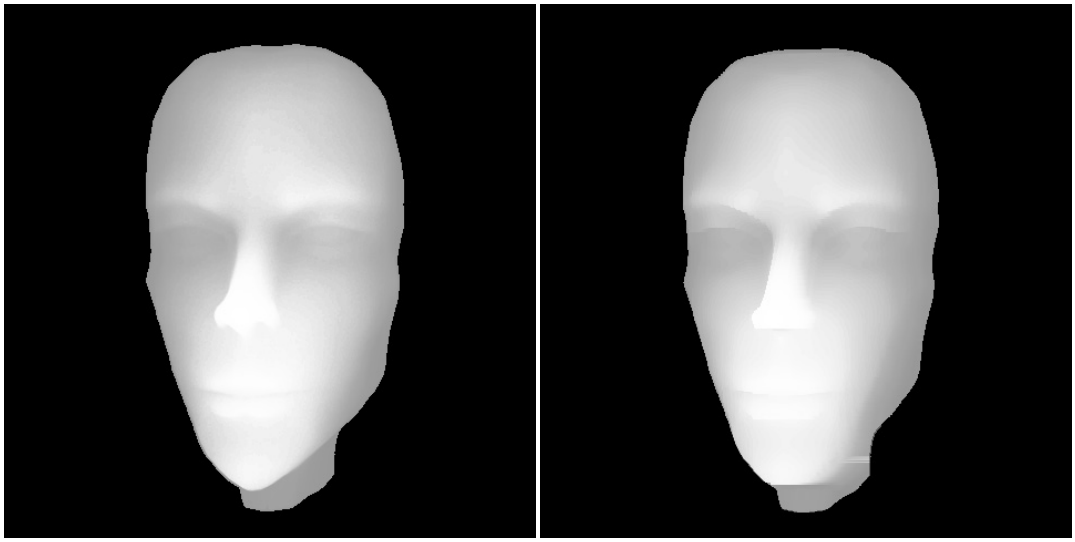

Figure 3.11 Left true disparity of synthetic head, Right result of Criminisi et al.[19] algorithm using cross correlation cost function over 3x5x8 space time window with noise pattern (synthetic data)

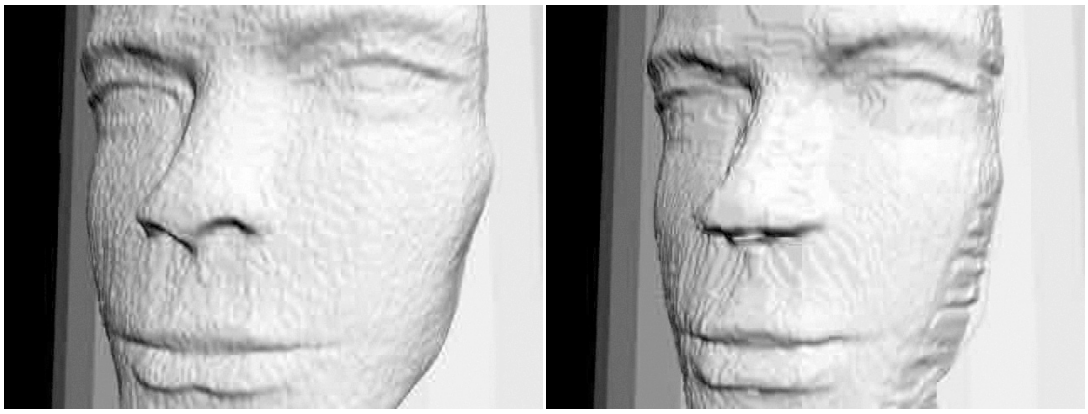

Figure 3.12 Delaunay Triangulation of true and computed disparities from Figure 3.12 (synthetic data)

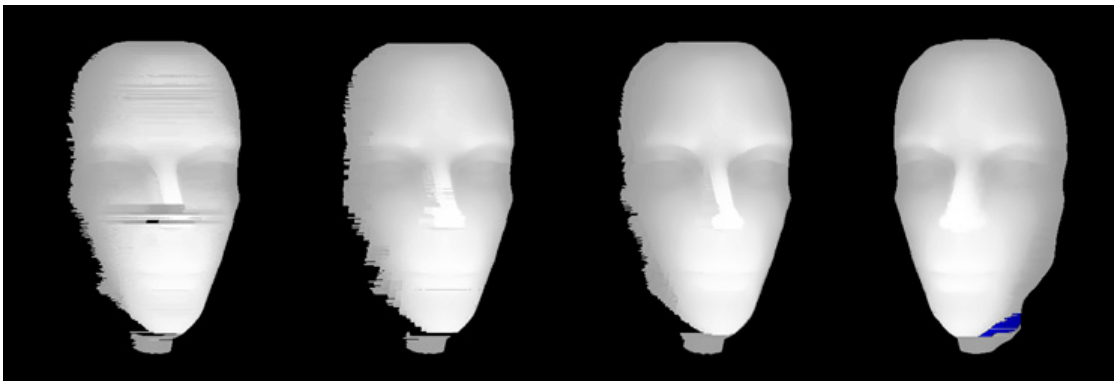

Figure 3.13 Comparison of Birchfield et al.and Criminisi et al.[19] algorithms (synthetic data)

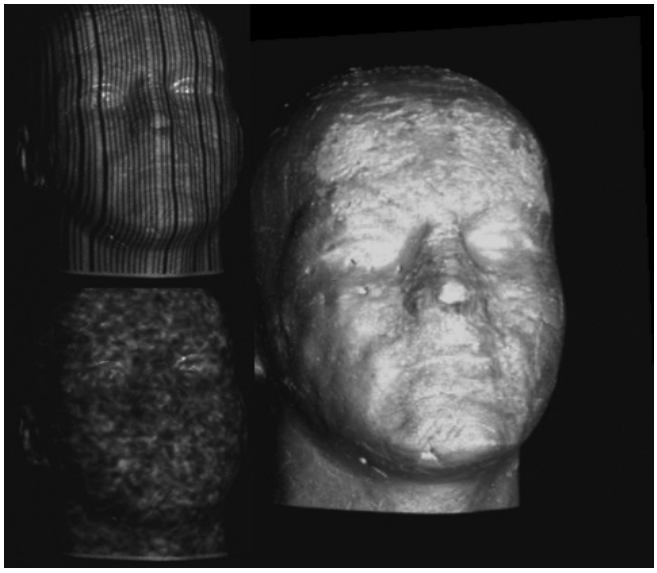

Figure 3.14 Sample of real captured images of polystyrene head with stripe and noise (real data)

Figure 3.15 Smooth Delaunay triangulation based on Criminisi et al..algorithm with a space-time window on images from Figure 3.16 (real data)

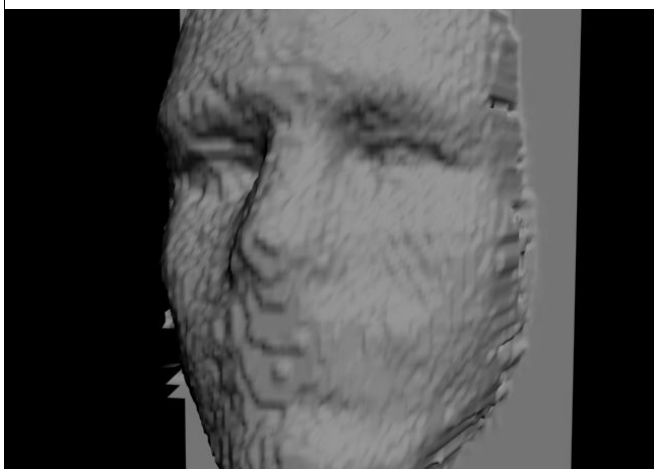

Figure 3.16 Non Smooth Delaunay triangulation (real data)

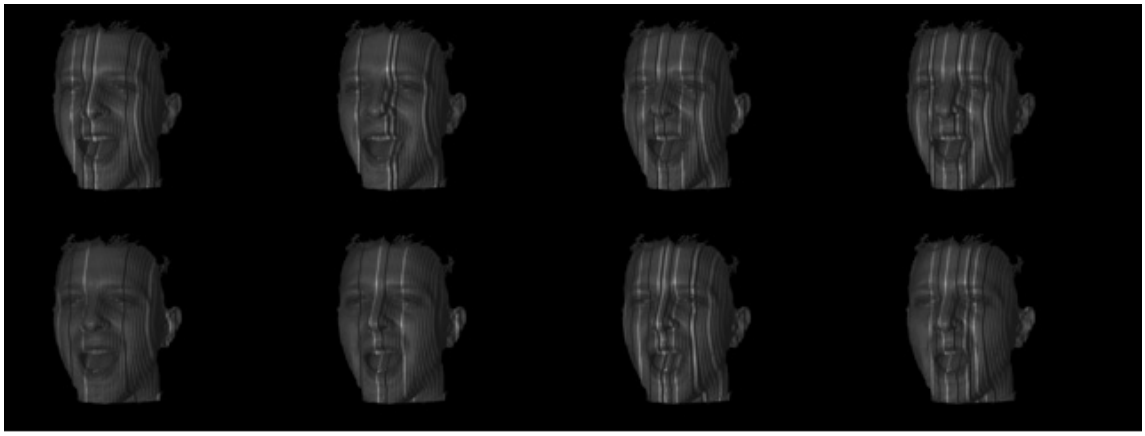

Figure 3.17 Sample of eight real captured images of a head using the striped pattern (real data)

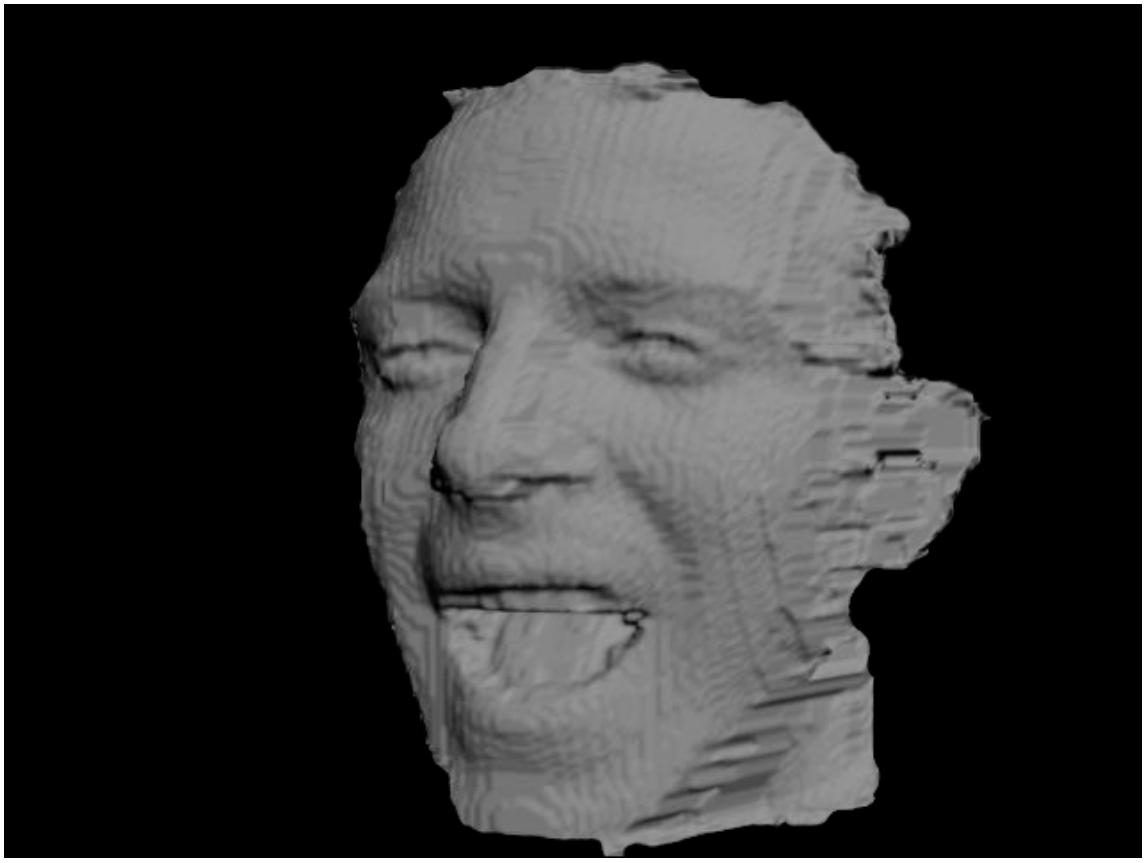

Figure 3.18 Delaunay triangulation based on Criminisi et al. [19] algorithm with SSD cost function on images from Figure 3.19 using a space time window using striped pattern (real data)

3.6 Conclusion

This chapter has shown that structured light can be a powerful tool for the improvement of the DP algorithms described in Section 2. The results show that the algorithm of Criminisi et al. [19] was superior for our test sets. We also find that when using the cross correlation cost function superior results are obtained with the structured light pattern generated from noise. In [58] it was found that generally the SSD cost function with the striped structured light performs better than the cross correlation cost function with the simulated input. However, these results were taken from synthetic images and cannot serve as an accurate measure of real world performance. After further experimentation with real images it became more apparent that the cross correlation cost function is more robust against noise. It has also been shown that using structured light eliminates some of the commonly associated stripping artefacts usually common to DP based algorithms.

Some of the benefits in using temporal information when evaluating dissimilarities between pixels have been highlighted. It has also be demonstrated that the superior dynamic programming algorithm developed by Criminisi et al.[19] and subsequently extended into the time domain in this section can be made to run in real-time using a hybrid CPU/GPU scalable implementation that is both parallel across CPUs and GPUs in Section 6.2.

One apparent limitation of structured light techniques is that they make it difficult to capture texture information. However, DLP projectors could be modified to project infrared light as suggested in Section 3.1. This would potentially allow a system to combine cameras with different filters and also capture texturing information, thereby eliminating the apparent limitation of using structured light. Another serious limitation is the fact that the disparities are calculated on a discreet level (i.e. disparities are measured in pixel values). Although this may be adequate for some applications, if one is using them to reconstruct 3D geometry this can be very limiting, as the range of values is very small and they produce 3D models that look somewhat blocky (see Figure 3.20).

The blocky artefacts can of course be removed by using a smoothing function such as parabola fitting however this results in a smooth mesh that is somewhat lacking in detail. One way of improving these DP algorithms is to use deformable support regions thereby eliminating the assumption of front-to-parallel surfaces. To achieve this,

the DP algorithms would have to become multi-pass and their computational complexity would increase substantially. Much better alternatives that combine the benefit of warped support regions with sub-pixel disparity estimates will be discussed in Chapter 4.

Chapter 4

Space-Time Stereo

The previous chapter demonstrated how dynamic programming can solve the stereo correspondence problem using structured light to produce pixel level disparity maps accurately in real-time. Although this per pixel map can further be refined by fitting a polynomial surface to the data, or more commonly by fitting some parabolic function, the results lack high frequency details. One alternative approach is to formulate the correspondence problem as a least squares non-linear optimization problem. This formulation was first proposed by Lucas and Kanade [51]. Their initial formulation targeted the stereo correspondence problem and optimized one parameter per pixel, namely the disparity along the epipolar lines or in the case of rectified images, a simple translation across the x axis of the stereo pair images. This method was then subsequently adapted to solve the optical flow problem by extending it to optimize the parameters of an affine warp. For a more detailed explanation of this extension readers are referred to Chapter 2 as well as [5]. Most of these developments target the optical flow and structure from motion problem as opposed to the stereo correspondence. Although the latter problem could be view as a subset of the former problems, there are certain subtle differences. The calibrated stereo correspondence problem contains certain constraints that make solving the pixel correspondence using a generic affine warp somewhat redundant. The general affine warp contains a translation in both x and y axes in image space. As shown from Section 3.3 by rectifying the stereo images it is no longer necessary to solve the disparity in the y axis. Another redundant affine warp parameter is the rotation, this would not improve the solution. However, introducing scaling and shearing properties to the support region would improve the solution. As opposed to having a symmetrical support region as originally proposed by Lucas and Kanade [51], which assumes forward facing parallel surfaces, scaling the support region along the x-axis would take into account the gradient of surface along the x axis and

shearing the support region somewhat compensates for the gradient along the y axis. This is visualised in Figure 4.1.

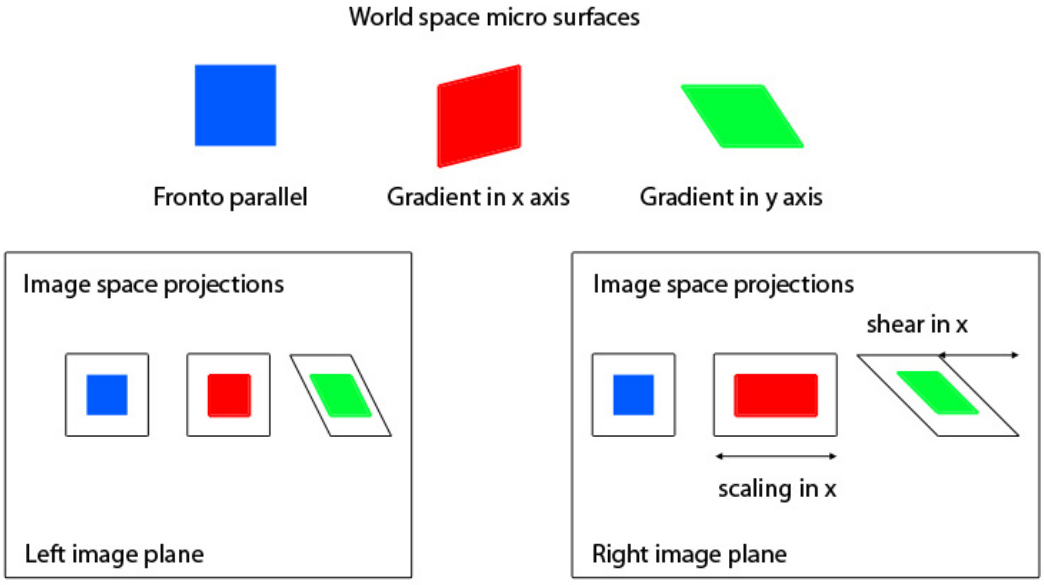

Figure 4.1 Illustration of the effects of surface gradients on the support region

These properties along with the idea of using temporal information led to the development of an algorithm first proposed by Zhang et al.[91] which forms the basis of the work presented in this chapter.

The motivation behind this chapter was to look at the effect of various non-linear optimization algorithms in the context of space-time stereo. The focus was to examine which algorithms converged more quickly and to find the trade-off between the number of iterations necessary for convergence and cost per iteration.

Section 4.1 describes the correspondence problem formulated as a local non-linear optimization problem using the space time warp function as described in Section 2.3 as opposed to just optimizing the parameters of affine warp as is usually the case. This section demonstrates how different solvers such as conjugate gradients, Levenberg-Marquardt and some of their variations can be used. Non-linear optimization is a very broad field of research. However, the purpose of this work is to primarily focus on the algorithms that find local minima, as they can be initialized with the results obtained from the previously described modified three plane DP algorithms presented in Chapter 3 as opposed to the more traditional DP algorithm used by Zhang

et al[91]. Other differences between the work carried out in this section in comparison to the work presented in [91] also include running these algorithms on a dataset containing not only the striped structured light patterns but also the noise patterns presented in Section 3.3.2, modifying the solvers to use a scaled initialization and therefore turning these methods into multi-scale methods, and adding a simple Tikhonov [86] regularization term and therefore removing certain artefacts, without incurring the computational cost of the Levenberg-Marquardt method. The various implementation details of these methods are presented and discussed in Section 4.3. Section 4.4 describes the experiments that were conducted. This is then followed by the results (Section 4.5) and the conclusions that can be drawn from them, in Section 4.6.

4.1 Space-Time Stereo as a Non-Linear Optimization Problem

This section demonstrates how the space-time stereo algorithm [91] is solved using an extended (into the time domain) version of the Lucas and Kanade [51] algorithm (i.e. Gauss-Newton) for the specified warp function proposed by Zhang et al.[91] using a variety of solvers such as conjugate gradients.

The Gauss-Newton method seeks to optimize a parameter vector p by minimizing the sum of the squared error r, which are all functions of p defined as follows:

$$E(p) = \sum_i r_i^2(p) \tag{4.1}$$

Its starts with an initial value p_0 and iteratively minimizes E by updating $p_k = p_{k-1} + \delta p_k$ δp is computed firstly by taking the first order Taylor series approximation to E:

$$E(p_{k-1}+\delta p)=\sum_i r_i^2(p_{k-1}+\delta p)$$
$$E(p_{k-1}+\delta p)\approx\sum_i\left(r_i(p_{k-1})+\frac{\partial r_i}{\partial p}(p_{k-1})^T(\delta p)\right)^2$$
$$E(p_{k-1}+\delta p)=\sum_i\left(r_i(p_{k-1})^2+2r_i(p_{k-1})\frac{\partial r_i}{\partial p}(p_{k-1})^T(\delta p)+\delta p^T\frac{\partial r_i}{\partial p}(p_{k-1})\frac{\partial r_i}{\partial p}(p_{k-1})^T\delta p\right)$$
$$E(p_{k-1}+\delta p)=c+2g^T\delta p+\delta p^T H\delta p$$

4.2

where:

$$c=\sum_i r_i(p_{k-1})^2$$
$$g=\sum_i\frac{\partial r_i}{\partial p}(p_{k-1})r_i(p_{k-1})$$
$$H=\sum\frac{\partial r_i}{\partial p}(p_{k-1})\frac{\partial r_i}{\partial p}(p_{k-1})^T$$

4.3

To minimize *E* the optimal update is:

$$\delta p=-H^{-1}g$$

4.4

In the context of space-time stereo algorithm presented by Zhang et al.[91] the cost functions becomes:

$$\mathrm{E(p)}=\sum_{\mathrm{t}}\sum_{\mathrm{x,y}}\left(I_l(x,y,t)-I_r\left(x+\hat{d},y,t\right)\right)^2$$

4.5

Where

$$\hat{d}=d+\ d_x(x-x_0)+d_y(y-y_0)+d_t(t-t_0)$$

4.6

In the case of dynamic scenes or for quasi static scenes where

$$\hat{d}=d+\ d_x(x-x_0)+d_y(y-y_0)$$

4.7

The optimization parameters p become d, d_x, d_y, d_t and these represent the disparities and their gradient in the x, y, and t axes. The terms *x, y, t* represent the coordinates of

the support region (or window) and x_0, y_0, t_0 represent the centre of that region. By minimizing Equation 4.5 the space-time stereo framework presented by Zhang et al.[91] is effectively also minimizing a warped support region extended into the time domain. The warp of the support region contains a translation, scaling, skew in the x-axis as well as a shearing in the time domain represented by the first, second, third and fourth terms of Equation 4.6. This can also be visualized in Figure 4.2.

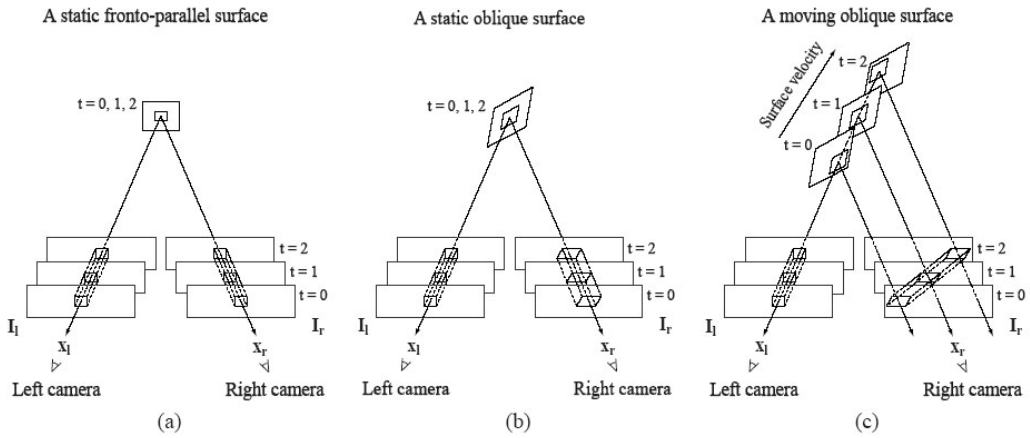

Figure 4.2 Representation of: (a) symmetrical warp, (b) quasi-static warp equation (4.7) & (c) dynamic warp equation (4.6) taken from Zhang et al.[91]

The cost function used for a pixel at (x,y,t) using a support region centred around x_0, y_0, t_0, can be rewritten in the following notation

$$r_i(p) = I_l(x_i, y_i, t_i) - I_r\left(x_i - \left[d, d_x, d_y, d_t\right]\begin{bmatrix} 1 \\ x_i - x_0 \\ y_i - y_0 \\ t_i - t_0 \end{bmatrix}, y_i, t_i\right) \tag{4.8}$$

With partial derivatives as:

$$\frac{\partial r_i}{\partial p} = \begin{bmatrix} 1 \\ x_i - x_0 \\ y_i - y_0 \\ t_i - t_0 \end{bmatrix} \frac{\partial}{\partial x} I_r \tag{4.9}$$

Solving this problem using the standard Gauss-Newton method is achieved by initializing the parameter *d* with the results from dynamic programming algorithm while setting the other parameters to zero. In Zhang et al.[91] a simple DP algorithm was used. However, in the results presented in Section 4.5 the modified Criminsi et al.[19] algorithm presented in Section 3 was used. This is followed by computing the Jacobian matrix using Equation 4.9. The Jacobian is then multiplied with its transpose to produce the Gauss-Newton approximation to the Hessian. The Jacobian is also multiplied with *r* to produce the gradient vector. The resulting linear problem represented by Equation 4.4 can then be solved using a variety of linear solvers such LU decomposition. However, since the Hessian is symmetrical it is more efficient to use Cholesky decomposition in this particular case, the solution of which produces the optimal update in the following iteration until convergence or some predetermined error threshold is reached.

One alternative approach to the Gauss-Newton algorithm is to replace:

$$\begin{aligned} H\delta p &= -g \\ \left(J^T J\right)\delta p &= -J^T r \end{aligned}$$

4.10

With a damped version

$$\left(J^T J + \lambda I\right)\delta p = -J^T r$$

4.11

The (non-negative) damping factor λ is adjusted in every iteration. If the reduction of the sum of *r* is rapid a small value for λ is used therefore bringing the solution closer to the Gauss-Newton. If an iteration gives insufficient reduction of the residual λ is increased, bringing the solution closer to the steepest descent direction. This algorithm is referred to as the Levenberg-Marquardt algorithm [49].

Yet another alternative to solving Equation 4.4 is to use the iterative conjugate-gradient [75] algorithm as opposed to explicitly inverting the Hessian matrix. Suppose one wishes to minimize a function *f* which is roughly approximated as a quadratic form:

$$f(x) \approx c - g \cdot x + \frac{1}{2} x \cdot A \cdot x$$

4.12

Starting with an arbitrary initial vector g_0 and letting $h_0 = g_0$, the conjugate gradient method constructs two sequences of vectors from the recurrence:

$$g_{i+1} = g_i - \lambda_i A \cdot h_i \qquad h_{i+1} = g_{i+1} + \varsigma_i h_i$$

4.13

The vectors satisfy the orthogonality and conjugacy conditions:

$$g_i \cdot g_j = 0 \qquad h_i \cdot A \cdot h_j = 0 \quad g_i \cdot h_j = 0 \qquad j < i$$

4.14

The scalars λ_i and ς_i are given by:

$$\lambda_i = \frac{g_i \cdot h_i}{h_i \cdot A \cdot h_i} \qquad \varsigma_i = \frac{g_{i+1} \cdot g_{i+1}}{g_i \cdot g_i}$$

4.15

Given the Hessian matrix A one can solve the system by iteratively applying line minimizations along the conjugate directions h_i. It is however possible to solve without the need of the Hessian matrix A as follows.

$$g_0 = h_0 = -\nabla f(x_0)$$

4.16

Find α that minimizes $f(x_i + \alpha_i h_i)$,

$$\begin{aligned}
x_{i+1} &= x_i + \alpha_i h_i \\
g_{i+1} &= -\nabla f(x_{i+1}) \\
\varsigma_{i+1} &= \frac{g_{i+1} \cdot g_{i+1}}{g_i \cdot g_i} \; or \; \varsigma_{i+1} = \max\left\{ \frac{(g_{i+1} - g_i) \cdot g_{i+1}}{g_i \cdot g_i}, 0 \right\} \\
h_{i+1} &= g_{i+1} + \varsigma_{i+1} h_i
\end{aligned}$$

4.17

The choice of ς_{i+1} is between the Fletcher-Reeves and the Polak-Ribier variation of the algorithm. The line minimization can be performed with various algorithms such as Newton-Raphson, Secant, Bracketing or equivalents.

4.2 Space-Time Stereo Implementation

The sample acquisitions were performed using the same setup described in Section 3.2. The various optimization algorithms were implemented in Matlab. The Conjugate-Gradient algorithms were implemented using both the successive Newton-Raphson line minimisations as described [68] along with the algorithm described in Section 4.1 using Gauss-Newton approximation to the Hessian. The Gauss-Newton algorithm that explicitly inverts the Hessian matrix was also implemented. The same cost function (Equation 4.5) described in Section 4.1 was minimized with these algorithms and also minimized using the Matlab optimization toolbox [53].

All these implementations used a support region of 5x5x8 pixels unless specified otherwise. The vectors of parameters $p = [d, d_x, d_y, d_t]$were initialized using the results from the Criminisi et al.[19] algorithm that was modified to support a space-time support region with the normalized SSD cost function. The structured light patterns used were both the shuffled Gray code and noise pattern presented in Section 3.3. Certain Matlab operators are very well optimized. This library is outperformed by commercially available libraries such as the Intel Maths Kernel library [38]. However, other Matlab operators and built in functions perform very poorly. One such example is the bilinear interpolation function. This function's performance was so poor that an alternative was implemented.

4.3 Space-Time Stereo Non-Linear Optimization Experiments

The motivation behind the experiment described in this section was to find the best performing algorithm in the frame work of space-time stereo, as a non-linear optimization formulation. This formulation used the warp function and shuffled Gray code specified by Zhang et al.[91], as well the noise structured light pattern. Performance was measured in terms of the rate of convergence and the qualitative results of each non-linear optimization algorithm. It was also necessary to determine

whether or not the advantages in quality attained by this formulation warrant the added computational burdens. The following is a list of criteria that the experiments conducted in this section were designed to answer.

- The rate of convergence of the various non-linear optimization algorithm described in Section 4.1
- The qualitative impact of using different support region or window sizes
- The qualitative impact of terminating the non-linear optimization before convergence
- The qualitative impact of using the noise structured light pattern as opposed to the stripes suggested by Zhang et al.[91]
- The qualitative impact of using a lower resolution disparity map to initialise the non-linear optimization step and therefore potentially speedi up the algorithm
- The qualitative impact of Tikhonov regularization.
- Determining the relative computational cost of the various subroutines in each optimization algorithm.

All the data used in following experiments was captured using the capturing setup described in Section 3.1. The same cameras were used along with the same structured lights patterns. This was followed by calculating the per-pixel disparities using the modified Criminisi et al.[19] algorithm presented in Section 3.3.4. This included extending the support region into the time domain and using the normalized SSD cost function. Once the disparity maps were computed using the previously described DP algorithm, they were subsequently used as initializations for the non-linear solvers.

4.4 Space-Time Stereo Non-Linear Optimization Results

Having devised a list of criteria, the following experiments were conducted, the results of which will be presented and discussed in Section 4.4. The various implementations discussed in Section 4.2 used to solve Equation 4.5 as well as the non-linear optimization toolbox in Matlab were tested for convergence against the same dataset. The results of which are presented in Section 4.4.1. The Preconditioned Conjugate

Gradient, Levenberg Marquardt and Gauss-Newton optimizations algorithms from the Matlab toolbox were all used, as well as a re-implementation of the Gauss-Newton. Each of these algorithms were run with the analytical and finite difference approximation to the Jacobian, presented in Equation 4.9. With the motivation of potentially reducing aliasing artefacts and improving computation speed, these experiments were then followed by experiments that changed the size of the support region as well as the number of iterations performed (see Section 4.4.2). Section 4.4.3 will demonstrate the advantage of using a warped support region. Subsequently with the goal of further reducing artefacts without incurring the computational burden of the Levenberg-Marquardt algorithm, the simpler Tikohnov regularization was tested in Section 4.4.4, as well as the impact of extreme motion with a window that was not warped in the time domain. With the further motivation of getting these non-linear optimization algorithms running in real-time, Section 4.4.5 will determine the impact of using a lower resolution disparity map for initialization. Having performed all the experiments up to this point using the same structured light patterns presented by Zhang et al.[91] it was also necessary to establish the effects of using the noise structured light patterns presented in Section 3.2, the results of which are discussed in Section 4.4.6. Although all the experiments conducted in this section were implemented in Matlab and do not come close to running in real-time for a multitude of different reasons it was still beneficial to profile the Matlab implementations in order to determine computational bottlenecks and gain more insight into the potential gains acquired by reformulating them as streaming algorithms that could leverage the performance of GPUs. Section 4.4.7 will demonstrate how some of these algorithms are appropriate for implementation on a GPU.

4.4.1 Convergence

To establish the rate of convergence of each algorithm, the disparity and respective gradient values along particular scan-lines were optimized, while taking the average of the residuals during the iterations. The results are illustrated in Figure 4.3. In all cases, using the analytical partial derivatives based on Equation 4.9, produced better results and quicker convergence then their finite difference approximation. Both the Gauss-Newton and Levenberg-Marquardt produced very similar results. They converged on

average within 4-6 iterations. Depending on the particular pixels being optimized one or the other would converge more quickly. The conjugate-gradient algorithms would converge far more slowly, on average more than 10 iterations were required for convergence. The Newton-Raphson line optimization seemed to perform the best with conjugate-gradient. Unfortunately this is a very costly line minimization. Figure 4.4 illustrates a close up of a 3D reconstruction based on the Levenberg-Marquardt optimized disparity map. Figure 4.5 compares the difference between a 3D reconstruction based on the Gauss-Newton optimization of space-time stereo, and therefore shows the added details acquired by the sub-pixel disparities versus the previously mentioned Criminsi et al.[19] space-time stereo used as initialization.

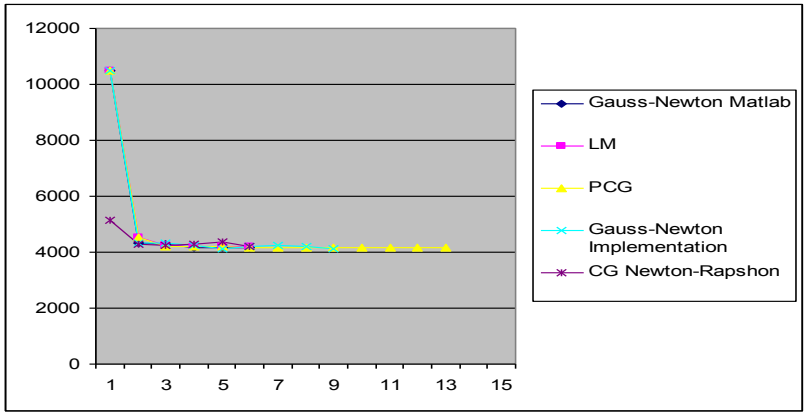

Figure 4.3 Graph illustrating convergence with x-axis representing number of iterations while the y-axis represents residuals

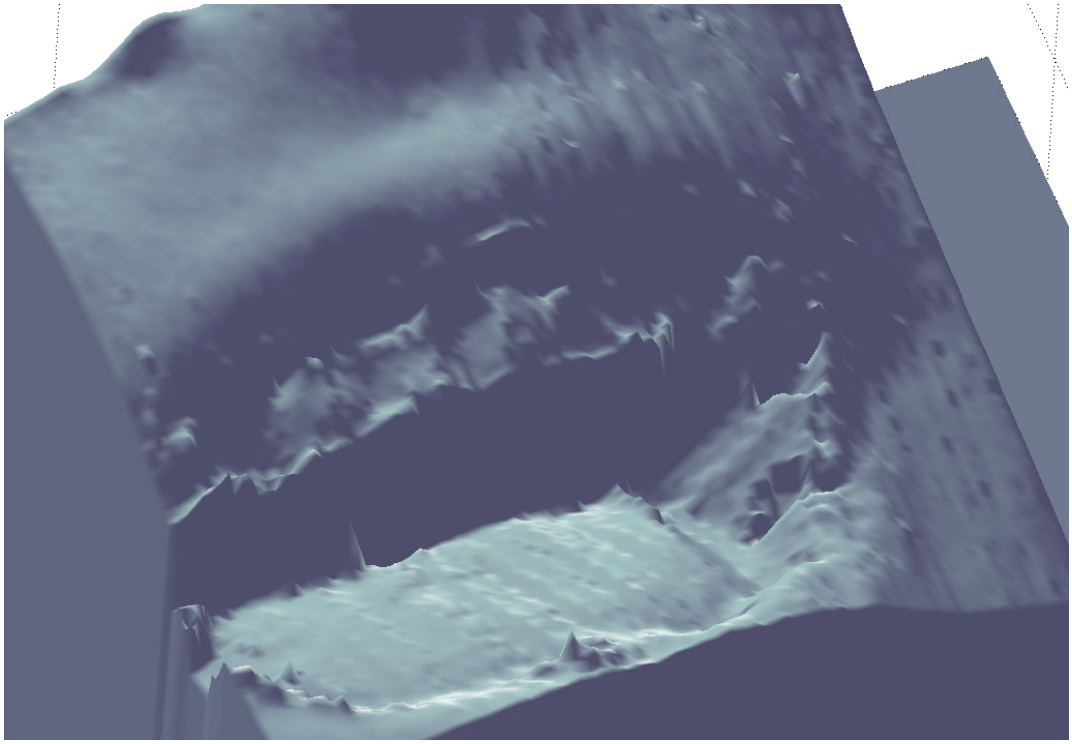

Figure 4.4 3D Reconstruction of the mouth area using disparity map produced with Levenberg-Marquardt optimization after 5 iterations using 5x5x8 support region.

Using an adaptive support region defined by the warp function in Equation 4.5 produces much smoother disparity maps without the compromise of losing high frequency detail. Using the analytical solution for the Jacobian not only improves performance and convergence rate, but also has the advantage that part of its calculation (i.e. ∇I_r) can be pre-computed for each frame and remains constant for each disparity computation. Another observation is that the first couple of iterations contribute significantly to the result. This implies that a potential compromise between quality and the number of iterations that might be performed can be achieved.

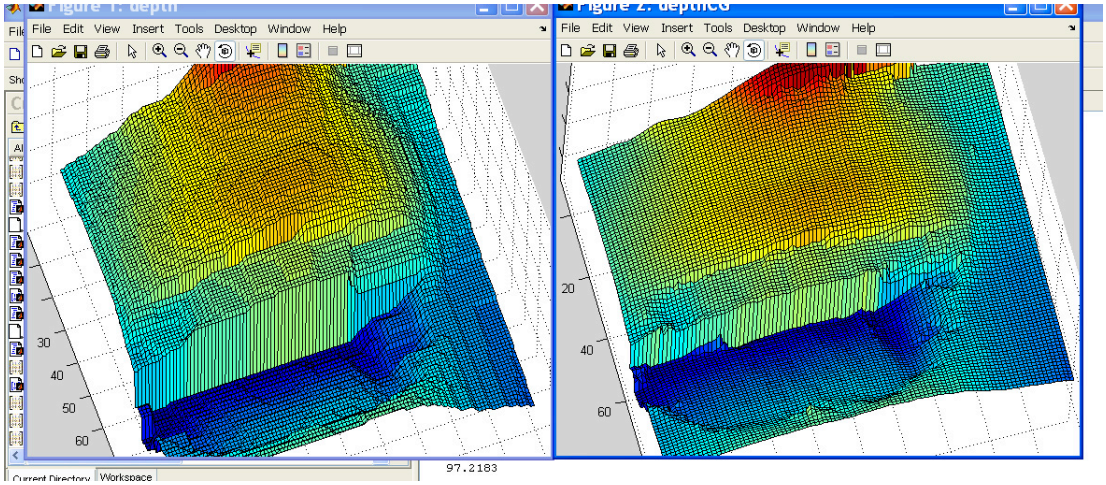

Figure 4.5 Illustrating the difference between the Criminisi et al.[19] space-time algorithm (left) versus the Gauss-Newton optimized disparity map (right)

In summary the Gauss-Newton algorithm performed suitably and produced results comparable to the Levenberg-Marquardt and Conjugate-Gradient. Not only did it converge more quickly than Conjugate-Gradient, but it also displayed the benefit of not having to compute the regularization or dampening term associated with the Levenberg-Marquardt method, that involves a further line optimization step, therefore increasing the computational time per iteration significantly.

4.4.2 Parameters (Window Sizes and Iterations)

Having determined the advantages of the Gauss-Newton optimization method, the following results examine the impact of the support region size as well as potentially stopping the optimization before convergence. These parameters have significant implications in both terms of the quality of the output disparity maps and the computation time required. Although the previous section demonstrates that convergence using the Gauss-Newton algorithm is usually reached after five iterations, one can significantly speed up computation by terminating with fewer iterations. The support region size has more subtle effect on the results, by increasing the support regions, the accuracy of the correlation matching is increased (i.e. the number of incorrect disparities and noise is reduced). However, this comes at the cost of high frequency details that are removed along with the noise. This is a common trade-off in correlation based algorithms, but using space-time support region somewhat alleviates this trade-off. By extending the support region into the time domain one can maintain

the same number of pixels in support region while reducing the size the space domain (i.e. a 5x6x8 window contains the same number of pixels as a 15x16 window). This is effectively trading resolution in the space domain against resolution in the time domain.

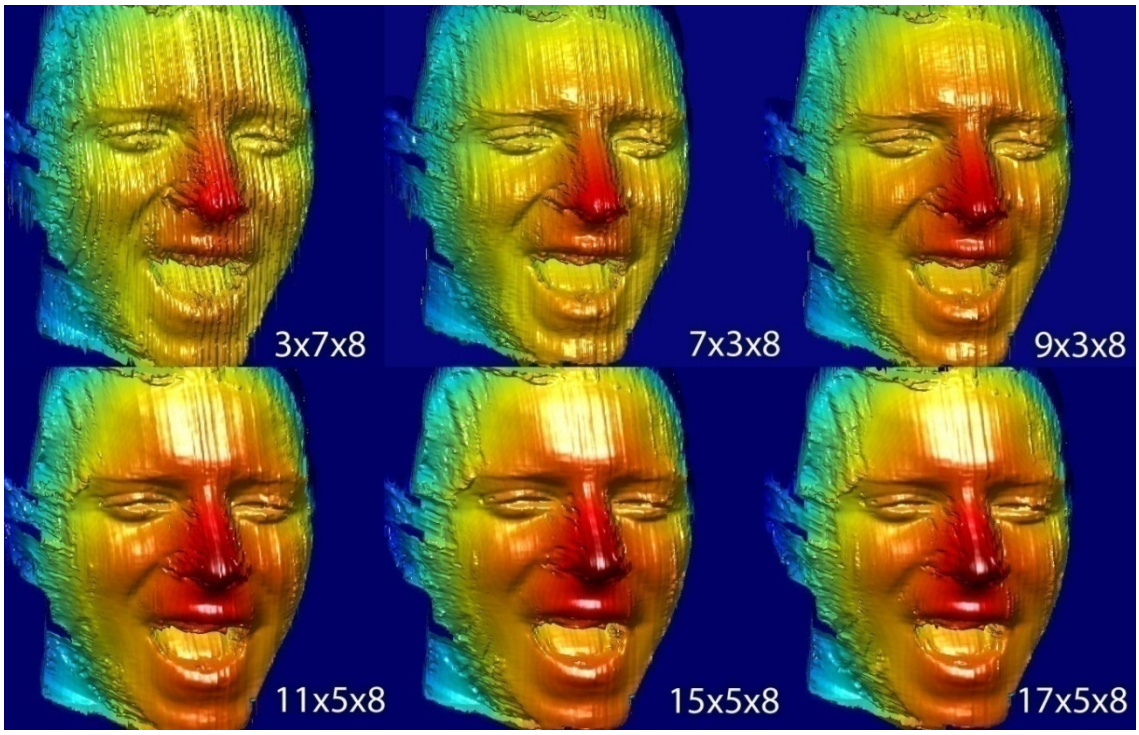

Figure 4.6 Gauss-Newton non-linear optimization using shuffled Gray code light pattern and space-time warp function on various window sizes after 5 iterations

Figure 4.6 depicts six reconstructions all using different support region sizes, solved using the Gauss-Newton non-linear optimization on the identical dataset containing images of a face illuminated by the shuffled Gray code described in Chapter 3. This figure clearly shows that using different window or support region sizes can have a significant impact on the results of the Gauss-Newton optimization. Another observation determined from Figure 4.6 is that increasing the window size along the x-axis improves the results more significantly than increasing the window size in the y-axis. This is shown by the first two reconstructions in Figure 4.6 which both contain support regions with the same number of pixels (3x7x8 and 7x3x8). However, the reconstruction with the window of 7x3x8 produces results with significantly less noise.

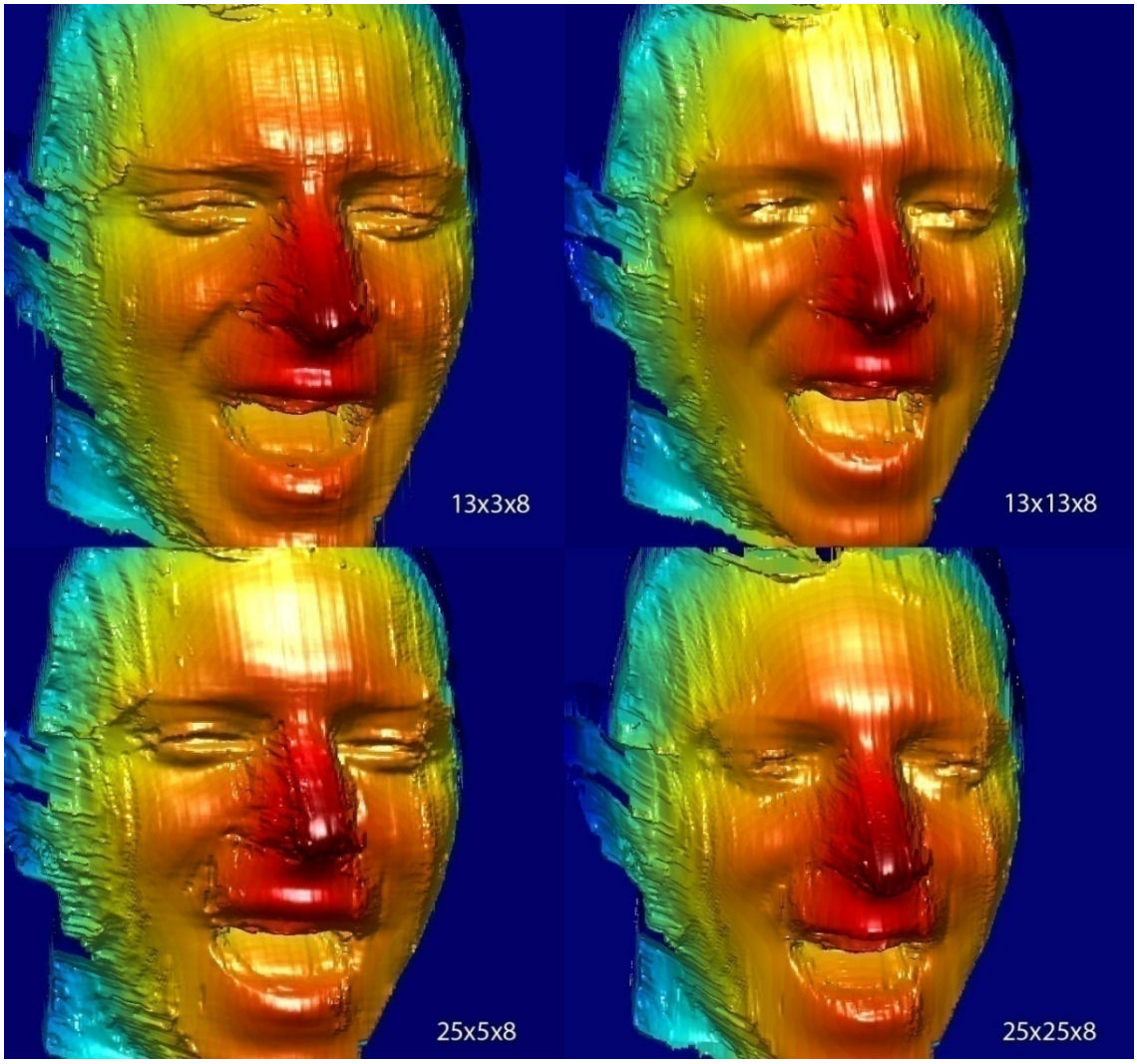

Figure 4.7 Gauss-Newton on various window sizes after 5 iterations

The noisy artefacts are caused by the fact that the illumination patterns contain far greater frequency changes in the x-axis as opposed to the y-axis and thus increasing the support region along the x-axis is effectively adding a lot more entropy to the solver. This also demonstrates that this solver is also sensitive to high frequency changes in the images and requires the appropriate considerations with regard to window sizes depending on the illumination pattern. Figure 4.6 also illustrates that increasing the window sizes further along the x-axis further improves results while still maintaining high frequency details in the reconstruction. This is due to the fact that the support region is adaptive. However, this statement only remains valid up to a certain point. Once the support regions become too large, the reconstruction results start to deteriorate, as can be seen from Figure 4.7.

Having examined the effect of the window sizes on the results, it was also necessary to determine the impact of performing fewer iterations. The previous section demonstrated that convergence of the Gauss-Newton optimization would occur after five to six iterations for most disparity values. After six iterations the residual error tended to oscillate, with a few outlier pixels that would diverge. The convergence is itself non-linear and tends to be exponential (Figure 4.3). For each iteration the residuals of the cost function are reduced less than during the previous iteration, which creates a trade-off between the number of iterations performed and the quality of the results. By potentially stopping the algorithm sooner one can significantly increase the computational speed and also potentially reduce artefacts created by pixels that create ill-conditioned systems within the solution.

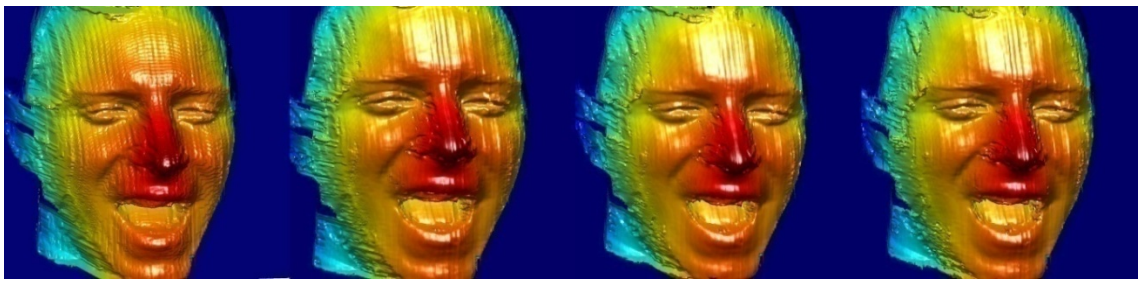

Figure 4.8 Gauss-Newton reconstruction using 11x5x8 window after 1,3,5 and 10 iterations

Figure 4.8 illustrates the reconstruction results performed using a window of 11x5x8 pixels after 1,3,5 and 10 iterations respectively. It demonstrates the diminishing returns of performing more iterations. After three iterations the results are close to optimal, reaching convergence at five iterations, while performing ten iterations actually has a negative impact as certain errors and artefacts are actually accentuated. This also demonstrates that not all disparity values converge after five iterations and the values that do not converge tend to produce aliasing artefacts. These non-convergent values tend to be created by occlusions. This highlights one inherent weakness of the non-linear optimization part of this algorithm, which is that is does not deal with occlusions.

One thing that all the results presented in this section have in common is the presence of errors. Some of these errors are produced by pixels that are converging to incorrect values, and all these results contain the banding artefacts mentioned by Zhang et al.[91] that were subsequently removed using a global optimization scheme with a disparity gradient constraint. The parameters (window sizes and iterations) examined in

this subsection can reduce the significances of these the errors but do not entirely eliminate them.

4.4.3 Slanted versus Non-Slanted Window and Extreme Motion

To demonstrate the advantages of using a warped support region, a comparison of two reconstructions was made. Figure 4.9 clearly demonstrates the advantages in terms of quality afforded by using a slanted window for the reconstruction. Although the reconstruction using the slanted window still contains some banding artefacts, they are significantly less pronounced than the reconstruction that only optimizes translations along a rectangular support region.

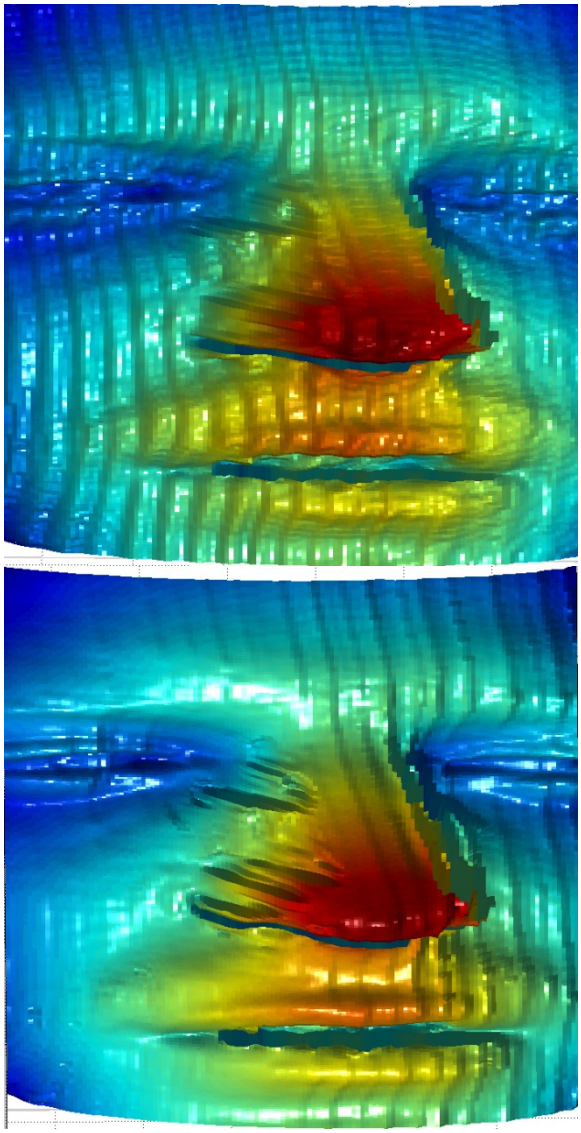

Figure 4.9 Comparison between reconstruction using non-slanted (left) and slanted (right) windows after 3 iterations using a window size of 7x5x8

It is worth noting that all the reconstruction performed in this thesis have used samples from an image sequence that contained motion (i.e. a subject talking) as seen in Figure 4.10. This motion would be typical in a face to face communication scenario. In order for this implementation to be achieved in real-time as demonstrated in Chapter 5 & 6, it was necessary to drop the time domain warp parameter. Doing so has little impact when performing the reconstruction on our sample data set. This is illustrated in Figure 4.10, which does not contain extreme motions and was sampled at 60 frames per second. However, if a reconstruction is performed for every third frame and therefore simulating the effect of speeding up the motion by a factor of three, one will notice degradation of the results as shown in Figure 4.11.

Figure 4.10 Frames taken from sample data set illustrating motion of the mouth as the subject is seen to be talking

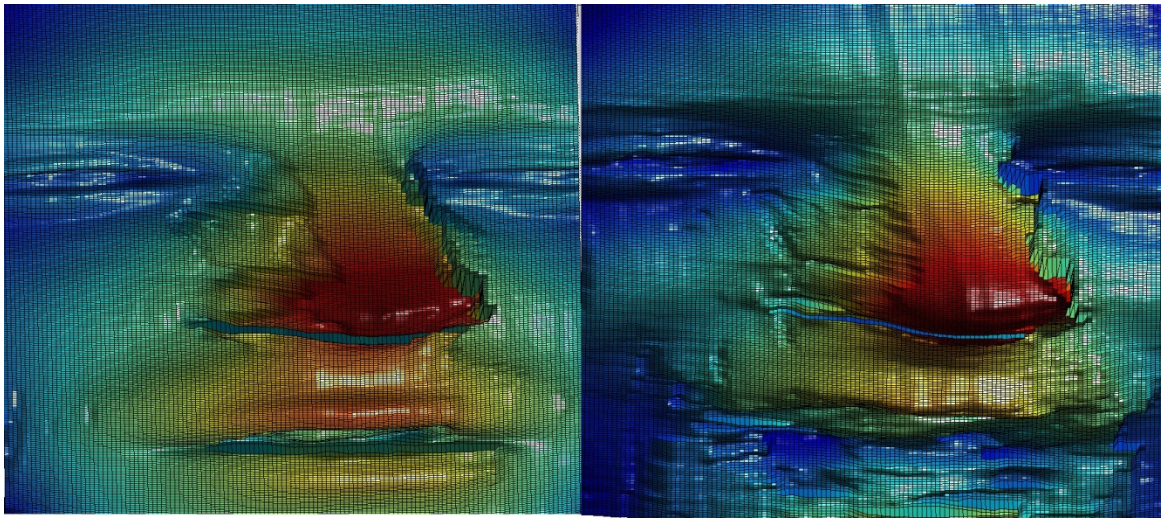

Figure 4.11 A comparison between reconstruction based on every frame (left) and every third frame (right) without warp in the time domain.

4.4.4 Regularization and Artefacts

Having experimented with various window sizes and number of iterations it was necessary to try and determine the cause of some of the resulting artefacts. As described in the initial section of this chapter, each disparity value for each pixel is obtained by solving a system of equations containing four parameters, the disparities and their gradients with respect to the x,y, and time axis. However, these parameters are solved as independent parameters whereas they are not in reality. The over-parameterization produces the very noticeable banding artefacts. This was pointed out in a follow up paper by Zhang et al.[91], in which some of the banding artefacts previously described were eliminated by reformulating the optimization problem as a global optimization, and therefore allowing the disparity gradients constraint by a finite difference operator. Although this approach is elegant it requires constructing a large sparse matrix and solving it using a conjugate gradient method. This not only reduces the speed at which the system will converge but add significant memory and computational burdens making it non feasible for real-time applications at the time this work was carried out. The motivation behind the results presented in this section was firstly to determine which disparity values are part of ill-conditioned systems and then to determine whether the results could be improved upon by using a simple Tikonov regularisation on those particular values to improve the results.

The standard method normally used to determine the conditioning of a system of linear equations is to perform SVD or Eigen decomposition and examine the ratio of the

highest and lowest singular values. For this application, the decomposition would be performed on the Gauss-Newton approximation to the Hessian in Equation 4.10. However performing the decomposition for every Hessian of every pixel at runtime would be impractical for real-time applications. Given that the Hessian in Equation 4.10 is symmetric by definition, the Equation 4.10 can be solved using Cholesky decomposition, which itself contains a mechanism for detecting singular matrices (see numerical recipes [68]). Figure 4.12 illustrates the results of a reconstruction that was altered to zero out all the disparity values that form a Hessian that are close to being singular. These pixels are depicted in Figure 4.12 by being the same colour as the background. One interesting observation is that not all occluded pixels form ill-conditioned systems, one is also surprised by the number of zeroed disparity values that tend to be in part of the image that can be considered good (i.e. fronto-parallel and visible by both stereo cameras). One potential method of dealing with these pixels would be to remove them from the solver and then, having solved the other pixels in a further pass, use some quadratic fitting function to obtain sub-pixels values for these particular disparity values.

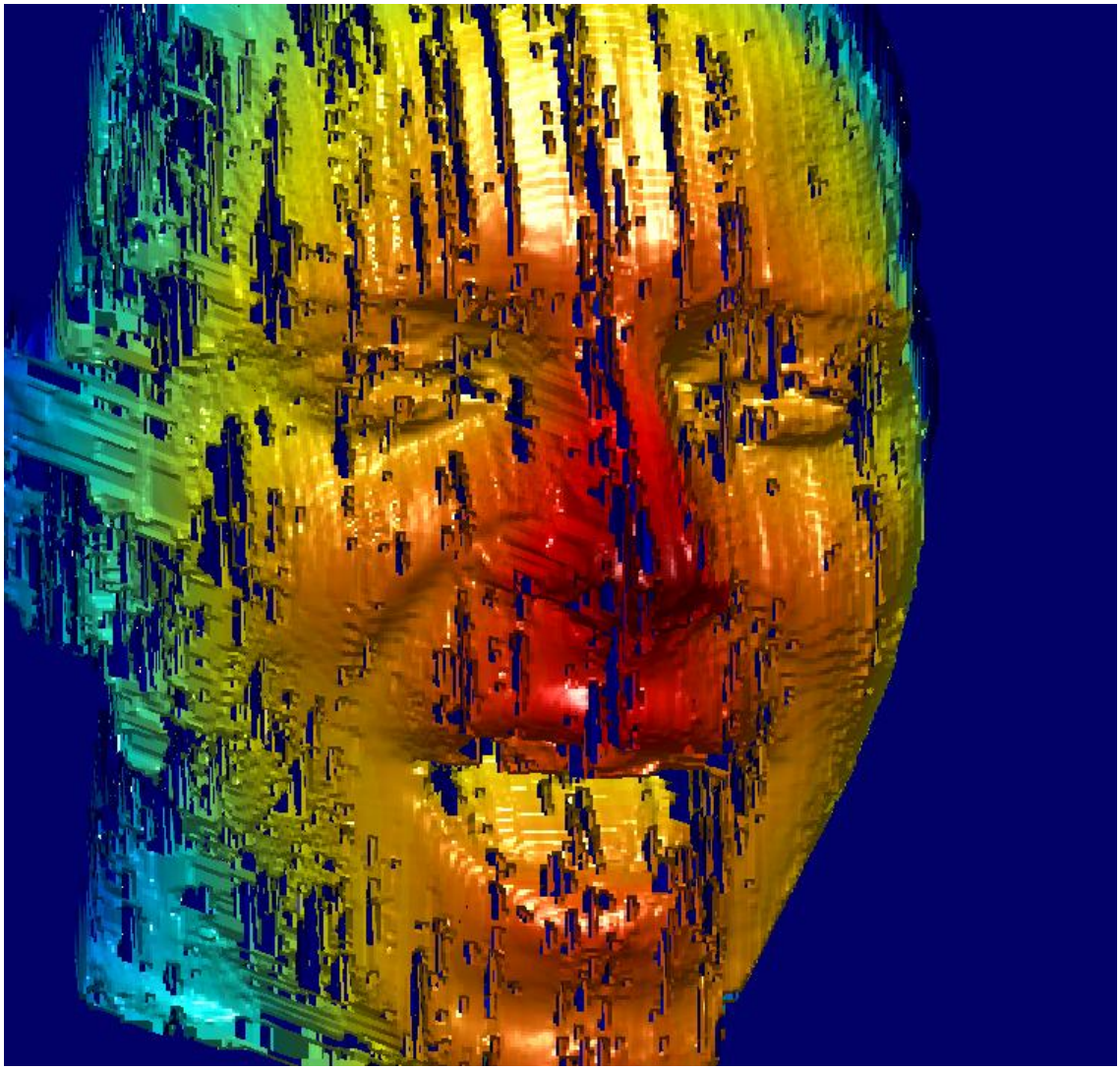

Figure 4.12 Reconstruction with the wholes representing pixels forming close to singular hessians

One alternative is to regularize the data, by replacing Equation 4.10 with the following.

$$\left(J^T J + \lambda I\right)\delta p = -J^T r$$

4.18

Although this is the same as the Levenberg-Marquardt there is a subtle difference. In the Levenberg-Marquardt algorithm the dampening parameter is adjusted for each iteration and therefore requires a line-search further increasing the computational complexity. With this proposed method the dampening parameter is fixed and can be determined offline. The method therefore only requires a few additions to the Hessian matrix, making it very cheap to perform. This can be either applied across the whole dataset or to only the values representing ill-conditioned systems. The effect this has is to slow

down convergence and therefore proves to be un-advantageous for the application across all values. This is illustrated in Figure 4.13.

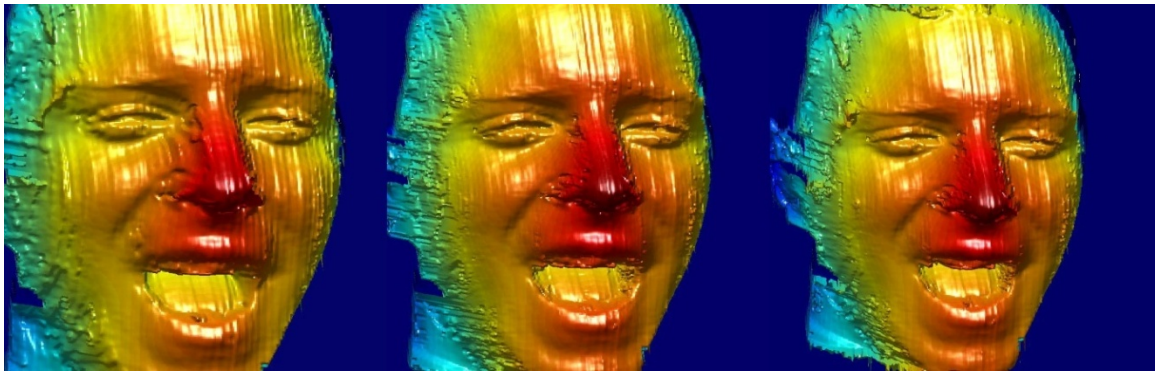

Figure 4.13 Reconstruction left (with all pixels regularized), centre (with only close to singular regularized), right no regularization all using 11x5x8 window

Unfortunately as Figure 4.13 illustrates using Tikonov regularization across all pixels produces little benefits as it severely slows down the convergence. Furthermore, performing this regularization solely on the pixels containing close to singular Hessian matrices yields little benefit due to the fact that the pixels detected by the Cholesky solver as being close to singular tend not to correspond to the pixels producing the most artefacts. The artefacts tend to be caused by a variety of factors, the primary ones being occlusions, specular reflections and the fact that the light pattern is distorted when projected onto near oblique surfaces. Fortunately the Criminisi et al.[19] stereo algorithm presented in Section 3.3 contains a built in mechanism for detecting occlusions. This information could potentially be carried over, to instruct the non-linear optimization solver to, not solve occluded pixels and instead carry out a quadratic fitting for those particular problematic disparity pixels. The impact of occluded pixels could also further be reduced by reducing the baseline line of the stereo cameras. This however, would also reduce the precision of the depth information. In further attempts to reduce artefacts the same reconstructions were also performed on Gaussian blurred versions of the input images. This however had a negative impact on the results and was quickly dismissed.

4.4.5 Multi-Scale

Although the Zhang et al.[91] space-time non-linear optimization algorithm presented in this section was not originally intended for real-time application, it is quite suitable for implementation on massively parallel architectures. This is due to the fact that each pixel is treated as an independent set of simultaneous equations across an adaptive support region, and therefore each disparity is treated independently. This makes the algorithm ideal for implementation on some type of streaming computing architecture such as the latest versions of GPUs. The computational performance is therefore of great significance. One easy method of increasing this performance is to initialize the data using lower resolution disparity maps. This would enable the initialization step of the algorithm to be run on a much lower resolution dataset, thereby dramatically reduce its computational cost. All the results presented in the previous sub-sections used the same resolution integer based disparity maps to initialize the non-linear space-time solver. It is therefore important to determine the impact of using lower resolution initialization disparity maps that would then be appropriately scaled. Figure 4.14 illustrates three reconstructions all initialized using different resolution disparity maps. It is worth noting that linear interpolation was performed while up-scaling the lower resolution initialization maps.

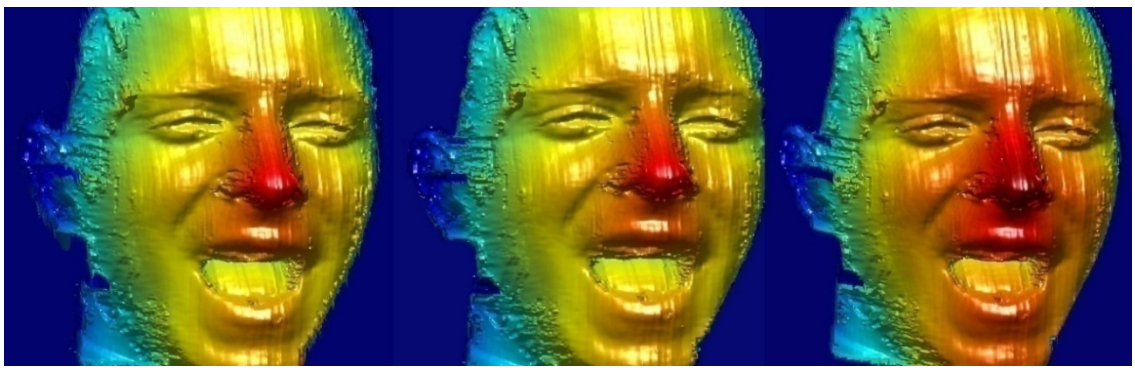

Figure 4.14 Reconstruction using 160x120 (left), 320x240 (centre), 640x480 (right) disparity maps for initialization of the non-linear optimization

The reconstruction performed using sub sampled half resolution disparity map has the effect of removing certain artefacts while introducing new ones, making it similar in quality to that of the reconstruction initialized by a full resolution disparity map. The reconstruction performed using the lowest sub sampled disparity map introduces

artefacts near depth discontinuities, however the final results are good and close to the non-scaled reconstruction.

The results presented by Figure 4.14 demonstrate the small trade-off in terms of quality produced when using a sub-sampled disparity map for initialization, and therefore point to one avenue of speeding up the computational time required by vastly reducing the computational burden of the initialization step.

Figure 4.15 Close up of Figure 4.11

4.4.6 Structured Light

The previous sub-sections all contain reconstructions performed with images illuminated with the shuffled Gray code patterns discussed in Section 3.2. In an attempt to further reduce the banding artefacts present in these reconstructions, the algorithm was run on a dataset of images illuminated with the low-pass filtered noise patterns presented in Section 3.2. Although the banding is created by the fact that the space time warp parameters are optimized as independent variables when in actual fact they are not, the analytical approximation to the Jacobian matrix uses the image gradients, that appear to be correlated to the high frequency changes of the striped illumination pattern. It was felt that using a different structured light pattern could potentially reduce these artefacts without the need for reformulating this problem as a very computationally costly global optimization problem.

Figure 4.16 depicts the results of using the noise structured light pattern. They still exhibit the banding artefacts, and are far inferior results with the striped pattern. Using the noise structure light pattern required increasing the window size considerably in both the x and y axis to obtain similar results to the reconstruction performed using the striped code pattern. Whereas while using the striped pattern one could get away

with a relatively small window size in the y-axis of 3 to 5 pixels, this was not the case on with the noise light pattern which required the window size to be of similar dimensions in both axes to obtain adequate results. This implies a performance penalty in terms of computational time for using the noise patterns as well as a reduction in the output quality.

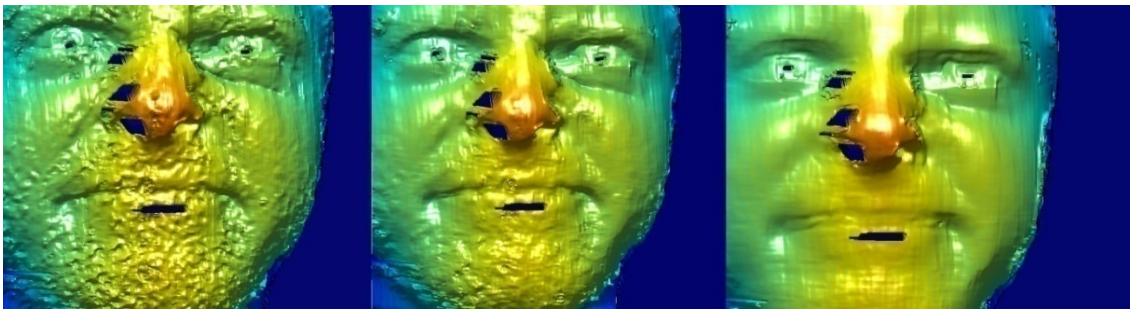

Figure 4.16 Reconstruction using 7x7x8 (left), 11x11x8 (centre), 21x21x8 (right) Windows on scene illuminated with low pass filtered noise patterns

4.4.7 Algorithm Profiling

The following Table 4.1 is a breakdown of the time spent in different functions called by the re-implemented Gauss-Newton non-linear optimizer, for one iteration for one disparity value. As can be clearly seen from this table the major bottleneck is the gradient function which calculates the image gradient used for the partial derivatives of the Jacobian. This however only needs to be calculated once per frame. 3.344s but is still very slow. Another testament that Matlab built in function are not always implemented with performance in mind. The other major bottleneck is the SeitzSSDJ function. This is the function that calculates the error term as well as the Jacobian. Table 4.2 is a breakdown of the SeitzSSDJ function, which illustrates another big bottleneck, namely the re-implementation of the bilinear interpolation function. This re-implementation runs an order of magnitude faster than the equivalent built in function.

Function name	Calls	**Total Time**	Self Time*	Total Time Plot (dark band = self time)
convergeTest2	1	6.094 s	0.047 s	
gradient	1	3.672 s	3.344 s	
gaussnewton	1	1.656 s	0.031 s	
SeitzSSDJ	1	1.625 s	1.625 s	

Table 4.1 Profile Matlab Gauss-Newton High level where total time represents the time spent in the function and all its subroutines while self-time represent the total time minus the subroutine calls.

Line Number	Code	Calls	Total Time	% Time	Time Plot
109	`grad = bilinear(Grads(:,:,t), ...`	200	1.375 s	84.6%	
64	`i = bilinear(Limgs(:,:,t), v, ...`	200	0.234 s	14.4%	
131	`end`	40	0.016 s	1.0%	
74	`test =  int32(i)-int32(Rimgs(v...`	200	0.001 s	0.0%	
112	`J(Idx, 2) = (u-X) * double(gra...`	200	0.000 s	0.0%	
Other lines & overhead			0 s	0%	
Totals			1.626 s	100%	

Table 4.2 Profile of SeitzSSDJ function

The profile shown in Table 4.2 also contains timing with regard to loading the various images as well as other redundant operations. If these are removed the performance can be increased to take less than 1.0s per iteration per disparity value. By deduction one will notice that 0.031s is spent on the following operations $J^T J, J^t F(x)$ inverting the Hessian $J^T J$ updating the parameters and calculating the residuals for the convergence analysis. Ideally the bottleneck should lie with both matrix multiplications and the Hessian inversion. Table 4.3 contains a similar profile, but of the preconditioned conjugate gradient algorithm and Table 4.4 contains the profile of the Newton-Raphson conjugate gradient algorithm. Although both Tables 4.3 and 4.4 show the conjugate gradient algorithms performing more slowly than the Gauss-Newton algorithm per iteration, this can be explained by the fact that the main bottle neck is the cost computation which in the case of both conjugate gradient methods is called more than once. Should the cost computation bottle-neck be removed the conjugate gradient method without the pre-conditioning step should be faster as it does not require the

matrix inversion step. This algorithm can also be sped up further by using a bracketing line minimization algorithm as opposed to the Newton-Raphson line minimization.

Function name	Calls	**Total Time**	Self Time*	Total Time Plot (dark band = self time)
convergeTest2	1	9.188 s	0.078 s	
SeitzSSDJ	3	4.859 s	4.797 s	
cgn	1	4.859 s	0.000 s	
gradient	1	3.594 s	3.266 s	

Table 4.3 CG Newton-Raphson

Function name	Calls	**Total Time**	Self Time*	Total Time Plot (dark band = self time)
convergeTest	1	8.109 s	0.047 s	
gradient	1	3.906 s	3.594 s	
optim\private\lsqncommon	1	3.406 s	0.016 s	
lsqnonlin	1	3.406 s	0.000 s	
SeitzSSDJ	2	3.250 s	3.250 s	
optim\private\snls	1	1.734 s	0.047 s	

Table 4.4 Precondition Conjugate Gradient

However, as demonstrated in Section 4.4.1 these performance advantages are minimized by the fact that many more iterations are required for convergence. In the case of the local space-time stereo algorithm where each disparity value is optimized separately and therefore the Hessian matrix is small (4x4), the advantages usually associated with the conjugate-gradient method are unwarranted. Further tests have shown that the matrix inversion step that runs in O(n^3) time is performed in less than 0.001s in Matlab, which demonstrates the efficiency of Matlab at performing certain calculations, combined with its inefficiency at performing others (e.g. bilinear interpolation). This makes this platform less than ideal for profiling the performance of algorithms. However, initially the concern was with the quality of the results and convergence rates, as well as identifying some of the potential bottlenecks. This helped to determine the suitability of the type of algorithm for implementation on GPUs.

4.5 Conclusions

This chapter presented various alternatives for space-time stereo algorithms. It has illustrated the advantages of the non-linear optimization algorithm over the more computationally efficient DP algorithms. The use of the Gauss-Newton algorithm for solving this problem locally has been justified. Some of the weaknesses of this algorithm have been examined as well as potential solutions. The impact of Tikonov regularization was tested. Modifying the algorithm to use scaled sub-sampled disparity maps for initialization was performed and shown to produce comparable results, and therefore removing burden from the initialization steps. The use of a noise structure light pattern was shown not to give any benefits unlike the results obtained from Section 3.2. Section 4.4.6 also highlighted some of the major bottlenecks, notably the bilinear interpolation function. However there is hardware that is designed specifically to perform these types of operations, namely GPUs. These are incredibly adept at performing bilinear interpolation as well as sum operations and it will be shown in Chapter 5 they can also be used for more general computations such as matrix inversions using LU decomposition with partial pivoting. They have been successfully employed to solve a number of general problems that resemble the stereo correspondence problems. However there are limitations such as latency and the constrained nature of the programming language. Chapter 5 will present ways to overcome some of these limitations and leverage benefits of modern day GPUs in order to shift the bottleneck from the cost computation to the actual Gauss-Newton optimization step.

Chapter 5

Space Time Stereo on the GPU

Having demonstrated the advantages obtained by performing a non-linear optimization step in the previous chapter, the main objective of this chapter is to determine the feasibility of getting the non-linear optimization step to run in real-time using GPUs. Consumer demand over the last decade has pushed the graphics processor industry to develop ever more computationally powerful and flexible GPUs. The development of these processors has so far have exceeded Moore's law prediction of doubling in computational performance every 18 months, and over the last few years have overtaken CPUs in terms of outright GFLOPS performance (Figure 5.1). Initially developed to accelerate 3D graphics pipelines and handle vast amounts of geometry and texture data, these GPUs have evolved by becoming fully programmable and now support 32 bit floating point arithmetic. This has led to a recent trend of using these processors for general computational tasks across the board of computer science. Section 2.2 supports this by reviewing some recent applications in 3D computer vision running on these new GPU platforms.

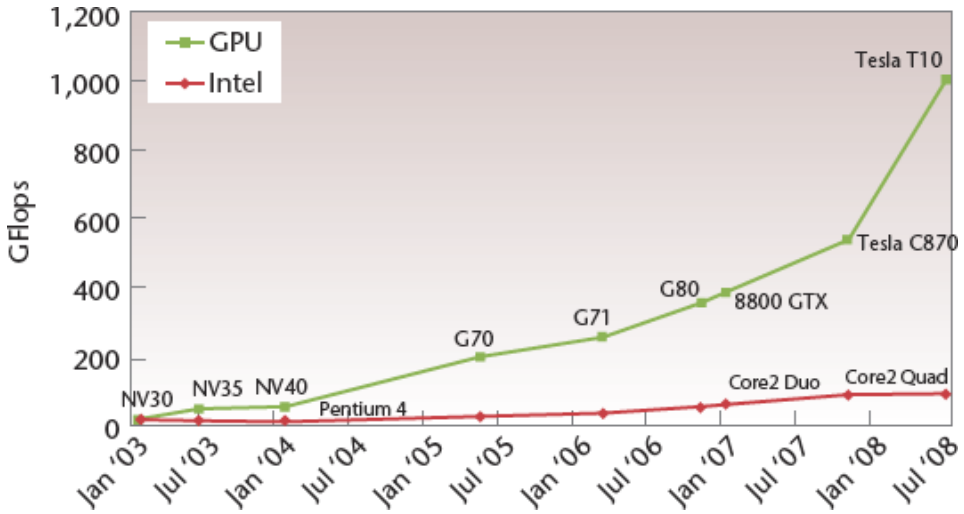

Figure 5.1 Illustrating the performance evolution of two brands of GPUs versus the Intel Pentium 4 CPU

GPUs are massive multi pipeline streaming architectures that take advantage of the SIMD (Single Instruction Multiple Data) type approach similar to the one found in the Intel CPUs. The GeForce 8800 GTX is quoted as having a theoretical maximum performance of 334 Gflops and a memory bandwidth of 86 GB/s versus the Intel Core i7 that can only manage 107 Gflops and has a memory bandwidth of just 26 GB/s. On paper these figures indicate that GPUs have a significant computational performance advantage over general purpose processor. To maximise the performance of GPUs general purpose computation tends to be carried out using fragment shaders as opposed to vertex shaders.

Figure 5.2 GeForce 8800GTX architectural diagram

This approach allows a much greater number of instructions to be performed in parallel. It requires the data to be organised into textures which in turn restricts the way the data is packed. These textures are then streamed and a shader program acts as a kernel performing operations on the stream. The results are then rendered onto another texture and the process can be repeated with a different kernel. This allows feedback, to a limited degree. It is much preferred to write to memory sequentially, as cache misses on

random access incur a very significant performance penalty. Figure 5.2 illustrates the architecture of the GeForce 8800, indicating the various stages of its pipeline.

The purpose of the work presented in this chapter was to implement the space-time stereo algorithm developed by Li Zhang et al.[91] and which was examined in Section 4 on GPU, or more specifically a GeForce 8800 GTX architecture. The aim is to leverage some of the advantages of GPUs that lend themselves particularly well to algorithms and can be made to run in parallel. With performance in mind, the work presented in this chapter was created using the standard OpenGL [76] graphics library with the OpengGL Shading Language (GLSL). All of the work presented in this section was developed with the GeForce 8800 architecture in mind. A more detailed examination of the various shaders and their implementations will then be presented followed by an evaluation of their performance and qualitative results.

5.1 GPGPU OpenGL Framework

All of the work carried out in this section was implemented using OpenGL and GLSL. Most general-purpose computations carried out on GPUs are done on the fragment shaders as they tend to perform better in most situations, being able to handle a greater number of parallel pipelines. The following will examine some of the analogies between a CPU and GPU.

- **Streams GPU Textures = CPU Arrays**
  The fundamental data structures used by fragment shaders on GPUs are textures. Anywhere one would use arrays on a CPU architecture one would use textures on the GPU. This also adds the further restriction imposed by the available texture formats in OpenGL and maximum size and dimensionality.

- **Kernels Fragments Shaders = CPU Inner Loops**
  As opposed to performing instructions on an array by looping through each element as is done on CPUs, the instructions that would be executed inside the loop tend to be implemented in a fragment shader and simultaneously applied to all elements within the texture. The number of possible parallel instructions is limited by the number of fragment shaders available to a particular GPU.

- **Render to Texture = Feedback**

  In order to write the output to another array or texture that can subsequently be used as an input for another kernel one must render the output into a texture. This allows feedback into the next step of the given algorithm. With the recent advance in GPU, multiple render targets are now available allowing up to four textures to be rendered simultaneously.

- **Geometry Rasterization = Computation Invocation**

  In order to invoke a computational kernel on a texture one must typically render a simple geometric primitive such as a quadrilateral polygon onto the screen with the appropriate textures and fragment shaders bound.

- **Texture Coordinates = Computational Domain**

  Generally a kernel takes multiple input streams and generates one output stream. However the computational domain may have a different dimension to the input stream. GPUs provide an easy mechanism to deal with this in the form of texture coordinates. Texture coordinates are specified at each vertex. When the geometric primitive is rendered these coordinates are linearly interpolated for each fragment and passed as an input. One can view these as indices into an array or texture.

- **Vertex Coordinates = Computation Range**

  As the geometry is rasterized, fragments are generated and then processed by the kernel. Typically this is done by rendering a quadrilateral onto the screen, the vertex coordinates therefore directly control the output range of the fragment shaders.

- **Reductions**

  Parallel reductions can be performed very efficiently on GPUs. These can be implemented using two buffers or textures, one is initially bound as an input texture and renders a quadrilateral of the input onto the output texture with linear interpolation enabled. At each pass the output range is divided by some fraction.

The buffers are then swapped and the process is repeated. This is illustrated in Figure 5.3

For a more detailed overview of these analogies as well as various GPU architecture specifics readers are referred to [66].

With these analogies in mind a very simple OpenGL frame work was developed and implemented. A standard GLUT application was used, to initialise OpenGL and all the relevant extensions.

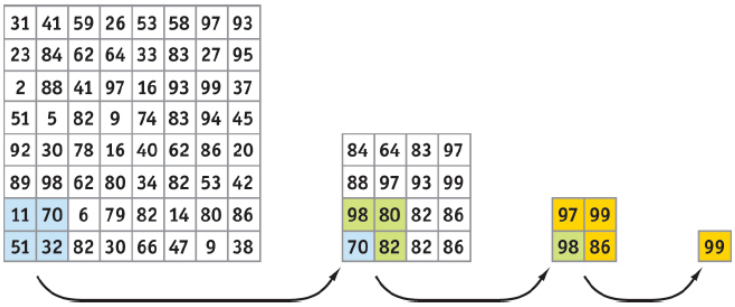

Figure 5.3 Reduction operation performed on a GPU

OpenGL was initialised with an orthographic projection and the relevant call-back functions (i.e. keyboard handler, window resize, rendering loop, etc…) were specified. A GPGPU class was developed, which would handle all the streams (i.e. textures). The class would also compile and bind the relevant fragment shaders and bind all their specific uniform variables. All of the general purpose computations were performed in an update function and the results were be displayed in a draw function. For debugging purposes the ability to dump all render targets in simple csv files that could then be loaded into Matlab to verify results was also implemented.

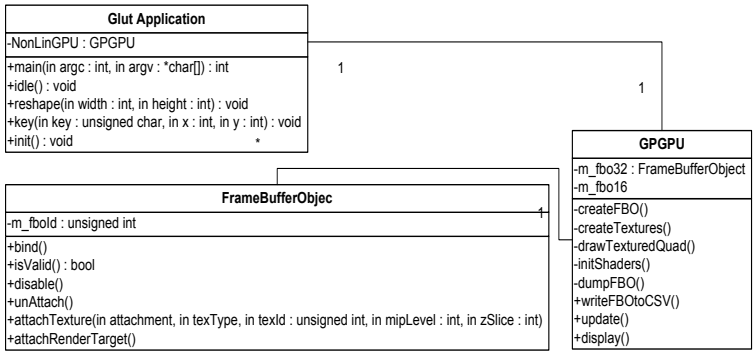

Figure 5.4 UML Class Diagram of GP GPU OpenGL Framework

The GPGPU class also generates fragments and invokes the kernel computations by drawing a quadrilateral of the relevant size. Finally to perform the feedback mechanism a frame buffer object class, also managed by the GPGPU class. More specifics of the implementation will be covered in greater detail in Section 5.3. All the fragment shaders were implemented in GLSL [52], an addition to the OpenGL 2 [63] specification.

5.2 Space Time Stereo GPU Formulation

We now give a high level description of how the space-time stereo algorithm is broken down into various stages that are implemented as various kernels performed on different streams of data. Chapter 4 described the various possible implementations of the space-time stereo algorithm using different non-linear optimization algorithms. For the GPU implementation Equation 5.1 was minimized using the Gauss-Newton algorithm (Equation 5.3), implemented with various kernels on data streams.

Space-time stereo cost function with adaptive window warp function:

$$E(d\ ,d_x,d_y,d_t)=\sum_{t\in I_0}\sum_{(x,y)\in W_0}\left(I_l(x,y,t)-I_r(x-\hat{d}\ ,y,t)\right)^2=\sum_i r_i^2(p)$$

5.1

It is also worth noting that Equation 5.1 is used if one is optimizing a disparity maps for the left to right images. If however, one wishes to optimize a disparity map from right to left images Equation 5.1 can simply be transformed into:

$$E(d\ ,d_x,d_y,d_t)=\sum_{t\in I_0}\sum_{(x,y)\in W_0}\left(I_l(x+\hat{d}\ ,y,t)-I_r(x,y,t)\right)^2=\sum_i r_i^2(p)$$

5.2

$$\hat{d}\ (x,y,t)\equiv d\ +d_x\cdot(x-x_0)+d_y\cdot(y-y_0)+d_0\cdot(t-t_0)$$

5.3

Where the Gauss-Newton equation for parameter update id

$$\hat{d}^{k+1} = \hat{d}^{k} - \left(J_E\left(\hat{d}^{k}\right) J_E\left(\hat{d}^{k}\right)^T \right)^{-1} J_E\left(\hat{d}^{k}\right) E\left(\hat{d}^{k}\right)$$

5.4

where J is the Jacobian of the cost function E which is minimised with respect to the parameters d that include disparity and disparity gradients along the x, y, and t axis. For the GPU implementation a window of size 5 by 5 by 8 was chosen, and these 4 parameters were optimized for each pixel. The solution was initialised with a disparity map created by one of the dynamic program algorithm presented in Chapter 3 of this thesis. The impact of choosing different DP algorithms on the solution will be discussed in Section 5.4. Given a 5 by 5 by 8 window the Jacobian becomes a 200 by 4 matrix for each pixel assuming that each value would be stored in a 32 bit float and the image resolution would be 640 by 480 pixels the total memory foot print for the Jacobian becomes (200 * 4 * 32 * 640 * 480) 937.5 MB which makes it impossible to store the Jacobian explicitly on a GeForce 8800 GTX with 512 MB of RAM. One could potentially use 16 bit floats, thereby halving the memory foot print and making it possible to store the Jacobian explicitly on some more recent GPUs containing 512MB of RAM. However, the computation of the Jacobian with the product of its transpose (i.e. Gauss-Newton approximation to the Hessian) becomes a 4 by 4 matrix and can therefore be explicitly stored on the GPU.

Before giving an overview of the shader framework, it is worth noting how the Jacobian is computed, as this will have an impact on certain design decisions. To quickly reiterate, in Chapter 4, for this particular implementation the approximation to the analytical solution was used as follows.

$$J \equiv \begin{bmatrix} \frac{\partial r_{1,1,1}}{\partial d} & \cdots & \frac{\partial r_{1,1,8}}{\partial d} & \cdots & \frac{\partial r_{5,5,8}}{\partial d} \\ \frac{\partial r_{1,1,1}}{\partial d_x} & \cdots & \frac{\partial r_{1,1,8}}{\partial d_x} & \cdots & \frac{\partial r_{5,5,8}}{\partial d_x} \\ \frac{\partial r_{1,1,1}}{\partial d_y} & \cdots & \frac{\partial r_{1,1,8}}{\partial d_y} & \cdots & \frac{\partial r_{5,5,8}}{\partial d_y} \\ \frac{\partial r_{1,1,1}}{\partial d_t} & \cdots & \frac{\partial r_{1,1,8}}{\partial d_t} & \cdots & \frac{\partial r_{5,5,8}}{\partial d_t} \end{bmatrix}$$

5.5

where

$$\frac{\partial r}{\partial d} \equiv \frac{\partial}{\partial d}\left(I_l(x,y,t) - I_r\left(x + d + d_x(x - x_0) + d_y(y - y_0) + d_t(t - t_0), y, t\right)\right)$$

5.6

Using the chain rule and setting

$$u = x + d + d_x(x - x_0) + d_y(y - y_0) + d_t(t - t_0)$$

5.7

This is equivalent to the warp function in the Lucas and Kanade [51] derivation.

$$\frac{\partial r}{\partial d} \equiv \frac{\partial r}{\partial u} * \frac{\partial u}{\partial d} \equiv \frac{\partial}{\partial u}(-I_r(u,y,t)) * \frac{\partial}{\partial d}(x + d + d_x(x - x_0) + d_y(y - y_0) + d_t(t - t_0))$$

5.8

$$\frac{\partial}{\partial d}(x + d + d_x(x - x_0) + d_y(y - y_0) + d_t(t - t_0)) = 1$$

5.9

and

$$\frac{\partial}{\partial u}(-I_r(u,y,t)) \equiv -\nabla I_r(u,y,t)$$

5.10

Equation 5.8 represents the warped image gradient along the x axis. Therefore:

$$\frac{\partial r}{\partial d} \equiv -\nabla I_r(x + \hat{d}, y, t)$$

5.11

Similarly using the same derivation

$$\frac{\partial r}{\partial d_x} \equiv -\nabla I_r(x + \hat{d}, y, t) * (x - x_0)$$

5.12

$$\frac{\partial r}{\partial d_y} \equiv -\nabla I_r(x + \hat{d}, y, t) * (y - y_0)$$

5.13

$$\frac{\partial r}{\partial d_t} \equiv -\nabla I_r(x + \hat{d}, y, t) * (t - t_0)$$

5.14

As the image gradient of the right images are used to calculate each component of the Jacobian it is more efficient to perform this operation once and store the results in textures that can then subsequently be reused during every iteration of the Gauss-Newton solver.

The Space-Time Gauss-Newton non-linear optimization shader framework was broken down into the following steps as illustrated by Figure 5.5:

1. **Image Gradient Shader**

   This shader would compute the image gradient of the right images using a simple finite difference operation along the x-axis. It would only be run once per solution and not once per iteration.

2. **Hessian Shader**

   This shader would compute the $J_E\left(\hat{d}^k\right)J_E\left(\hat{d}^k\right)$ product from Equation 5.3 and store the result across four RBGA 32 bit floating point textures. This enables the storage of a 4 by 4 Hessian matrix using the RBGA values to store the rows and the multiple textures to store the columns. It would firstly compute the Jacobian followed by the product.

3. **Jacobian Cost Function Product**

   This shader would also perform the same computation of the Jacobian as the previous shader, however it would also compute the cost function specified by Equation 5.1 after doing so it calculates their product which is equivalent to the gradient in the Gauss-Newton (Equation 5.3).

4. **Cholesky Solver**

   Given the nature of the Hessian matrix (i.e. it is a positive semi-definite symmetrical 4 by 4 matrix), the most efficient algorithm for solving Equation 5.3 is Cholesky Decomposition which take advantage of the symmetric nature of the Hessian matrix. This shader implements the Cholesky Decomposition along with the parameter update.

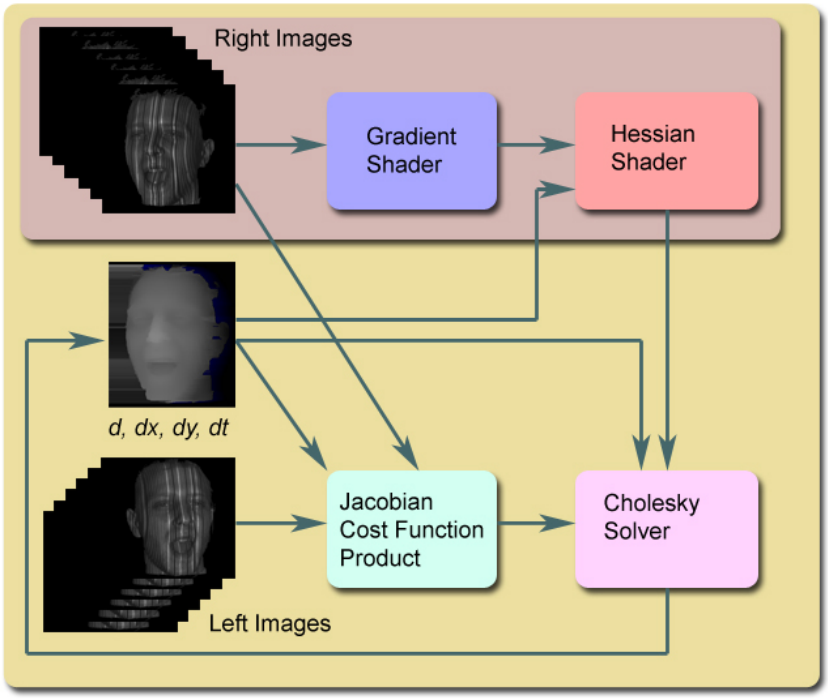

Figure 5.5 Diagram illustrating fragment shaders and data streams

This shader framework constitutes the GPGPU implementation of the Gauss-Newton optimization algorithm with regard to this particular space-time stereo problem and specific warp function. Each of these shaders with the exception of the first gradient shader are run for each iteration of the Gauss-Newton algorithm. The following Section 5.3 will describe the implementation specifics of each one of these shaders as well as the data structure for their various input and output streams. It will be noted that some of these shaders also require more than one rendering pass.

5.3 Shader Implementation Specifics

Having described the overall high level view of the space-time stereo algorithms on the GPU, this section will delve into the specifics of each shader, their implementation the data structures of their input and output streams and some of the limitations and design decision made to try and improve performance.

5.3.1 Gradient Shader

The purpose of this shader is simply to compute the gradient of images along the x-axis. Using a window of 8 pixels in the time domain, the gradient for 8 of the right images,

needs to be computed. Using the advantage of the multiple rendering targets available on the GeForce 8800 one can compute the image gradient of four images at a time. The shader shown in Listing 5.1 was developed in GLSL.

This shader has 4 inputs (i.e. 4 images and return 4 outputs). It not only computes the image gradient along the x axis of the 4 input images but also copies them into the output. In this particular example the images are grey scale, they are copied into the R component of an RGB 16 float texture along with their gradient that is copied into the G component of the output texture. The uniform variable offset is simply there as a scalar that represents the size of a pixel in normalised texture coordinates. This is used to sample the left and right pixels to compute the central differences.

Multiple Image Gradient Shader Listing 5.1

```
static const char *imgGradientSource = {
uniform sampler2D tex0,tex1,tex2,tex3;
uniform float offset;
void main(void)
{
   vec2 texCoord = gl_TexCoord[0].xy;
   vec4 a  = texture2D(tex0, texCoord);
   vec4 b  = texture2D(tex1, texCoord);
   vec4 c  = texture2D(tex2, texCoord);
   vec4 d  = texture2D(tex3, texCoord);

   vec4 aRight  = texture2D(tex0, texCoord+vec2(+offset, 0.0  ));
   vec4 bRight  = texture2D(tex1, texCoord+vec2(+offset, 0.0  ));
   vec4 cRight  = texture2D(tex2, texCoord+vec2(+offset, 0.0  ));
   vec4 dRight  = texture2D(tex3, texCoord+vec2(+offset, 0.0  ));
   vec4 aLeft   = texture2D(tex0, texCoord+vec2(-offset, 0.0  ));
   vec4 bLeft   = texture2D(tex1, texCoord+vec2(-offset, 0.0  ));
   vec4 cLeft   = texture2D(tex2, texCoord+vec2(-offset, 0.0  ));
   vec4 dLeft   = texture2D(tex3, texCoord+vec2(-offset, 0.0  ));

   gl_FragData[0].y = (aRight.x-aLeft.x)*0.5;
   gl_FragData[1].y = (bRight.x-bLeft.x)*0.5;
   gl_FragData[2].y = (cRight.x-cLeft.x)*0.5;
   gl_FragData[3].y = (dRight.x-dLeft.x)*0.5;
   gl_FragData[0].x = a.x;
   gl_FragData[1].x = b.x;
   gl_FragData[2].x = c.x;
   gl_FragData[3].x = d.x;
};
```

5.3.2 Hessian Shader

This shader functions by iteratively calculating the temporary Hessian at a certain point in the space window across all time values simultaneously. Due to certain limitations of GPUs that restrict the maximum number of instructions available to any particular fragment shader, this shader was implemented using multiple passes. The Jacobian maybe reorganised as follows.

$$J \equiv \begin{bmatrix} -\nabla I_r(x_{-2}-d_0, y_{-2}, t_0) & \cdots & -\nabla I_r(x_{-2}-d_0, y_{-2}, t_7) & \cdots & -\nabla I_r(x_2-d_0, y_2, t_7) \\ -\nabla I_r(x_{-2}-d_0, y_{-2}, t_0)(x-x_0) & \cdots & -\nabla I_r(x_{-2}-d_0, y_{-2}, t_7)(x-x_0) & \cdots & -\nabla I_r(x_2-d_0, y_2, t_7)(x-x_0) \\ -\nabla I_r(x_{-2}-d_0, y_{-2}, t_0)(y-y_0) & \cdots & -\nabla I_r(x_{-2}-d_0, y_{-2}, t_7)(y-y_0) & \cdots & -\nabla I_r(x_2-d_0, y_2, t_7)(y-y_0) \\ -\nabla I_r(x_{-2}-d_0, y_{-2}, t_0)(t_0-t_0) & \cdots & -\nabla I_r(x_{-2}-d_0, y_{-2}, t_7)(t_7-t_0) & \cdots & -\nabla I_r(x_2-d_0, y_2, t_7)(t_7-t_0) \end{bmatrix}$$

5.15

$$H = J * J^T$$

5.16

Let J_d be the first row of J and J_{dx}, J_{dy}, J_{dt} be the second, third and fourth, than H becomes:

$$H = \begin{bmatrix} J_d \bullet J_d & J_d \bullet J_{dx} & J_d \bullet J_{dy} & J_d \bullet J_{dt} \\ J_d \bullet J_{dx} & J_{dx} \bullet J_{dx} & J_{dx} \bullet J_{dy} & J_{dx} \bullet J_{dt} \\ J_d \bullet J_{dy} & J_{dx} \bullet J_{dy} & J_{dy} \bullet J_{dy} & J_{dy} \bullet J_{dt} \\ J_d \bullet J_{dt} & J_{dx} \bullet J_{dt} & J_{dy} \bullet J_{dt} & J_{dt} \bullet J_{dt} \end{bmatrix}$$

5.17

If one further breaks down the rows into vectors of 1 by 8, each representing the components of Jacobian at a certain space window index, for all 8 frames in the time domain, it becomes possible to compute the Hessian iteratively for each window index by accumulating the sum of all the dot products. This becomes the basis of the implementation of the Hessian shader listed below. The shader receives as input the previously accumulated Hessians which for the first iteration are set to zero, along with the image gradients and the previously estimated set of parameters used to calculate to warp function. It firsts calculates the Jacobian components at the current space window index across 4 time frames. It then proceeds to calculate the accumulated Hessian estimation of the currently computed Jacobian parameters. This is then repeated for the following 4 time frames at the same window index. Having done so, the current Hessian

estimation is then written out to the multiple render targets across the RBGA 32 bit float components.

In order to create the feedback mechanism, two sets of Hessian textures are created in OpenGL, as the GLSL specification does not allow writing to an input texture. After each iteration these two sets of Hessian textures are swapped (i.e. the previous input Hessian textures become the output stream and the previous output Hessian textures the new input). This is known as ping-pong of textures in the GPGPU community and is used as a feedback mechanism. In this particular instance it feeds back into the same shader for each window index. Using a 5 by 5 by 8 space time window results in performing 25 rendering passes to calculate the Hessian matrix. This approach also has the added benefit of allowing the space window size to be dynamically adapted across the solution or across subsequent solutions. This feature could potentially be used to create elegant degradable solutions based on certain criteria such as computational resources or performance constraints.

The following listing is of the Hessian shader. However, the space-time warp function is not warped in the time domain for clarity.

Hessian Shader Listing 5.2

```
static const char *hessianSource = {
uniform sampler2D tex 0,tex1, tex2, tex3, tex4, tex5, tex6,
tex7,disparity,hess0,hess1,hess2,hess3;
uniform float offsetX, offsetY;
uniform float windowX, windowY;
void main(void)"
{"
        vec2 texCoord = gl_TexCoord[0].xy;
        vec2 warpedTexCoord = gl_TexCoord[0].xy;
        vec4 d = texture2D(disparity, texCoord);
        vec4 H1, H2, H3, H4;
        H1 = texture2D(hess0, texCoord);
        H2 = texture2D(hess1, texCoord);
        H3 = texture2D(hess2, texCoord);
        H4 = texture2D(hess3, texCoord);
        float d0;
        float scaleX, scaleY;
        scaleX = windowX-3.0f;
        scaleY = windowY-3.0f;
        d0 = (d.x + d.y*(windowX-3) + d.z*(windowY-3));
        warpedTexCoord.x = texCoord.x + ((windowX-3)*offsetX) + d0*offsetX;
        warpedTexCoord.y = texCoord.y + ((windowY-3)*offsetY);"
        J1.x = texture2D( left0, warpedTexCoord).y;
```

```
        J1.y = texture2D( left1, warpedTexCoord).y;
        J1.z = texture2D( left2, warpedTexCoord).y;
        J1.w = texture2D( left3, warpedTexCoord).y;
        J2 = J1*scaleX;
        J3 = J1*scaleY;
        H1.x += dot(J1,J1);
        H1.y += dot(J1,J2);
        H1.z += dot(J1,J3);
        H2.x += dot(J2,J1);
        H2.y += dot(J2,J2);
        H2.z += dot(J2,J3);
        H3.x += dot(J3,J1);
        H3.y += dot(J3,J2);
        H3.z += dot(J3,J3);
        J1.x = texture2D( left4, warpedTexCoord).y;
        J1.y = texture2D( left5, warpedTexCoord).y;
        J1.z = texture2D( left6, warpedTexCoord).y;
        J1.w = texture2D( left7, warpedTexCoord).y;
        J2 = J1*scaleX;
        J3 = J1*scaleY;
        H1.x += dot(J1,J1);
        H1.y += dot(J1,J2);
        H1.z += dot(J1,J3);
        H2.x += dot(J2,J1);
        H2.y += dot(J2,J2);
        H2.z += dot(J2,J3);
        H3.x += dot(J3,J1);
        H3.y += dot(J3,J2);
        H3.z += dot(J3,J3);
        H4.w = 1.0f;
        gl_FragData[0] = H1;
        gl_FragData[1] = H2;
        gl_FragData[2] = H3;
        gl_FragData[3] = H4;
};
```

5.3.3 Jacobian Cost Function Product Shader

This shader uses a similar multi-pass rendering approach to the Hessian shader to calculate the product of the Jacobian with the cost function. This product is also broken down into a series of vector dot products performed at each space window index across multiple time frames and iteratively accumulating the result into an RGBA 32 bit texture. This time there is no need to use multiple render targets as the result is a series of 4 component vectors for each pixel that can fit into the RGBA texture components. The feedback is also performed using two ping-pong textures.

It is also worth mentioning that texture lookups in any shader have a certain latency associated with them and therefore whenever possible they are worth reducing to a minimum. The latency of fetching one RGBA texel is significantly lower than that of fetching 4 texels from 4 separate textures, even if each texture only contains one luminance component. It is therefore important to pack information into one texture, as opposed to using multiple textures whenever possible. This combined with the fact that the GeForce 8800 only contains a set number of texture units, makes it often critical to pack textures. As a warped space time window of pixels in the left images with a straight window of pixels in right images is being optimized or vice versa depending on which disparity map is being optimised. It becomes not only possible but advantageous to pack the 8 non-warped images into two RGBA textures. This also reduces the textures fetches and the latency associated with them.

This shader makes use of packing the non-warped images into two textures. The shader also uses the fact that the gradient shader already packed warped images with their gradients used to compute the Jacobian. This shader then has as input 8 textures containing the images to be warped in the R channel along with their gradients in the G channel. It also has two textures containing the packed images that won't be warped. Another input is the current parameters being optimized that are contained in one RGBA texture containing the disparities and their gradients. And finally the last input is used as the accumulation buffer containing the previous estimate the Jacobian cost function product, for a particular window index.

This shader proceeds as follows: firstly it computes the current estimate of the Jacobian for a particular space window index for the first 4 frames. It then computes the cost function for the same window index and the same 4 time frames. This is basically the difference between the warped samples and the packed non-warped samples. The product between the Jacobian and cost function vector is calculated using dot products. This process is repeated for the following 4 frames in time and the results are accumulated to the previous ones using subsequent rendering passes until each index in the spatial window is covered. The following is the shader listing again assuming a quasi-static scene therefore d_t is assumed to be zero.

Jacobian Cost Function Product Shader Listing 5.3

```
static const char *JtFxSource = {
uniform sampler2D left0,left1,left2,left3,left4,left5,left6,left7, rightPack0,
rightPack1, disparity, previous;
uniform float offsetX, offsetY;
uniform float idxX, idxY;
void main(void)"
{
        vec2 texCoord = gl_TexCoord[0].xy;
        vec2 winCoord = gl_TexCoord[0].xy;
        vec2 warpedTexCoord = gl_TexCoord[0].xy;
        vec4 d = texture2D(disparity, texCoord);
        vec4 JtFx, left, right, tmp, cost;
        JtFx = texture2D(previous, texCoord);
        vec4 J1 = vec4(0.0f, 0.0f, 0.0f, 0.0f);
        vec4 J2 = vec4(0.0f, 0.0f, 0.0f, 0.0f);
        vec4 J3 = vec4(0.0f, 0.0f, 0.0f, 0.0f);
        vec4 J4 = vec4(0.0f, 0.0f, 0.0f, 0.0f);
        float d0;
        float scaleX, scaleY;
        scaleX = idxX-3.0f;
        scaleY = idxY-3.0f;
        d0 = (d.x + d.y*(idxX-3) + d.z*(idxY-3));
        warpedTexCoord.x = texCoord.x + ((idxX-3)*offsetX) + d0*offsetX;
        warpedTexCoord.y = texCoord.y + ((idxY-3)*offsetY);
        winCoord.x = texCoord.x + ((idxX-3)*offsetX);
        winCoord.y = texCoord.y + ((idxY-3)*offsetY);
        tmp            = texture2D( left0, warpedTexCoord);
        J1.x    = tmp.y;
        left.x  = tmp.x;
        tmp            = texture2D( left1, warpedTexCoord);
        J1.y    = tmp.y;
        left.y  = tmp.x;
        tmp            = texture2D( left2, warpedTexCoord);
        J1.z    = tmp.y;
        left.z  = tmp.x;
        tmp            = texture2D( left3, warpedTexCoord);
        J1.w    = tmp.y;
        left.w  = tmp.x;
        J2 = J1*scaleX;
        J3 = J1*scaleY;
        right   = texture2D(rightPack0, winCoord);
        cost    = left - right;
        JtFx.x  += dot(J1, cost);
        JtFx.y  += dot(J2, cost);
        JtFx.z  += dot(J3, cost);
        tmp            = texture2D( left4, warpedTexCoord);
        J1.x    = tmp.y;
        left.x  = tmp.x;
        tmp            = texture2D( left5, warpedTexCoord);
```

```
        J1.y    = tmp.y;
        left.y  = tmp.x;
        tmp             = texture2D( left6, warpedTexCoord);
        J1.z    = tmp.y;
        left.z  = tmp.x;
        tmp             = texture2D( left7, warpedTexCoord);
        J1.w    = tmp.y;
        left.w  = tmp.x;
        J2 = J1*scaleX;
        J3 = J1*scaleY;
        right   = texture2D(rightPack1, winCoord);
        cost    = left - right;
        JtFx.x  += dot(J1, cost);
        JtFx.y  += dot(J2, cost);
        JtFx.z  += dot(J3, cost);
    gl_FragData[0] = JtFx;
};
```

5.3.4 Cholesky Decomposition Shader

Having computed the Hessian and gradient (Jacobian cost function product) the last stage of the Gauss-Newton is to update the parameters by solving the set of linear equations presented in Equation 5.3. There are a few different algorithms capable of performing the task. These may include Gaussian elimination, LU decomposition, SVD, etc… However, given the Hessian is a dense positive semi definite symmetrical matrix the most efficient way to solve this set of linear equations is with the Cholesky decomposition algorithm. Similar to LU decomposition the matrix is decomposed into upper and lower matrices. However, with the Cholesky decomposition algorithm the upper matrix is simply the lower matrix transposed, giving:

$$L \cdot L^T = A$$

5.18

$$L_{ii} = \left(a_{ii} - \sum_{k=1}^{i-1} L_{ik}^2 \right)^{1/2}$$

5.19

And

$$L_{ji} = \frac{1}{L_{ii}} \left(a_{ij} - \sum_{k=1}^{i-1} L_{ik} L_{jk} \right)$$

5.20

This algorithm performs a factor of 2 better than LU decomposition. Given that the Hessian is only a 4 by 4 matrix in the case where using a space-time warp and in the case of just using a space warp only a 3 by 3 matrix, the loops were directly unrolled and implemented in a straight forward shader listed below. Readers are referred to [68] for a more detailed derivation of the Cholesky decomposition algorithm, Jung and O'Leary [39] have recently presented an implementation of the Cholesky decomposition algorithm targeting a GPU implementation using a very different approach. Although their implementation is more elegant and possibly superior in performance when dealing with larger matrices, it is targeted at much larger problems solving one system of equations as opposed to this implementation which targets solving lots of very similar systems (i.e. a 4 by 4 system for each pixel). Our shader accepts as input the 4 textures containing the Hessian, a texture containing the current estimate of the optimization parameters, and the texture containing the Jacobian cost function product. The shader then solves the system and updates the optimizations parameters. These are then piped back to the solver and the next iteration is started.

Cholesky Decomposition Shader Listing 5.4

```
static const char *choleskySource = {
uniform sampler2D hess0,hess1,hess2,hess3, JtFx, disparity;
uniform float offsetX, offsetY;
void main(void)
{"
   vec2 texCoord = gl_TexCoord[0].xy;
        vec4 disp = texture2D(disparity, texCoord);
        vec4 b  = texture2D(JtFx, texCoord);
        vec4 x = vec4(0.0f,0.0f,0.0f,0.0f);
        vec4 y = vec4(0.0f,0.0f,0.0f,0.0f);
        mat4 a = mat4(texture2D(hess0, texCoord),texture2D(hess1,
texCoord),texture2D(hess2, texCoord),texture2D(hess3, texCoord));
        vec4 p = vec4(0.0f,0.0f,0.0f,0.0f);
        int i, j, k;
        float sum;
        p.x = sqrt(a[0][0]);
        sum = a[1][0];
        if ( p.x != 0.0f ){
        a[0][1] = sum / p.x;
        };
        sum = a[2][0];
        if ( p.x != 0.0f ){
        a[0][2] = sum / p.x;
```

```
        };
        sum = a[1][1];
        sum -= a[0][1]*a[0][1];
        p.y = sqrt(sum);
        sum = a[2][1];
        sum -= a[0][1]*a[0][2];
        if ( p.y != 0.0f ){
        a[1][2]= sum / p.y;
        };
        sum = a[2][2];
        sum -= a[1][2]*a[1][2];
        sum -= a[0][2]*a[0][2];
        p.z = sqrt(sum);
        if ( p.x != 0.0f ){
        y.x = b.x / p.x;
        };
        if ( p.y != 0.0f ){
        y.y = (b.y - (a[0][1]*y.x)) / p.y;
        };
        if ( p.z != 0.0f ){
        y.z = (b.z - (a[0][2]*y.x) - (a[1][2]*y.y)) / p.z;
        };
        if ( p.z != 0.0f ){
        x.z = y.z / p.z;
        };
        if ( p.y != 0.0f ){
        x.y = (y.y - (a[1][2]*x.z)) / p.y;
        };
        if ( p.x != 0.0f ){
        x.x = (y.x - (a[0][1]*x.y) - (a[0][2]*x.z)) / p.x;
        };
   gl_FragData[0] = disp - x;
};
```

5.4 Experiments

The primary motivation for the GPU implementation of non-linear Gauss-Newton optimization was one of performance. Firstly, the algorithm lends itself very well to being parallelized as each pixel is optimized separately. Secondly this algorithm is not a candidate for real-time implementation on current CPUs. Could this algorithm be made to run in real-time by implementing it on a GPU platform? This section will describe a set of experiments in order to determine the performance of this implementation under varying conditions. Experiments to determine the performance of the full algorithm were carried out as well as variations of the algorithm that were comprised of a few

trade-offs for performance gains, such a constraining the warp function to space parameters, and terminating the Gauss-Newton optimization earlier as well as changing the size of cost function support region.

With these trade-offs being introduced, a second set of experiments were carried out in order to determine the impact in terms of quality these trade-offs would introduce. Other experiments were also carried out to determine the sensitivity of the Gauss-Newton optimization to the initial estimate of the parameters that are determined by DP or potentially other types of correlation based solvers. This section will be divided into two main subsections one outlining all the performance based experiments carried out while the other will present the qualitative based experiments. The results of all these experiments will be presented in the following Section 5.5.

5.4.1 Performance Oriented Experiments

The purpose of the following experiments was to determine the performance of the GPU non-linear optimization part of the space time stereo algorithm. In order to determine how to improve the GPU based implementation of this space-time stereo algorithm the bottlenecks had to be found. The performance in terms of computational time and scalability with regard to the algorithms parameters were found. With these goals each fragment shader was benchmarked individually with different valued parameters as well as the entire optimization iteration. The following benchmarks were performed on the standard rig used in all the previous experiments, namely a quad core Pentium running at 2.6Ghz with a GeForce 8800GTX:

- Benchmarking each fragment shader with different support region window sizes
- Benchmarking each fragment shader on differing image resolutions
- Benchmarking the entire non-linear optimization iteration with different window sizes
- Benchmarking the entire non-linear optimization iteration with differing resolutions
- Benchmarking a differing number of optimization iterations steps.

The results of these experiments are presented in Section 5.5. These results demonstrate the scalability of the GPU non-linear optimization step with regard to image resolution, algorithm parameters and convergence. The image resolution scalability gives one a good idea of the potential effects of using multiple GPUs. However, it does not take into account the data transfer times required to move data across the PCI express bus to multiple GPUs, which are relatively negligible compared to the algorithms overall computational cost.

5.4.2 Qualitative Experiments

Having devised experiments to determine the computational cost of the GPU non-linear optimization step as well as its scalability with regard to certain parameters, such as the number of optimization steps as well as the support region size, it was then possible to find the trade-offs between computational time versus reconstruction quality. Section 5.5 will present the quality of various reconstructions next to their computational performance.

5.5 Results

Table 5.1 presents the timings performed for each fragment used by the GPU non-linear optimization algorithm described in the previous section. This table allows us to chart the scalability of each shader with respect to the window size or number of pixels contained in the support region. It is also worth noting that although the timings for the gradient and Cholesky shader vary when using different window sizes for the solver, these were not due to the actual window sizes themselves. The computational cost of the gradient shader does not vary according to window size. This shader should have a fixed cost irrelevant of the support region size of the solver as it always used the two pixel finite difference method for the computation. The timings difference for this particular shader are less than 0.1 ms and are more likely due to drivers and clock speed variations of the GPU controlled by the driver given different temperatures. These effects also contribute to the Cholesky shader timing variations as this shader is always applied to 3x3 or 4x4 matrix irrelevant of window size. However, there is also the added effect that the Hessian matrix might be ill-conditioned and therefore produce NAN floats or divide by zero conditions that would also slightly affect that particular shader's

performance. As illustrated by Figure 5.1, one can clearly see that the scalability of the Jacobian Cost Function, Gradient and Cholesky shaders with respect to support region size is linear. As expected the Hessian shader as expected does not scale linearly and is also the most computationally expensive shader. This is effectively a matrix product which is known to run in O(n^2) time.

Resolution	Windows Size	3x3x8	6x3x8	9x3x8	12x3x8	9x6x8	12x6x8	9x9x8	12x9x8	12x12x8
Num Pixels In Window		72	144	216	288	432	576	648	864	1152
640x480	Gradient	0.0002	0.00018	0.00017	0.0002	0.00015	0.00018	0.00018	0.00015	0.0002
640x480	Hessian	0.00186	0.00308	0.00423	0.0058	0.00826	0.01143	0.01242	0.02161	0.0385
640x480	JacobCost	0.00073	0.0011	0.00156	0.0019	0.00266	0.00362	0.00399	0.00518	0.0069
640x480	Cholesky	0.00025	0.00032	0.00026	0.0003	0.00029	0.00025	0.00034	0.00025	0.0003

Table 5.1 Time taken in seconds for each shader described in Section 5.4 using different size windows

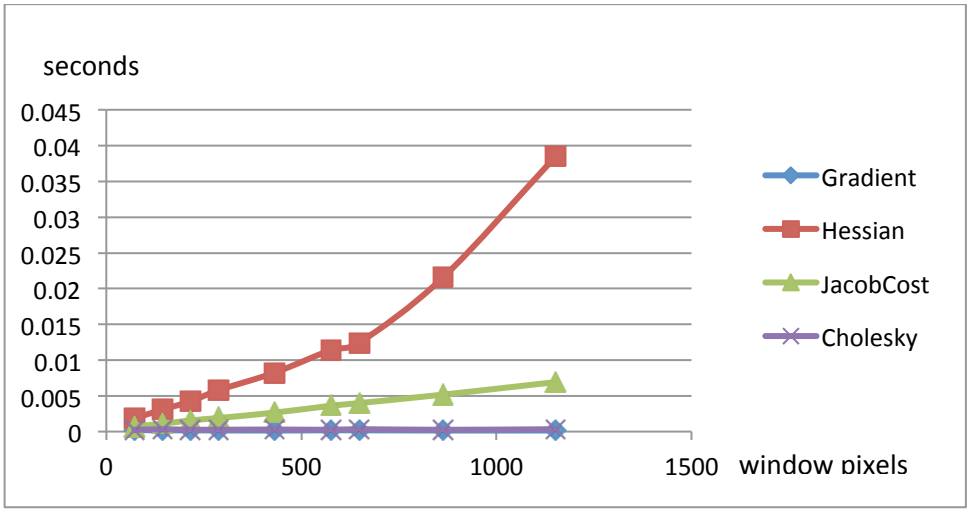

Figure 5.6 Graph showing scalability of each fragment shader with respect to windows sizes

Table 5.1 summarizes the timings of the complete GPU solver for varying numbers of iterations as well as window sizes. This table allows us to plot the relationship and scalability of the complete solver with respect to window sizes as illustrated in Figure 5.6 as well as the scalability with regard to the number of iterations for each set of window sizes as illustrated by Figures 5.8 to 5.12. Although Figure 5.6 clearly indicates that the Hessian shader does not scale linearly with respect to window sizes, when looking at the entire GPU solver timings expressed in Figure 5.7 for the same window sizes with varying number of iteration the scaling of the solver is almost linear. Table

5.2 also demonstrates that the algorithm scales better with regard to window sizes than with regard to number of iterations. This would indicate that if computational scalability is a primary objective, one is better off running the algorithm using a larger support window sizes than more iterations. Figures 5.8 to 5.12 all show the scalability of the GPU solver for each window size with respect to the number of iterations. These are all almost linear with the exception of the smallest window size of 3x3x8. The other observation is that as the window size increases so does the approximation to linear scalability.

Table 5.2 indicates that the computational time of the GPU solver after 3 iterations on a window size of 12x12x8 is similar to computational time after 4 iterations on a window of 12x9x8 and 5 iterations using a window size 9x9x8. There is roughly a 6ms difference between these different parameters. This demonstrates that within certain computational time constraints the algorithm could be run using very different parameters and it is therefore necessary to fine tune the system to produce the best possible qualitative results.

Resolution	Window	3x3x8	6x3x8	9x3x8	12x3x8	9x6x8	12x6x8	9x9x8	12x9x8	12x12x8
	Iterations	72	144	216	288	432	576	648	864	1152
640x480	1	0.0124	0.0139	0.0176	0.019	0.0233	0.0276	0.0313	0.039	0.052
640x480	2	0.0159	0.0223	0.0248	0.028	0.0437	0.0516	0.0615	0.0802	0.104
640x480	3	0.0269	0.0323	0.036	0.042	0.0625	0.0821	0.0938	0.1202	0.155
640x480	4	0.0309	0.0408	0.0446	0.059	0.0846	0.1115	0.1234	0.1598	0.211
640x480	5	0.0369	0.0465	0.0537	0.073	0.108	0.135	0.1535	0.2014	0.262
640x480	6	0.0402	0.0557	0.0694	0.086	0.1282	0.1659	0.1854	0.2369	0.316

Table 5.2 Timings in seconds for complete GPU solver with varying window sizes as well as varying number of iterations.

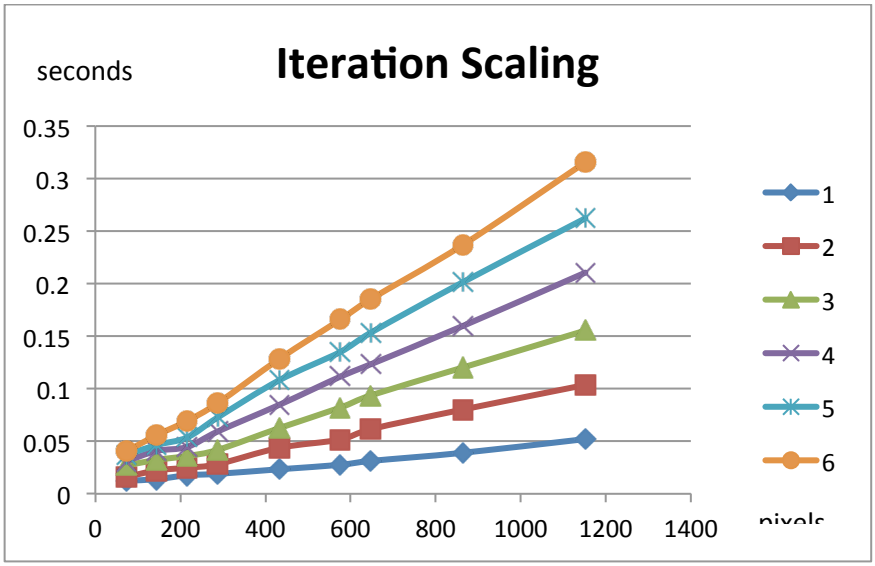

Figure 5.7 Timings for solver using varying number of iterations the x-axis represents the number of pixels contained in the support window and is scaled appropriately

Resolution	**Window**	**3x3x8**	**6x3x8**	**9x3x8**	**12x3x8**	**9x6x8**	**12x6x8**	**9x9x8**	**12x9x8**	**12x12x8**
	Iterations	72	144	216	288	432	576	648	864	1152
640x480	1	80.85	71.85	56.81	52.64	42.9	36.23	32	25.63	19.1938
640x480	2	62.8	44.91	40.28	35.42	22.9	19.38	16.2	12.47	9.64906
640x480	3	37.18	30.98	27.79	23.89	16	12.18	10.7	8.32	6.43571
640x480	4	32.32	24.52	22.42	16.85	11.8	8.968	8.11	6.26	4.74975
640x480	5	27.1	21.51	18.63	13.74	9.26	7.408	6.52	4.965	3.81340
640x480	6	24.88	17.94	14.41	11.58	7.8	6.029	5.39	4.221	3.16852

Table 5.3 Same Timings as Table 5.2 expressed in frames/second

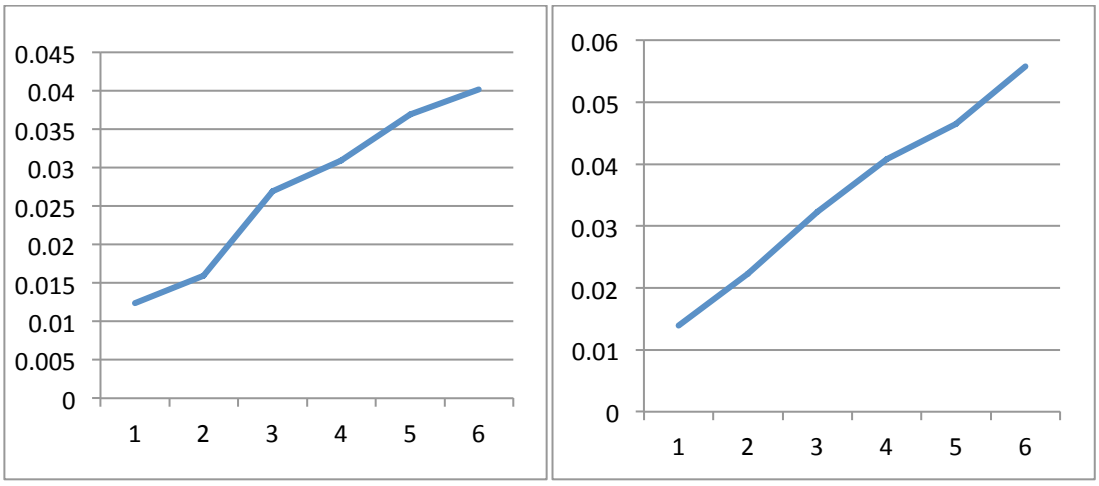

Figure 5.8 Timings for GPU solver after 1-6 iterations for windows (3x3x8 Left, 6x3x8 Right)

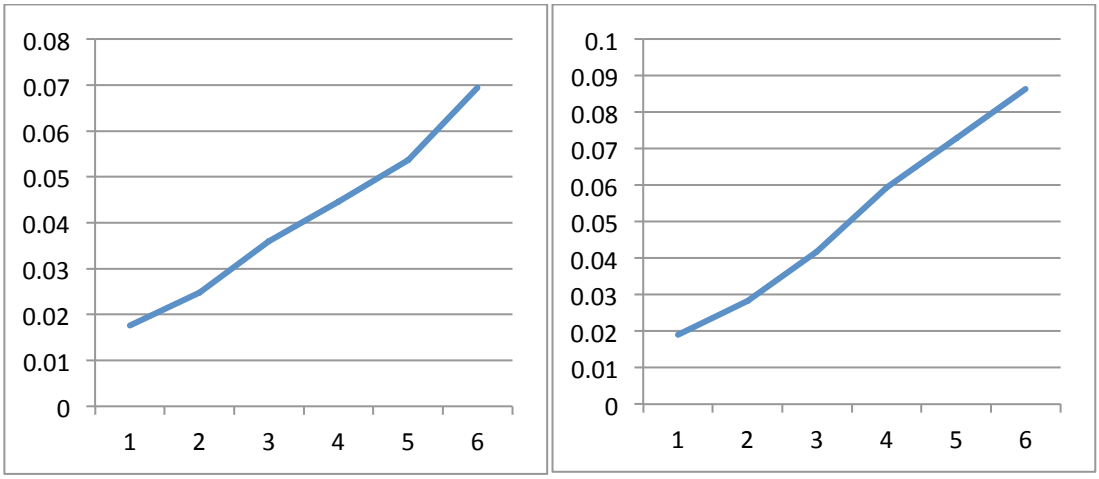

Figure 5.9 Timings for GPU solver after 1-6 iterations for windows (9x3x8 Left, 12x3x8 Right)

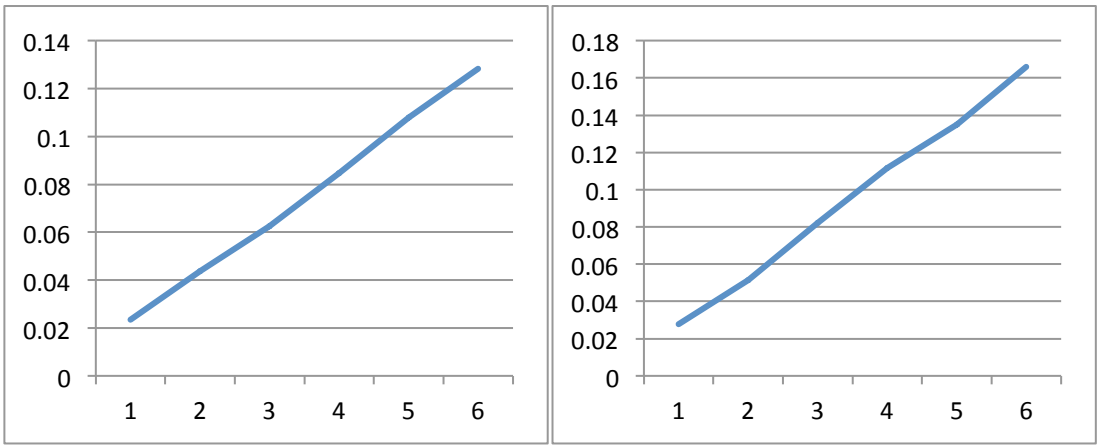

Figure 5.10 Timings for GPU solver after 1-6 iterations for windows (9x6x8 Left, 12x6x8 Right)

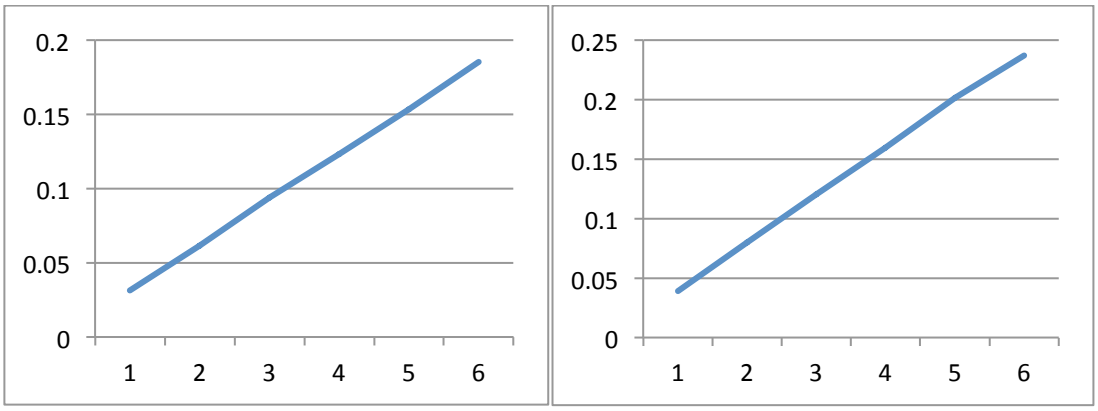

Figure 5.11 Timings for GPU solver after 1-6 iterations for windows (9x9x8 Left, 12x9x8 Right)

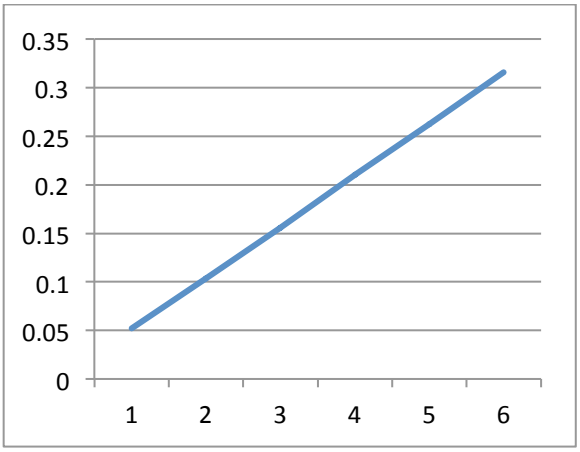

Figure 5.12 Timings for GPU solver after 1-6 iterations for windows (12x12x8)

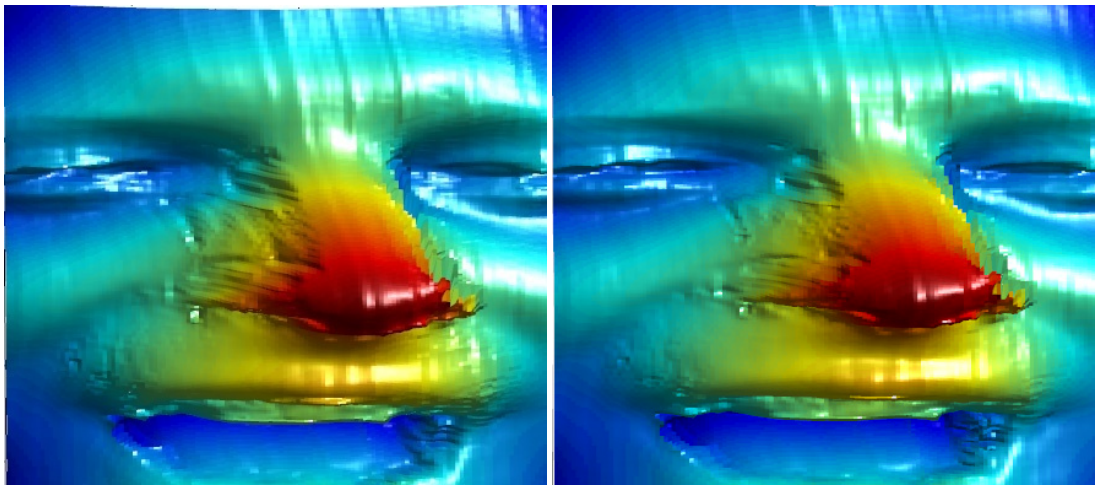

Figure 5.13 Reconstructions using (left 12x12x8 3iterations. right 12x9x8 4 iterations)

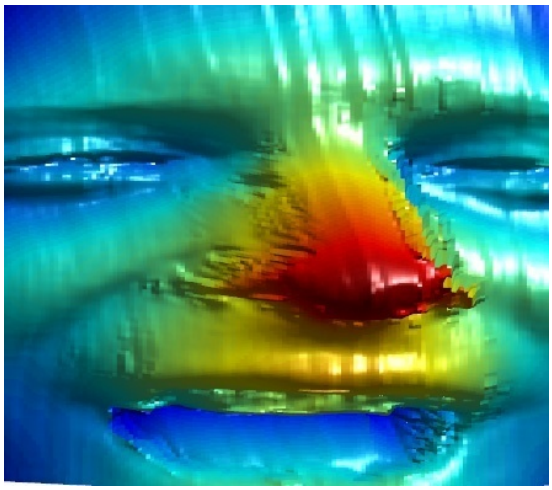

Figure 5.14 Reconstruction using (9x9x8 5 iterations)

Figure 5.2 demonstrates that choosing between window size and number of solver iterations is not always straight forwards. The figure illustrates three reconstructions all using various windows sizes as well as differing number of solver iterations. These three particular sets of parameters were chosen because their computational time is very similar (less than 6ms). Looking at these three images one deduces that using a 12x9x8 window and 4 iterations produce superior results than using 12x12x8 after 3 iterations (i.e. there are fewer artefacts in the prior). However, when comparing the 12x9x8 with 4 iterations with the 9x9x8 with 5 iterations it is not as clear which of these two examples produce superior results. The 9x9x8 reconstruction contains fewer artefacts around the mouth region but the forehead contains more artefacts. The forehead artefacts are caused by the fact that this GPU solver does not enforce a gradient disparity constraint. This could be potentially eliminated at the cost of computational performance, and this will be elaborated in Chapter 6.

5.6 Conclusion

This chapter has demonstrated how to reformulate the algorithm from Chapter 4 into a streaming algorithm which lends itself particularly well to be implemented on modern GPUs. We extended the work of [91] by initializing the algorithm using a modified multilayer dynamic approach and a lower scale and solving the non-linear optimization with a Cholesky solver. All of these are achieved on the GPU. It has also been demonstrated to run in real-time given certain parameters. This chapter has also demonstrated that the GPU implementation of the non-linear optimization algorithm scales close to linearly with regard to the support region window sizes as well as the number of solver iterations. Although this chapter addressed scalability with regard to window sizes and solver iterations, it has not addressed scalability with regard to image resolution and with regard to the overall system including the multi-scale dynamic programming initialisation step. These issues will be tackled in following the following chapter which will explore a scalable frame-work for the overall system. It will address certain issues with regard to parameter tuning for maximum quality results given varying computational time constraints.

Chapter 6

Scalability and Optimization

Having examined the initialization and non-linear optimization steps in the previous chapters, this chapter will discuss further optimization in the context of a scalable framework for real-time applications. The performance of the DP algorithm initialisation step although considered to be potentially interactive, fell short in terms of real-time performance when compared to the more optimized non-linear step described in Chapter 5. This chapter will show how to improve the computational efficiency of the initialization step of the algorithm by implementing a hybrid CPU/GPU implementation in Section 6.1, and how it can further be optimized to achieve real-time performance in Section 6.2 This chapter will also demonstrate the full scalability of all the system parameters in Section 6.3. and place them into the greater context of the scalable framework. The quality performance trade-offs of the system parameters presented in Section 6.4 will be determined by experiments described in Section 6.5, with the results being discussed in Section 6.6.

6.1 Dynamic Programming Hybrid CPU-GPU

The design decision to implement a hybrid approach to solving the Criminisi et al.algorithm was motivated by the fact that the inherent branching nature of dynamic programming does not lend itself well to GPU architectures. This was demonstrated by [27] in which the hybrid CPU-GPU implementation with the significant limitations of an APG bus ran significantly faster than the GPU only implementation. This algorithm was re-implemented to use the CPU and GPU in a hybrid fashion. The target platform was a PC with an Nvidia 680i chipset containing an Intel QX6700 CPU with a GeForce 8800GTX GPU. The CPU is a quad core architecture containing four processor cores that can be made to run in parallel using multiple threads. The GPU is an Nvidia

architecture using 128 unified shaders, making it ideal for processing large streams of data. The Nvidia 680i chipset also contains 3 PCIx16 interfaces allowing up to three GPUs. The reason not many hybrid CPU/GPU implementations of algorithm were performed in the past was because of the AGP interface having a slow read back performance. This interface was asymmetrical and was capable of writing data to the GPU at a much faster transfer rate than reading data. This made reading back large amounts of data from the GPU slow and created an undesirable performance bottle neck. With the new introduction of PCIx16 this no longer becomes such an issue as it is a symmetrical bus with data rates of 3GB/s using non-pageable memory transfers that are optimized by the driver. These new hardware capabilities motivated the development of this new hybrid implementation of the Criminisi et al.[19] algorithm.

The Criminisi et al.[19] algorithm can be broken down into two stages. The first stage is the computation of the cost function to produce a dissimilarity matrix. The second stage is the scan-line optimization performed using the three plane matrix as described in Chapter 2 In this hybrid implementation the first stage in performed on the GPU while the second on the CPU. The captured images are sent across the PCIx16 bus to the GPU. The GPU then computes the cost function for each pixel and each disparity value within a given range. The result is stored in the dissimilarity matrix (DSI matrix) that is then copied back across the PCIx16 bus into system ram that can be accessed by the CPU cores. To further optimize the computation of the DSI matrix on the GPU, this computation is performed again, in two stages. The first stage is computing the cost function in the space domain across two stereo images, with the result then being cached on the GPU. The second stage takes the previously cached results and computes the total cost across the temporal domain. This enables redundant computations to be eliminated.

The high level overview of the hybrid system is illustrated in Figure 6.1. The GPU implementation of the DSI matrix computation is further elaborated in Figure 6.2. This illustrates the GPU implementation of the sum of squared difference cost computation. This implementation was broken down using three computational kernels. The first kernel labelled as the difference kernel in Figure 6.2 computes the difference between each pixel in the left image with each pixel in the right image for every possible disparity value in a given range, usually determined by the base line of the stereo setup. The result producing a 3D data structure of dimensions image width by

image height by disparity range is then cached in the GPU memory along with the same computation of x number of previous frames where x represents the window size in the temporal domain.

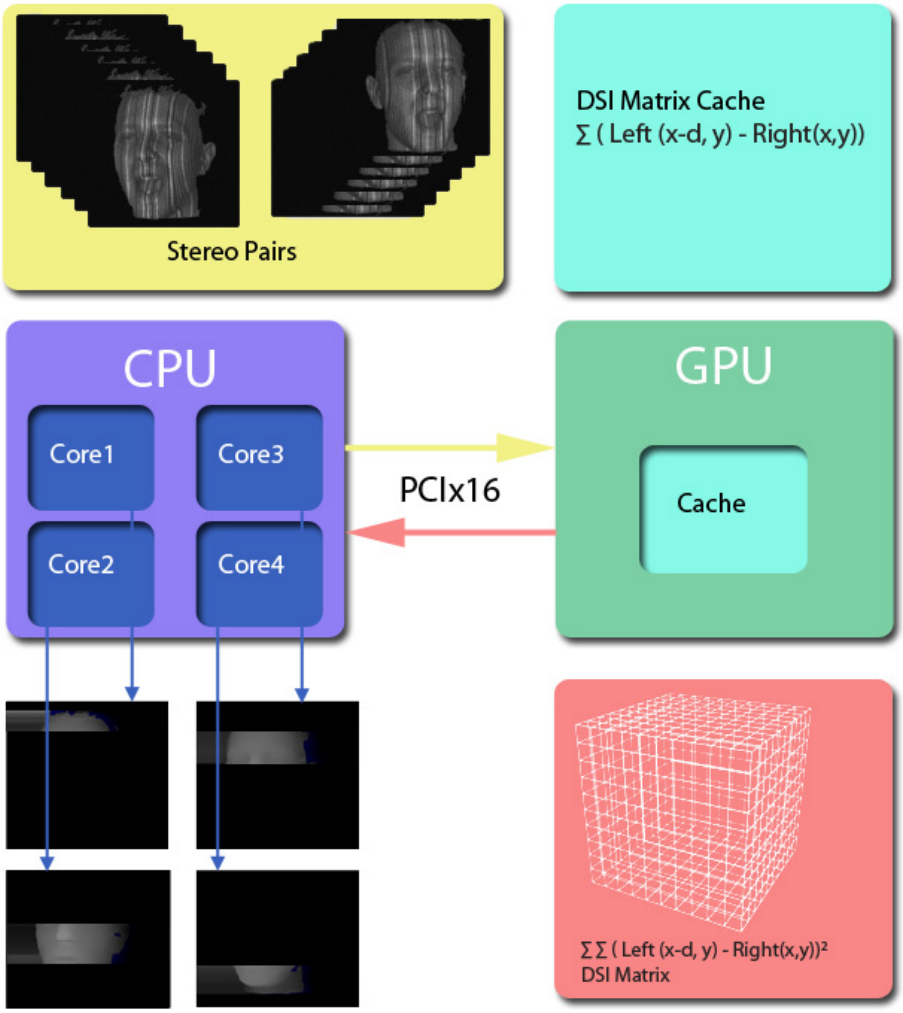

Figure 6.1 Overview of the hybrid CPU/GPU implementation

The second kernel takes the cached results squares them and sums them across the temporal domain. These results are then passed onto the final kernel that performs a sum across the space domain depending on the window size specified to produce the final DSI matrix that is then copied back across the PCIx16 bus into system RAM.

The only issue is the PCIx16 bandwidth of 3GB/s, given that the image resolutions are 640x480 pixels and the typical disparity range is 150 pixels. Using 32bit floats to store the DSI matrix produces a data rate of 176MB per frame. This would give us a theoretical maximum frame rate of 17fps. This can be doubled just by using int16

instead of floats to store the DSI matrix. However, as Chapter 4 demonstrated that due to the fact that the non-linear optimization can be initialized with a sub-sampled DP solution in this way, was deemed unnecessary. This implementation can also be improved by using multiple GPUs each on their own PCIx16 bus therefore dividing the bandwidth requirements by the number of GPUs. As demonstrated in Chapter 4 and Chapter 5, a new multi-scale non-linear optimization algorithm will be developed using the optimized version of this implementation as an initialization step.

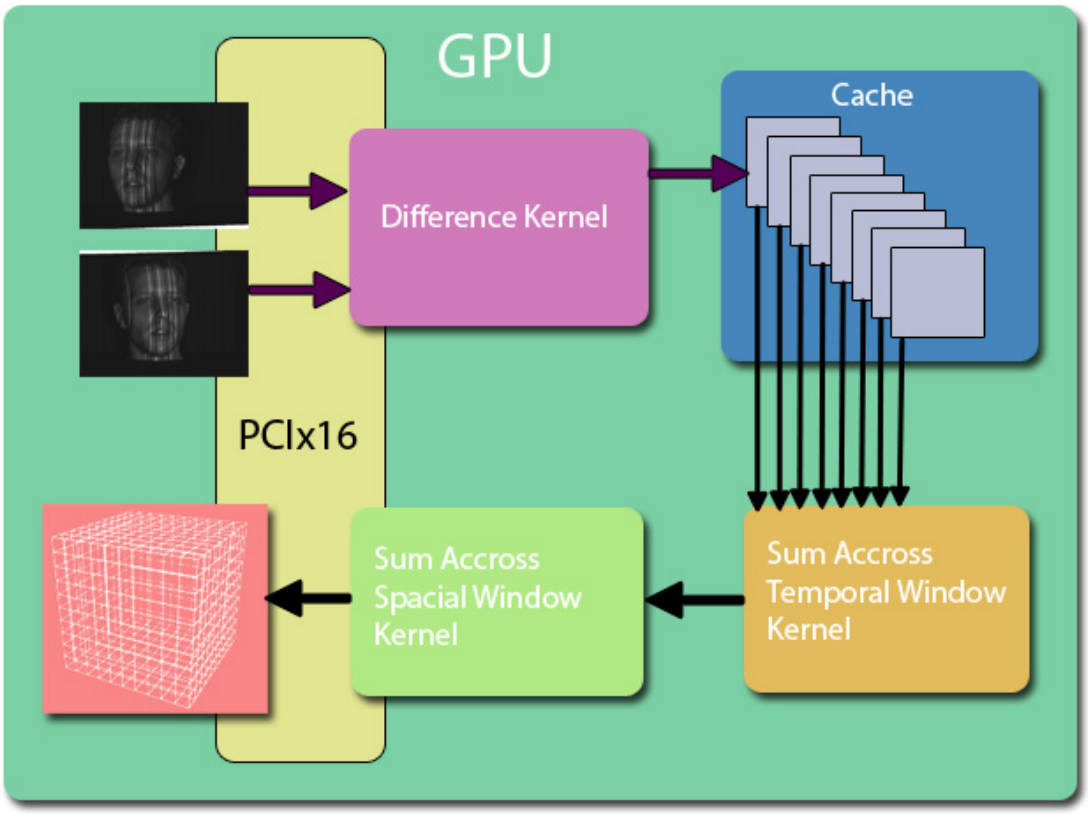

Figure 6.2 GPU DSI Matrix Computations

The initialization will then be performed at half resolution over half the disparity range and will completely eliminate this bottleneck.

The GPU implementation of the cost computation was implemented using CUDA [25] an API, and run time environment developed by Nvidia. This development environment consists of a pseudo C compiler that generates its own machine byte code that is then interpreted by a virtual machine residing inside the GPU's graphics driver. This was specifically developed to enable GPUs to be used for non-graphics related tasks and exposes extra functionality such as memory scatter operations as well as

creating a higher level interface and development environment. The GPU implementation was then validated against the CPU implementation and found to produce identical results.

6.1.1 Performance for Real-Time Applications

Having determined that the Criminisi et al.[19] algorithm produced superior results with fewer errors, it was necessary to answer the remaining question as to its suitability for real-time applications. The initial CPU implementation, although sufficient to evaluate the qualitative results, was not optimized and took a few seconds to calculate the disparity maps. This motivated the hybrid implementation described in Section 6.1. This implementation was benchmarked to determine its speed as well as the impact of various parameters such as window size, maximum disparity threshold and image resolutions. This enables a performance versus quality correlation to be determined, as well as finding the various bottlenecks in order to further improve the algorithms and refine them.

There are a number of parameters each affecting the computational performance of this hybrid CPU-GPU implementation. The benchmarking is achieved by measuring the time taken to perform each of the following tasks: computing the difference between two images for each disparity within a range (this represents the difference kernel), squaring and summing these results across the space and time domain (this represents both SSD kernels), transferring the resulting DSI matrix across the PCIx16 bus and finally, computing the dynamic programming optimization using the Criminisi et al.[19] algorithm on the CPU. Image resolution, disparity range and window size all have an impact on one or more of the previously mentioned sub-components of the algorithm. The image resolution obviously has the greatest impact on performance, by halving the resolution not only are there four times fewer pixels to process, but the disparity range is also divided into two, further more reducing the computational burden, and bandwidth requirements across the PCIx16 bus. Table 6.1 summarizes the time taken in each sub-component with the various parameters. All the timing measurements were performed on an Intel QX6700 clocked at 2.6 GHz with 2GB of ram clocked at 800 MHz and a GeForce 8800GTX clocked at 600 MHz

Image Resolution	Cores	Disparity Range	Window Size	Diff Kernel	SSD Kernels	PCIx Transfer	DP Optimization	Fps
320x240	1	70	3x3x1	0.0103	0.0125	0.0125	0.171	4.8473
320x240	4	70	3x3x1	0.0104	0.025	0.0124	0.044	10.893
320x240	1	70	5x5x1	0.0104	0.0304	0.0123	0.167	4.5434
320x240	1	70	5x7x1	0.0104	0.0304	0.0123	0.167	4.5434
320x240	1	70	5x7x8	0.0104	0.185	0.0124	0.171	2.6399
320x240	4	70	5x7x8	0.0104	0.185	0.0123	0.042	4.0048
640x480	1	120	3x3x1	0.064	0.186	0.08	1.248	0.6337
640x480	4	120	3x3x1	0.064	0.185	0.08	0.32	1.5408
640x480	1	120	5x5x1	0.064	0.219	0.083	1.229	0.627
640x480	1	120	5x7x1	0.064	0.219	0.082	1.225	0.6289
640x480	1	120	5x7x8	0.064	1.338	0.081	1.215	0.3706
640x480	4	120	5x7x8	0.064	1.341	0.081	0.306	0.558

Table 6.1 Benchmarks for hybrid CPU/GPU Dynamic Programming Implementation

From Table 6.1 one can observe that the window size does not affect the performance of the difference kernel, dynamic programming and PCI transfer times. It does however have an impact on the SSD kernels. This is generally due to the fact that although GPUs process large amounts of data simultaneously, there is a large latency penalty for memory fetches, the impact of which can be reduced by having a large ratio of computations per fetched data element. Section 6.2 will demonstrate how this can further be reduced in CUDA [25] with the use of what is referred to as shared memory (effectively a cache inside kernels). One can also observe from this table and Figure 6.3, that the two major computational burdens lie with the dynamic programming optimization performed on the CPU and the SSD computations performed on the GPU. Figure 6.3 is a plot of the computational time of both SSD and DP computations relative to the disparity ranges. One can observe a linear relationship between both the computation time taken for both the GPU and CPU versus the disparity range up to 110. Beyond this point the CPU computation times continue to increase in a linear fashion while the GPU computation times stop increasing. This was a strong indication of a hidden bottleneck in the SSD GPU implementation. This is caused by memory fetches. As the number of computations increase beyond a certain point the GPU's driver's internal scheduler manages to limit the impact of memory fetches and the computational time of the SSD actually decreases. In this particular implementation the SSD computation was performed explicitly in a CUDA [25] kernel. The implementation of the SSD computation again was optimized further and is presented in the following Section 6.2.

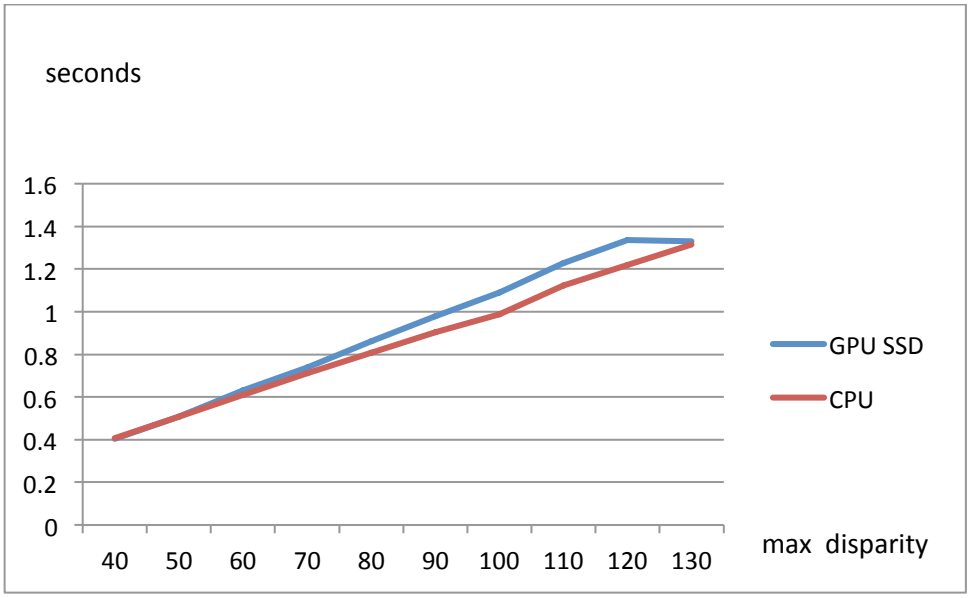

Figure 6.3 Computational Time of GPU/CPU for various disparity ranges for 640x480 5x7x8 window running on single GPU and single Core

The transfer times across the PCIx16 bus were measured for 32bit floating point values. These times can also be easily halved just by using int16. However once a multi-scale approach is applied this is no longer necessary. Further optimizations are achieved with the use of background removal achieved by using a threshold. Another form of performance optimization would also be achieved by assuming a disparity gradient constraint and limiting the disparity range for a given pixel based on the previous frames value and depending on whether or not it was near a depth discontinuity. This would limit the search space of the Criminisi et al.[19] algorithm on a per pixel basis and given that most disparity values fall well beneath the greatest disparity range should in theory speed up the DP optimization times considerably. This optimization was implemented by using a multi-scale approach. The lower resolution solution was used to constrain the maximum disparity values for each pixel. This achieved considerable speed up and will be discussed further in Section 6.2.

All of these optimizations were explored further and used to achieve considerable speed increases, making this algorithm very suitable for real-time applications. They are discussed in greater detail in Section 6.2. As it stands this current implementation scales quite nicely and can be trivially extended to support more CPU cores and multiple GPUs.

6.2 Further Optimizations

The Criminisi et al.[19] dynamic programming algorithm modified to use a space-time window as well as SSD cost function presented in previous Section 6.2 was slow compared to the GPU non-linear optimization step presented in Chapter 5. The optimizations presented in this chapter were motivated by the advantage of using the Criminisi et al.[19] dynamic programming algorithm as well as the normalized SSD cost function extended into the time domain when using striped structured light patterns. Unfortunately dynamic programming type algorithms do not map particularly well to being implemented on GPUs, as seen in [27]. Although they are easy to parallelize by their nature, due to fact that each scan-line is solved independently of each other, while solving a particular scan-line, there is a lot of interdependency within each scan-line combined with heavy branching. In spite of the hybrid CPU/GPU implementation proving beneficial and improving the computational time over the CPU only implementation, there was still room for further optimizations and computational performance gains. This section will discuss in more detail these optimizations and their benefits. All performance timings presented in this section were taken on QX6700 Intel CPU and a GeForce GT260 GPU, although the GPU differs from the 8800 GTX used in the previous chapter, it shares a similar architecture but with the added benefit of an increased memory bandwidth and more shader units.

6.2.1 GPU Read back Optimizations

Theoretically achieving the same bandwidth for reading and writing operations, although in practice this is not truly accurate, the difference in bandwidth between writing and reading across the bus is significantly reduced. The bandwidth requirements for this application are still high, and can be computed as:

$$B = Width * Height * MaxD * sizeof(float) \quad (6.1)$$

Where B is the read back bandwidth in bytes required per frame and MaxD is the maximum disparity value. A real world example would be using half the image resolution (320x240) for the dynamic programming with a maximum disparity value of

80 pixels, which would give a bandwidth requirement of 26.37 Mbytes per frame, with a desired frame rate of 30 fps. For real-time applications or ideally 60fps to match the video camera capture rate a bandwidth of 792MB/s or 1584MB/s respectively would be required. The theoretical maximum bandwidth of PCI express 16 Version 2 is 8GB/s, however, the CUDA runtime environment does not achieve this performance. It was therefore necessary to determine the real world performance of these memory transfers. The initial release of CUDA did not support page locked memory transfers that are optimized by the GPU drivers. With later releases this feature was added, and by using memory transfers from system memory that is paged locked, significant performance increases were achieved.

Maximum Disparity	**Width**	**Height**	**50**	**60**	**70**	**80**	**90**	**Locked**
Bytes Transferred			15360000	18432000	21504000	24576000	27648000	
Transfer Time in seconds	320	240	0.01380	0.01239	0.01406	0.01600	0.01775	No
Transfer Time in seconds	320	240	0.0054	0.00635	0.00728	0.00823	0.00902	Yes
Bandwidth Achieved MB/s	320	240	1061.09	1418.16	1458.17	1464.38	1485.22	No
Bandwidth Achieved MB/s	320	240	2712.67	2768.20	2816.62	2845.04	2920.27	Yes
Maximum Disparity			**100**	**140**	**150**	**200**	**250**	
Bytes Transferred			30720000	43008000	46080000	61440000	76800000	
Transfer Time in seconds	320	240	0.01967	0.02717	0.02832	0.03822	0.04802	No
Transfer Time in seconds	320	240	0.01003	0.01344	0.014651	0.018362	0.023461	Yes
Bandwidth Achieved MB/s	320	240	1489.41	1509.42	1551.248	1533.025	1525.148	No
Bandwidth Achieved MB/s	320	240	2918.30	3051.53	2999.475	3191.033	3121.869	Yes

Table 6.2 Timings for memory transfer across the PCIx16 bus in seconds for different sized cost matrices

Table 6.2 Illustrates the difference in bandwidth achieved in CUDA between using page locked memory transfer versus non-page locked transfers. In each case the bandwidth is doubled. This table also illustrates some differences in bandwidth when transferring varying amounts of data. When transferring larger chunks of data the bandwidth increases slightly up to a point and then starts to decrease once the data starts to exceed 60MB per frame mark. Finally, one also notices that the maximum bandwidth achieved is just over 3GB/s a figure significantly lower than the theoretical maximum of the PCI express 16 Version 2 specification of 8GB/s. This has certain implications that will be discussed in the scalability framework.

6.2.2 CUDA Latency Overhead

The initial CUDA implementation of the cost matrix computation described in Chapter 3 was sub-optimal. CUDA works by providing a high level programming interface similar to C for GPGPU computations. However in its effort to make development easy it is sometimes less obvious how to optimize the code in the best possible way. CUDA organises the data into streams and then launching kernels that process the data on the GPU. It is a massively parallel system, running blocks of threads that are organised into a grid. Each thread executes the kernel over the data stream, and inter thread communication is achieved with the use of shared memory. This shared memory is shared across a particular block of threads. One of the biggest potential bottle necks in CUDA applications is caused by the latency associated with accessing the GPU system memory. This latency is very high and can be in the order of 600 clock cycles. In order to hide this latency the system architecture schedules multiple blocks of threads onto the same hardware shading processor during thread synchronisations calls. Each shading processor has a limited number of registers and shared memory available, this along with kernel and block size, determine the maximum number of blocks that concurrently run on one multiprocessor (i.e. occupancy). To properly reduce the impact of latency associated with global memory reads this occupancy should be a 100% of the maximum supported for that particular GPU architecture. Reducing the number of registers used by a kernel as well as increasing the number of thread blocks can improve the occupancy and therefore offset the latency cost associated global memory access. Using the latest CUDA profiling tools enabled the fine tuning of these various CUDA parameters such as block size, grid size and register usage, to minimize latency associated with global memory operations. Another major source of latency is associated with launching a particular kernel on a grid of thread blocks. The initial implementation would launch a separate kernel for each disparity values for both the DSI and SSD kernels. This is effectively the equivalent of a for loop contained outside the GPU. By placing this loop inside the kernels themselves extra computations used in the kernels was massively offset by the gains in kernel execution latency. This small change probably had the most impact in terms of GPU performance as demonstrated by Table 6.3. and Table 6.4.

Width	Height	winX	winY	winZ	MaxD	Diff Kernel	SSD Kernel
320	240	3	7	1	50	0.00842	0.011751
320	240	3	7	1	60	0.016857	0.013606
320	240	3	7	1	70	0.011281	0.016009
320	240	3	7	1	80	0.012621	0.018152
320	240	3	7	1	90	0.015861	0.020643
320	240	3	7	1	100	0.01649	0.022656
320	240	3	7	1	110	0.018526	0.024946
320	240	3	7	1	120	0.019156	0.027282
320	240	3	7	1	150	0.024682	0.034117
320	240	3	7	1	200	0.035264	0.04514
320	240	3	7	1	250	0.039923	0.056489

Table 6.3 Computational Time of kernels used for the CUDA cost matrix calculation with the added latency of extra kernel calls

Width	Height	winX	winY	winZ	MaxD	Diff Kernel	SSD Kernel
320	240	3	7	1	50	0.000544	0.003823
320	240	3	7	1	60	0.000596	0.004384
320	240	3	7	1	70	0.000641	0.004974
320	240	3	7	1	80	0.00069	0.005498
320	240	3	7	1	90	0.000728	0.005904
320	240	3	7	1	100	0.000778	0.006739
320	240	3	7	1	110	0.000822	0.007006
320	240	3	7	1	120	0.001333	0.007813
320	240	3	7	1	150	0.001003	0.008856
320	240	3	7	1	200	0.001697	0.012881
320	240	3	7	1	250	0.00192	0.016529

Table 6.4 Computational Time of kernels used for the CUDA cost matrix calculation without the added latency of extra kernel calls

Table 6.3 contains the timings of both the difference and SSD kernels with the added overhead latency associated with launching a separate kernel for each of the different disparity values. Table 6.4 contains the timings of the same kernels in Table 6.3. However they contain an extra for loop allowing the removal of a separate execution for each differing disparity value. One can clearly see large performance gains in the region of one order of magnitude (20x speed increase on average), for the latter implementation. This might not have necessarily been the case had this been implemented in OpenGL, where although there is a cost associated with switching

fragment shaders in this particular case one is not switching shaders but would be performing more rendering passes with a much less severe impact on the performance.

Having removed the latency associated with kernel executions, we can minimise the latency associated with memory accessed by the kernel. The shared memory addressing in the GPU architecture used here is broken down into what is referred to as banks (see [60]). These 32bit banks are broken down into half warps with the number of banks per warp determined by the architecture, in this case 16.

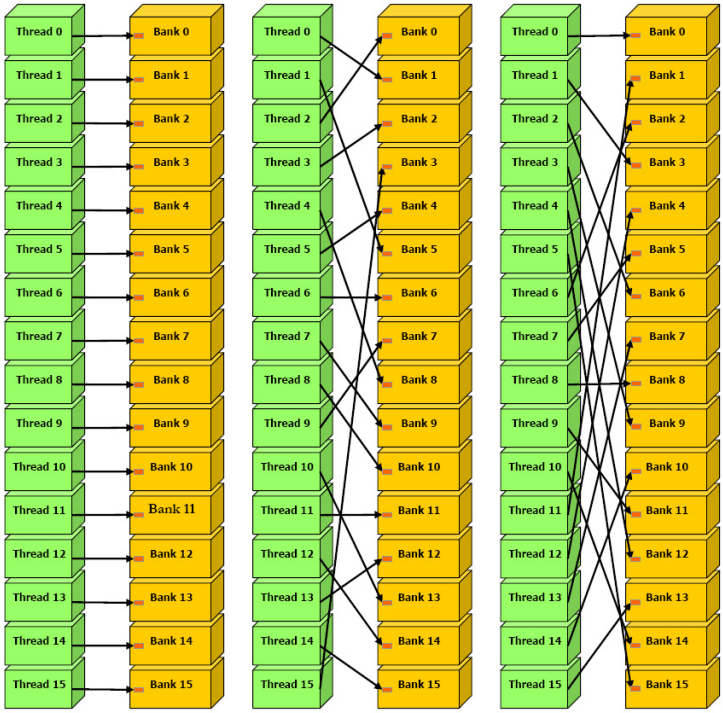

Figure 6.4 Shared Memory Access without Bank-Conflicts

Should sequential threads access the same bank of shared memory, a bank conflict arises and the memory accesses are then serialized and not performed in parallel. In the example of the difference kernel that computes the difference between images for a given disparity value, each pixel is read as an 8bit grey scale unsigned char. The initial implementation had each thread stored and accessed each image pixels in shared memory as unsigned char sequentially, and therefore produced bank conflicts as groups of four pixels were assigned to the same bank. This ended up causing the GPU to serialize each of the shared memory accesses into four separate operations. These serializations are easily removed by either using float arrays in shared memory, or by re-implementing the kernel in such a way that each sequential thread accesses every

fourth byte in shared memory. Both the difference and SSD kernels were re-implemented in such a way as to remove all unnecessary bank conflicts and therefore optimize all shared memory operations.

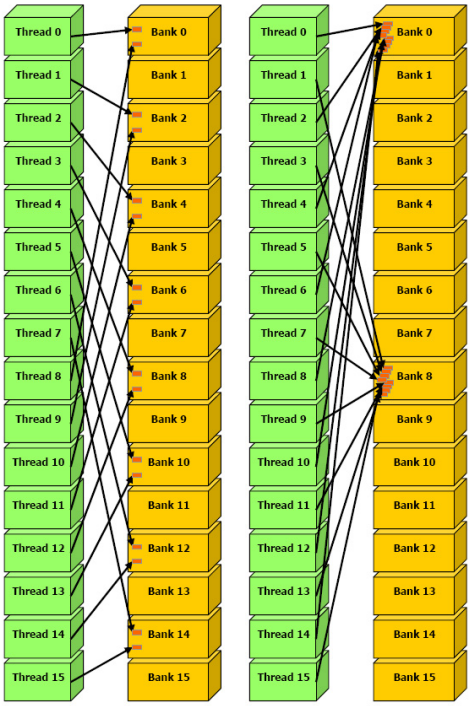

Figure 6.5 Shared Memory Access with Bank-Conflicts

6.2.3 SSD Kernel Optimizations for better scalability

The final GPU optimization targeted the SSD kernel. The initial implementation although optimized to remove all unnecessary bank conflicts and make use of shared memory to minimise the impact of global memory access latency, still performed unnecessary computations. The purpose of the kernel is to compute the sums across the support region from the squared difference data computed by the diff kernel. The initial implementation had each thread loop over each squared difference pixel in the support region compute the sum. Although implemented simply and performing well for small support regions this implementation does not scale well. In the case of a window of 3x3 each thread will perform 8 additions 5 of which will be recomputed by the neighbouring thread, see Figure 6.6.

A more efficient way to compute the results is to separate the sum into two sums. Firstly the columns are summed and then the result is then summed, therefore each thread is now computing 4 additions. This can then further be made more efficient

so that when the next row is processed, instead of re-computing the entire column sum again from scratch, one can use the previous column sum and subtract the first row and add the next row. Although this would not yield any benefits when using a 3x3 window size as the window size increases in height, the cost of computing the column sum remains fixed at 2 additions once the first n rows of each block are computed where n is equal to the window height.

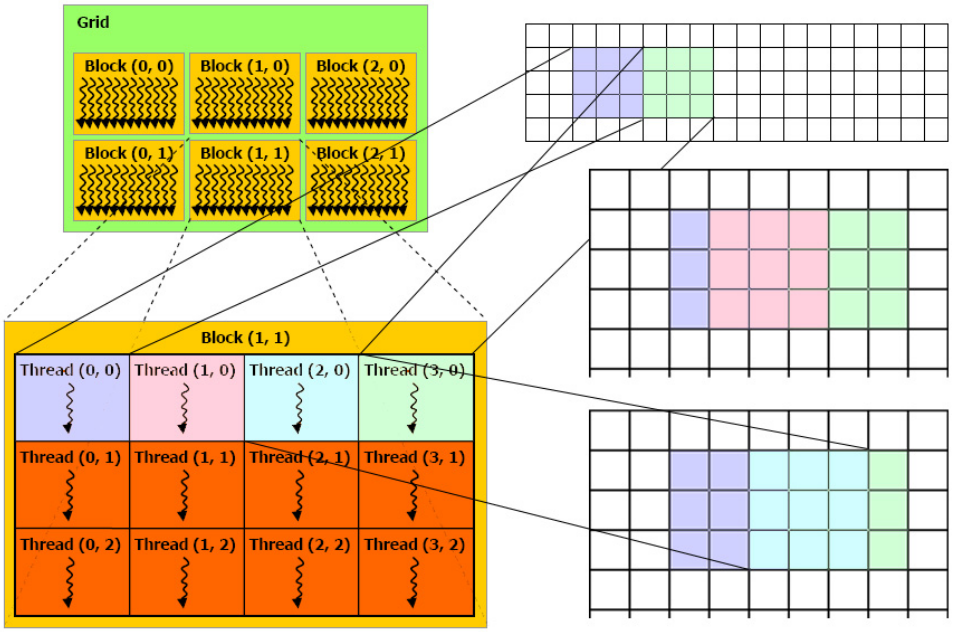

Figure 6.6 Naive SSD Kernel on 3x3 Window

This method contains some added overheads that are more than offset by the computational savings.

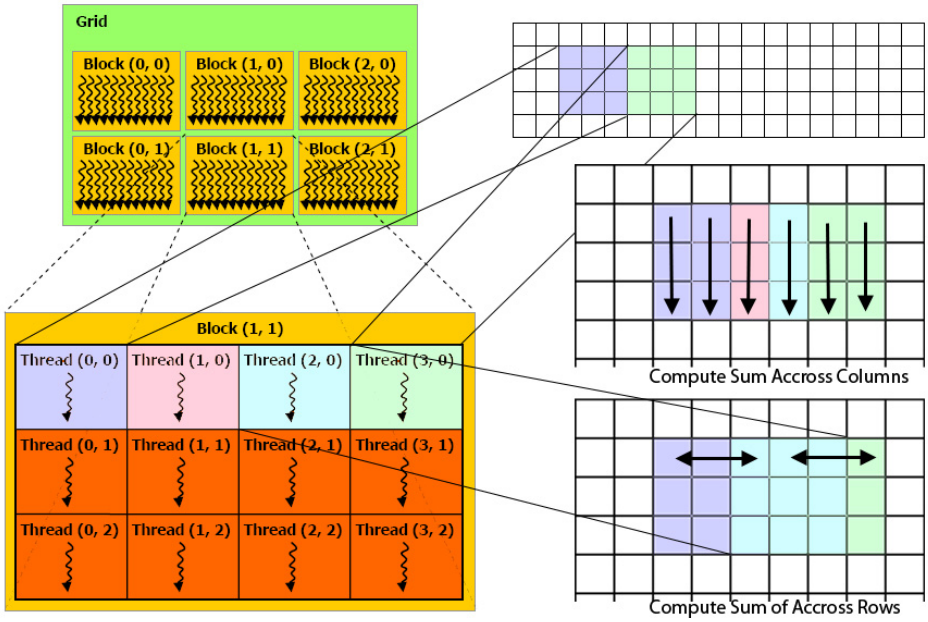

Figure 6.7 Optimized SSD kernel computation

Table 6.5 shows the computational time taken for both versions of the SSD kernel for various window sizes. The optimized SSD kernel not only runs faster but as one can see from Figure 6.8 that the computational time scales better (i.e. the computational cost of the optimised SSD kernel is almost constant) relative to window size.

	Width pixels	Height pixels	WinX pixels	WinY pixels	WinZ pixels	MaxD pixels	Win Size pixels	SSD Time in seconds
Naïve SSD	320	240	3	3	1	80	9	0.008091
	320	240	5	5	1	80	25	0.010821
	320	240	7	7	1	80	49	0.014375
	320	240	9	9	1	80	81	0.019907
	320	240	11	11	1	80	121	0.026195
	320	240	13	13	1	80	169	0.034381
	320	240	15	15	1	80	225	0.04321
Optimized SSD								
	320	240	3	3	1	80	9	0.007783
	320	240	5	5	1	80	25	0.007903
	320	240	7	7	1	80	49	0.007931
	320	240	9	9	1	80	81	0.008062
	320	240	11	11	1	80	121	0.007995
	320	240	13	13	1	80	169	0.008099
	320	240	15	15	1	80	225	0.008215

Table 6.5 Timings naive SSD versus optimized

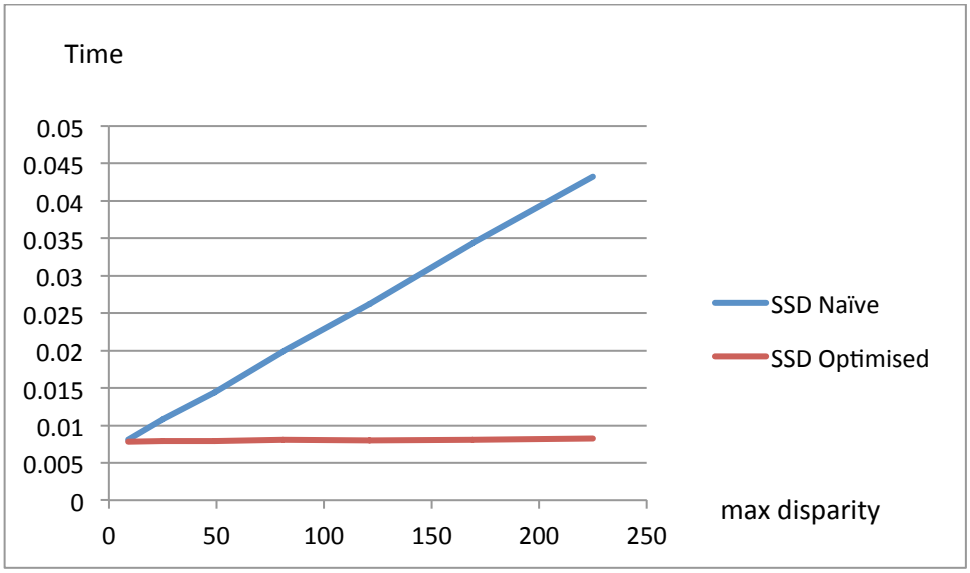

Figure 6.8 Scaling between naive SSD and optimised SSD kernel the x-axis represents the number of pixel in the space domain while the y-axis represents the computational time

6.2.4 Load Balancing and Further CPU gains

Having optimized the GPU parts of the algorithm, the CPU part (i.e. the multi-layer DP optimization) was then examined and optimized. Having re-implemented the SSD kernel as described in the previous section of this chapter, and by caching the squared dissimilarity matrix, the window size parameters almost have a fixed computational cost with regard to the hybrid dynamic programming initialization.

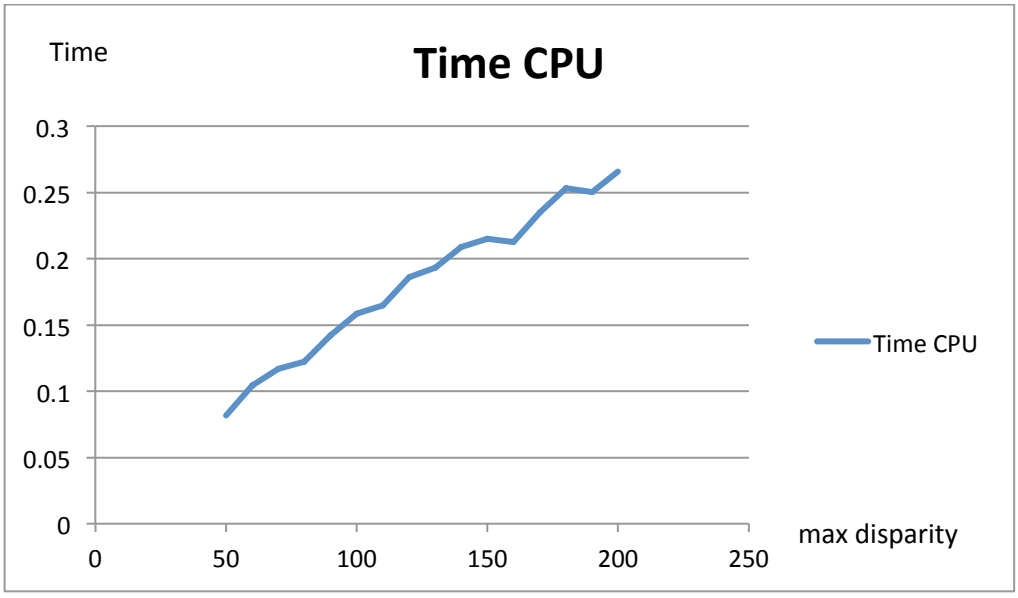

Figure 6.9 CPU computational time scalability with regard to maximum disparity

The only parameters left that could impact the computational cost of this part of the algorithm are the image resolution and the maximum disparity value. The maximum disparity value is determined by the position of cameras and capture volume as well as the image resolution, by down sampling the captured images one can effectively reduce the maximum disparity value necessary to capture the depths of the imaged object (i.e. reducing the image resolution from 640x480 to 320x240 implies reducing the maximum disparity value by half from 140 pixels to 70 pixels). It is clear from Table 6.5 and Figure 6.8 that although the DP Optimization computational time scales almost linearly with regard to the maximum disparity value, this parameter has the greatest impact on the overall performance. Reducing the maximum disparity value clearly improves the system computational performance substantially. It also reduces the maximum image resolution and depth range. One way to maintain some of the performance advantages of a reduced maximum disparity value yet keep this value high, is to modify the DP optimization algorithm to use per-pixel specified maximum disparity values. This however, requires determining what these per-pixel maximum disparity values should be in order to maintain results with the same quality. One potential cheap solution would be to use the results of the previous frame plus some added threshold determined by the maximum potential gradient.

Width pixels	Height pixels	WinX pixels	WinY pixels	WinZ pixels	MaxD pixels	Time CPU in seconds
320	240	5	7	1	50	0.081578
320	240	5	7	1	60	0.104211
320	240	5	7	1	70	0.116923
320	240	5	7	1	80	0.12216
320	240	5	7	1	90	0.141939
320	240	5	7	1	100	0.158527
320	240	5	7	1	110	0.164631
320	240	5	7	1	120	0.185802
320	240	5	7	1	130	0.193432
320	240	5	7	1	140	0.208493
320	240	5	7	1	150	0.214784
320	240	5	7	1	160	0.212478
320	240	5	7	1	170	0.234778
320	240	5	7	1	180	0.252913
320	240	5	7	1	190	0.25043
320	240	5	7	1	200	0.265732

Table 6.6 Timing of CPU one core multi-layer DP optimization

Width	Height	Win X	Win Y	Win Z	Max D	SubSampled Total	Per-Pixel MaxD	Total Time	Original Time	% Saving
320	240	5	7	1	60	0.023641	0.058532	0.08217	0.104211	21.14747
320	240	5	7	1	80	0.028418	0.065111	0.09352	0.12216	23.43729
320	240	5	7	1	100	0.033092	0.071979	0.10507	0.158527	33.72043
320	240	5	7	1	120	0.039647	0.08065	0.12029	0.185802	35.25527
320	240	5	7	1	140	0.042396	0.086537	0.12893	0.208493	38.15955
320	240	5	7	1	160	0.046196	0.093227	0.13942	0.212478	34.38238
320	240	5	7	1	180	0.050422	0.10415	0.15457	0.252913	38.88333
320	240	5	7	1	200	0.05584	0.101596	0.15743	0.265732	40.75384

Table 6.7 Timings of multi-scale DP optimizations versus single scale and computational savings

This however, would produce errors at depth discontinuities. One could potentially use the detected discontinuities from the previous frame's reconstruction along with neighbouring pixels and reset their maximum disparity values to the highest maximum disparity. The neighbouring pixels would account for motion. Although this could potentially eliminate the errors, a much simpler solution is to use a multi-scale approach. This approach is only worth using if the computational savings achieved by using a per-pixel maximum disparity values offset the added cost of performing a sub-sampled reconstruction. Table 6.7 displays the timings for the reconstruction with per-pixel maximum disparity values as well as the cost of the sub-sampled reconstruction used to compute them, with the final column indicating the computational savings achieved. Table 6.7 clearly demonstrates the advantages in terms of speed, of using a multi-scale approach where the sub-samples are used to compute a per-pixel maximum disparity that is then used to speed up the higher resolution reconstruction.

All these optimizations have significantly improved the computational performance of the DP algorithm, and as the following section will demonstrate, have enabled the creation of a system that can run at over 30 fps. Nevertheless what this shows is how sensitive all GPU accelerated algorithms are to implementation. It was not apparent at first that adding an extra for loop inside the CUDA shader and thereby reducing the number of kernel calls would yield such speed increases. In order to get the most out of GPUs one needs a thorough understanding of their architectures in order to leverage the most out of them and make unintuitive design decisions.

6.3 Scalability Framework

Having developed a real-time sub-pixel stereo reconstruction algorithm that consists of mixing and matching various different components with differing parameters, I present a unified scalable framework. The goal of this framework is to create a flexible mechanism for selecting different stereo algorithms dependent on varying constraints such as quality, computational speed and computation resources. This framework allows the tailoring of the reconstruction algorithm given various input image resolutions as well as speed constraints under limited computational resources. Figure 6.10 illustrates this framework. One can clearly see the division between the initialization and non-linear optimization parts of the algorithm. Decoupling the dissimilarity cost computation performed on the GPU from the dynamic programming scan-line optimization makes it possible to replace this part of the algorithm with alternatives such as a naive winner takes all algorithm which makes the framework more generalizable when future algorithms are proposed.

Each component within this framework has a set of parameters that determine the quality of the output as well as the computational performance (i.e. speed) of the system. Table 6.8 is an overview of all the system parameters and their relationship between speed versus quality. This table can also be visualized as a graph in Figure 6.9

The goal of the system I have presented is to achieve the best results within the constraint of real-time applications. Chapter 3 demonstrated that when using structured light superior results were achieved, using the normalized SSD cost function extended into the time domain in combination with the three layer dynamic programming scan-line optimization. In this application it is therefore more beneficial to use the SSD cost function for both superior speed as well as results.

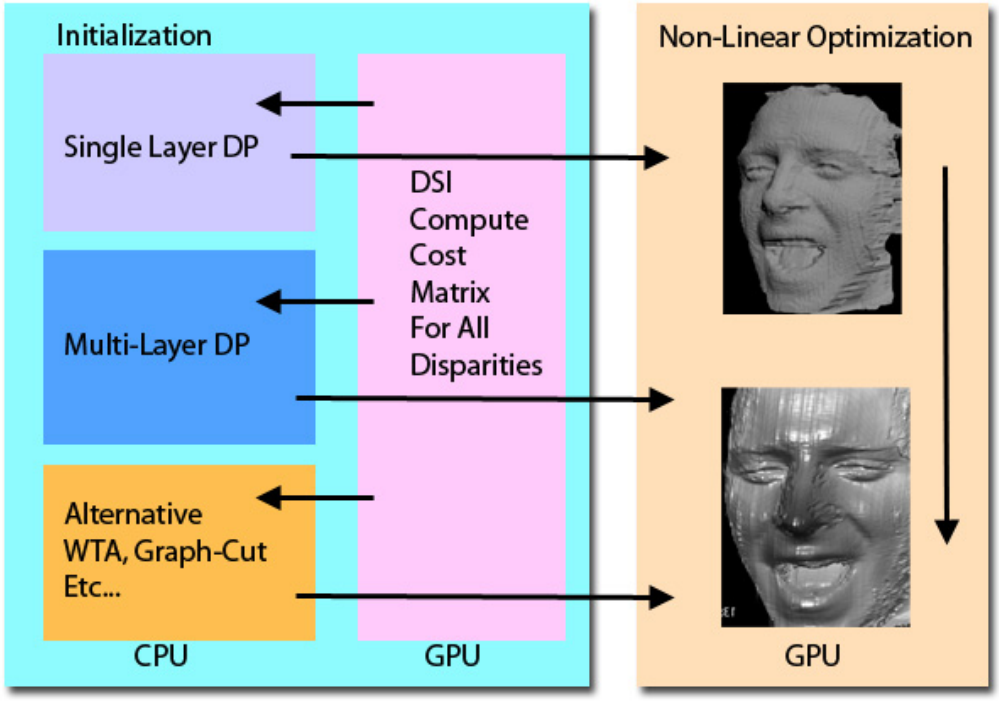

Figure 6.10 High-Level Scalability Framework

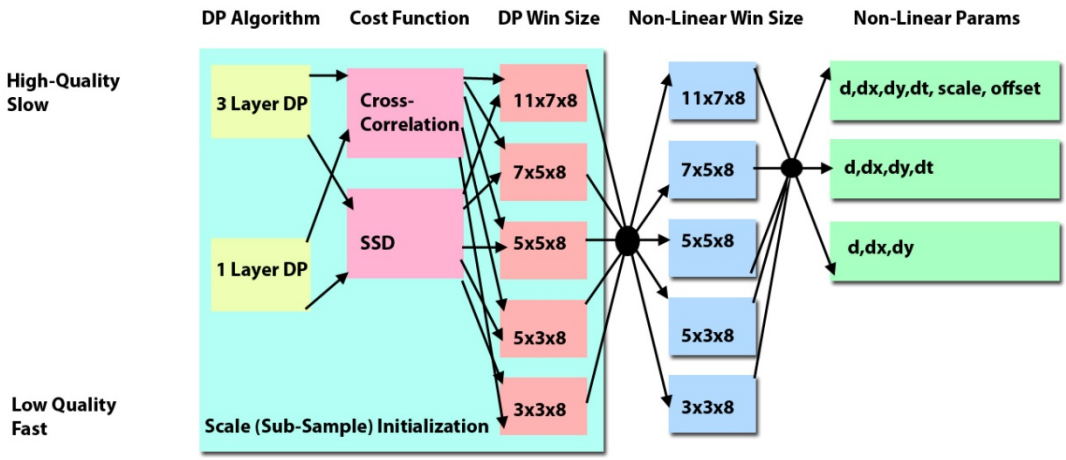

Figure 6.11 Scalable framework with System Parameters

	DP Parameters	**Optimization Parameters**	
High Quality	Full Resolution MaxD	5+ Iterations	**Low Performance**
3/4 Layer DP	Normalized Cross Correlation Cost Function	Disparity Gradient Constraint	
	Window 6x6x8	Optimize Colourmetric Params between Cameras	
	Window 3x6x8	Optimize d, dx, dy, dt	
	Window 3x3x4	Optimize d, dx, dy	
	SSD Cost Function		
	Window 6x6x8	Window 6x6x8	
	Window 3x6x8	Window 3x6x8	
	Window 3x3x4	Window 3x3x4	
Traditional DP			
Low Quality	Sub-Sampled MaxD	1-2 Iterations	**High Performance**

Table 6.8 System parameters quality versus speed

However, the normalized cross-correlation cost function is still included in the framework as in other circumstances, such as without the use of structured light it can provide superior results at the cost of speed. It is also worth noting that the size of the support region (i.e. window size) does not have to be the same for the initialization part of the algorithm as for the non-linear optimization part. Although real-time performance is achievable using most of the high quality settings for the parameters with the current cameras and resolution, the given framework is important as it enables a scalable system should higher resolution and more cameras be used. It also enables the system to scale not only with image resolution but also with CPU and GPU computational improvements. Table 6.9 demonstrates some of the performance achievable using differing parameters represented in Figure 6.11.

Width	Height	Win X	Win Y	Win Z	Max D	Iterations	DP 1/4 Scale	DP 1/2 Scale	GPU Non-Linear	Total-Time	Total Fps
640	480	3	3	8	80	3	0.010911	0.046368	0.014627	0.07190	13.907
640	480	5	3	8	80	3	0.010894	0.044446	0.01442	0.06976	14.334
640	480	5	5	8	80	3	0.011217	0.045079	0.015958	0.07225	13.840
640	480	7	5	8	80	3	0.011176	0.045518	0.016868	0.07356	13.593
640	480	11	7	8	80	3	0.011501	0.044016	0.023168	0.07868	12.708

Table 6.9 Timings for total reconstruction 4 cores with DP up to half-scale

Width	Height	Win X	Win Y	Win Z	Max D	Iterations	DP 1/4 Scale	GPU Non-Linear	Total-Time	Total Fps
640	480	3	3	8	80	3	0.010911	0.014627	0.025538	39.1573
640	480	5	3	8	80	3	0.010894	0.01442	0.025314	39.5038
640	480	5	5	8	80	3	0.011217	0.015958	0.027175	36.7985
640	480	7	5	8	80	3	0.011176	0.016868	0.028044	35.6582
640	480	11	7	8	80	3	0.011501	0.023168	0.034669	28.8442

Table 6.10 Timings for total reconstruction 4 cores with DP up to quarter-scale

Table 6.9 is a summary of the total time taken to perform reconstructions with various window sizes on 4 cores and using half the resolution to initialize the non-linear optimization step. Table 6.10 is very similar, with the exception, that the non-linear optimization step is initialized using a quarter of the resolution. For the non-linear optimization step only the *d, dx*, and *dy* where optimized to achieve a higher performance. It is also worth noting that for both these timings the load balancing on the CPU was not perfect. This is because differing scan-lines take varying amounts of time for the multi-layer DP to solve. This could potentially be improved by using dynamic load balancing where each CPU core would be allocated a differing number of scan-lines based on their previous computational time. In both these tables the window sizes for the DP part of the algorithm were the same as for the non-linear optimization part. However, this is not a forced constraint of the system and does not have to be the case. Other observations that can be made from both these tables are that the window size has a more significant impact on the non-linear optimization part of the algorithm than the DP solver. This is because of the previously described SSD optimization almost removes the impact of the window size with regard to the DP solver. This demonstrates that the different parameters all affect the computational time of the system to varying degrees, and it is therefore important, as one will see in the next section of the chapter, to determine the scalability of these different parameters on the overall system. These tables also demonstrate that the system can be made to run in real-time.

6.4 Overview of System Parameters

Having optimized both the hybrid CPU-GPU initialization and GPU non-linear optimization parts of the stereo algorithm, it was necessary to determine the scalability of all the system parameters with regard to overall system performance. This enables one to determine the overall system scalability as well as potential trade-offs between

quality and computational speed. The overall system can be broken down into two parts, the initialization which finds pixel level disparity values and the non-linear optimization that gives us the high frequency sub-pixel detail. These two parts of the algorithm both have a set of parameters that affect the quality of the output as well as the computational speed of the algorithm. In previous chapters, the effects of certain parameters on very specific parts of the algorithm were examined. This chapter will examine the effect of every parameter on the overall system performance. With this information one can determine which parameters affect the performance the least and then optimize those in order to achieve the highest quality.

The following lists all the framework parameters:

- Initialization
    - Algorithm DP, multi-Layer DP, WTA, etc...
    - Maximum initialization resolution
    - Maximum Disparity
    - Window Size X,Y,Z
    - Cost Function
- Non-linear Parameters
    - Final Reconstruction Resolution
    - Window Size X, Y, Z
    - Number of optimization parameters per disparity (d, dx, dy, dt)
    - Radiometric Calibration (scale, offset)
    - Number of non-linear optimization iterations

With the constraint of real-time performance and quality trade-offs discussed in previous chapters I have limited the scope of some of these parameters. This motivated the optimization presented previously in the hybrid CPU-GPU implementation and therefore limited the scope of parameters examined. All the timings presented in this section will be taken from a single core implementation to remove all CPU load balancing issues that could potentially skew the scalability results. The fact that the squared difference images as well as SSD are cached, increasing the window size past two frames into the temporal domain, does not affect the computational performance of

the hybrid CPU-GPU DP implementation. The non-linear optimization algorithm was optimized to use eight temporal frames by packing them into two RGBA textures. Again reducing the temporal window size will also have almost no impact on performance, therefore all the following experiments will be using eight temporal frames.

6.5 Experiments

Using the previously mentioned objectives the following experiments were carried out. The non-linear optimization step was timed with the following parameters.

Non-Linear Optimization GPU

Image Size	**Window**	**Size**				**Number of Iterations**
640x480	3x3	5x3	5x5	7x5	11x7	2-5
640x240	3x3	5x3	5x5	7x5	11x7	2-5
640x240	3x3	5x3	5x5	7x5	11x7	2-5
640x120	3x3	5x3	5x5	7x5	11x7	2-5
320x240	3x3	5x3	5x5	7x5	11x7	2-5
320x120	3x3	5x3	5x5	7x5	11x7	2-5
320x120	3x3	5x3	5x5	7x5	11x7	2-5
320x60	3x3	5x3	5x5	7x5	11x7	2-5
160x120	3x3	5x3	5x5	7x5	11x7	2-5
160x60	3x3	5x3	5x5	7x5	11x7	2-5
160x60	3x3	5x3	5x5	7x5	11x7	2-5
160x30	3x3	5x3	5x5	7x5	11x7	2-5

Certain image sizes are listed twice this is because the input images were divided into two halves of equal resolution and the timings were performed on both halves separately. This was done in order to compare each half with the other and gain and insight into the computational burden of different parts of the input dataset. Subsequently the initialization hybrid CPU-GPU step was timed with the following parameters.

Hybrid CPU- GPU Initialization

Image	Window	Size					Maximum	Disparity		
640x480	3x3	5x3	5x5	7x5	11x7	60	80	100	120	160
640x240	3x3	5x3	5x5	7x5	11x7	60	80	100	120	160
640x240	3x3	5x3	5x5	7x5	11x7	60	80	100	120	160
640x120	3x3	5x3	5x5	7x5	11x7	60	80	100	120	160
320x240	3x3	5x3	5x5	7x5	11x7	60	80	100	120	160
320x120	3x3	5x3	5x5	7x5	11x7	60	80	100	120	160
320x120	3x3	5x3	5x5	7x5	11x7	60	80	100	120	160
320x60	3x3	5x3	5x5	7x5	11x7	60	80	100	120	160
160x120	3x3	5x3	5x5	7x5	11x7	60	80	100	120	155
160x60	3x3	5x3	5x5	7x5	11x7	60	80	100	120	155
160x60	3x3	5x3	5x5	7x5	11x7	60	80	100	120	155
160x30	3x3	5x3	5x5	7x5	11x7	60	80	100	120	155

6.6 Results

This section will examine all the timing results taken from the experiments carried out. Firstly, the timing results from the initialization multi-layer DP using various combination and of all the parameters for differing resolutions. Secondly the results from the non-linear optimizations step, again with all the different combinations of parameters. All this data will be analysed and insight into the overall systems scalability will be determined and discussed. For each set of experiments an example table is shown in this section. For the full set of results readers are referred to Appendix A.

All the timings taken in Tables 6.10 through Table 6.13 are taken using the hybrid CPU-GPU multi-layer DP solver but without using the per-pixel maximum disparity optimization previously discussed in Section 6.2. The following tables show the computational times using the per-pixel maximum disparity optimization. These maximum disparity values were computed using the ¼ scale reconstruction and therefore the Tables A.14 to A.21 timings includes the time taken for the ¼ scale reconstruction. The Tables A.22 through A.33 will represent all the timings from the non-linear optimization step.

Width	Height	WinX	WinY	WinZ	Max Disparity	Total Time
160	120	3	3	8	60	0.033347
160	120	3	3	8	80	0.037874
160	120	3	3	8	100	0.045399
160	120	3	3	8	120	0.054052
160	120	3	3	8	160	0.058045
160	120	5	3	8	60	0.03301
160	120	5	3	8	80	0.037918
160	120	5	3	8	100	0.045161
160	120	5	3	8	120	0.05329
160	120	5	3	8	160	0.056952
160	120	5	5	8	60	0.032838
160	120	5	5	8	80	0.038327
160	120	5	5	8	100	0.045239
160	120	5	5	8	120	0.053065
160	120	5	5	8	160	0.057578
160	120	7	5	8	60	0.032537
160	120	7	5	8	80	0.037708
160	120	7	5	8	100	0.044391
160	120	7	5	8	120	0.052973
160	120	7	5	8	160	0.056389
160	120	11	7	8	60	0.032105
160	120	11	7	8	80	0.037076
160	120	11	7	8	100	0.043921
160	120	11	7	8	120	0.052127
160	120	11	7	8	160	0.055659

Table 6.11 1 DP Initialization on sub-sampled images using single maximum disparity value

Width	Height	WinX	WinY	WinZ	Max D	Total	Total With 1/4 Scale
640	480	3	3	8	60	0.301388	0.354361
640	480	3	3	8	80	0.383018	0.435991
640	480	3	3	8	100	0.467609	0.520582
640	480	3	3	8	120	0.589471	0.642444
640	480	3	3	8	160	0.719986	0.772959
640	480	5	3	8	60	0.297865	0.350838
640	480	5	3	8	80	0.384081	0.437054
640	480	5	3	8	100	0.437641	0.490614
640	480	5	3	8	120	0.584831	0.637804
640	480	5	3	8	160	0.712621	0.765594
640	480	5	5	8	60	0.29948	0.352453
640	480	5	5	8	80	0.366984	0.419957
640	480	5	5	8	100	0.445689	0.498662
640	480	5	5	8	120	0.583963	0.636936
640	480	5	5	8	160	0.693221	0.746194
640	480	7	5	8	60	0.303334	0.356307
640	480	7	5	8	80	0.368601	0.421574
640	480	7	5	8	100	0.458332	0.511305
640	480	7	5	8	120	0.57774	0.630713
640	480	7	5	8	160	0.689129	0.742102
640	480	11	7	8	60	0.307562	0.360535
640	480	11	7	8	80	0.36834	0.421313
640	480	11	7	8	100	0.471021	0.523994
640	480	11	7	8	120	0.578921	0.631894
640	480	11	7	8	160	0.680127	0.7331

Table 6.12 DP Initialization on images using per-pixel maximum disparity values

Iterations	Width	Height	WinX	WinY	WinZ	Total
2	640	480	3	3	8	0.020767
3	640	480	3	3	8	0.025827
4	640	480	3	3	8	0.033568
5	640	480	3	3	8	0.039342
2	640	480	5	3	8	0.027037
3	640	480	5	3	8	0.037799
4	640	480	5	3	8	0.04699
5	640	480	5	3	8	0.05731
2	640	480	5	5	8	0.040821
3	640	480	5	5	8	0.056569
4	640	480	5	5	8	0.072151
5	640	480	5	5	8	0.087587
2	640	480	7	5	8	0.053771
3	640	480	7	5	8	0.074791
4	640	480	7	5	8	0.096117
5	640	480	7	5	8	0.11701
2	640	480	11	7	8	0.105941
3	640	480	11	7	8	0.152719
4	640	480	11	7	8	0.195734
5	640	480	11	7	8	0.242755

Table 6.13 Non-Linear Optimization GPU Full Resolution

Iterations	Width	Height	WinX	WinY	WinZ	Total
2	640	240	3	3	8	0.01221
3	640	240	3	3	8	0.01484
4	640	240	3	3	8	0.019101
5	640	240	3	3	8	0.021021
2	640	240	5	3	8	0.015598
3	640	240	5	3	8	0.021115
4	640	240	5	3	8	0.026113
2	640	240	5	5	8	0.022455
3	640	240	5	5	8	0.03038
4	640	240	5	5	8	0.038215
5	640	240	5	5	8	0.046375
2	640	240	7	5	8	0.028577
3	640	240	7	5	8	0.040728
4	640	240	7	5	8	0.050396
5	640	240	7	5	8	0.063763
2	640	240	11	7	8	0.056885
3	640	240	11	7	8	0.081567
4	640	240	11	7	8	0.10474
5	640	240	11	7	8	0.130141

Table 6.14 Non-Linear Optimization GPU Full Resolution lower half

The data contained in all these tables gives insight into all the scalability properties of each parameter of the system.

6.6.1 Scalability with regard to Window Size

Earlier in this chapter when discussing the SSD kernel optimizations in the hybrid CPU-GPU implementation, the window parameters in the space domain were shown to scale at almost a constant rate. This proves to be true for all differing configurations of the initialization step. Regardless of the resolution or maximum disparity values, increasing the window size has a very small impact on the computational time of the initialization step. This parameter is also the least expensive. The window size with regard to the non-linear optimization step has a different scalability, its impact on performance is more pronounced.

The following figures represent the computational time spent in the non-linear optimization step relative to the number of pixels contained in the support window. The

x-axis represents the number of pixels and the y-axis represents time. The different graphs represent the various resolutions. Within each graph each plot represents the total non-linear optimization time for varying numbers of iterations.

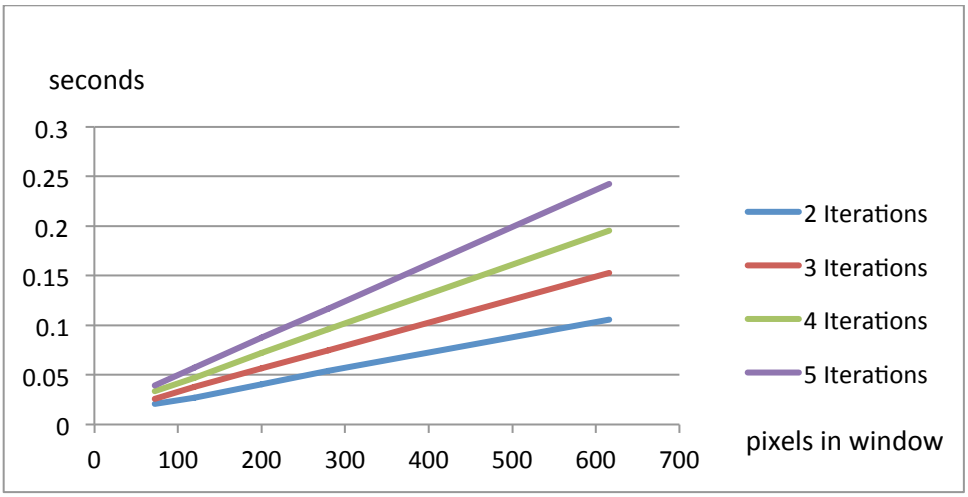

Figure 6.12 Window scalability at 640x480

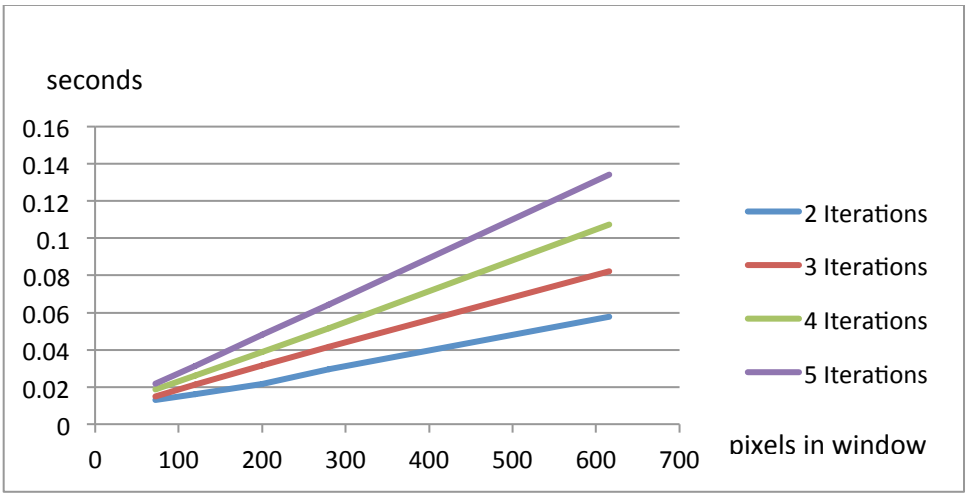

Figure 6.13 Window scalability at 640x240 Lower Half

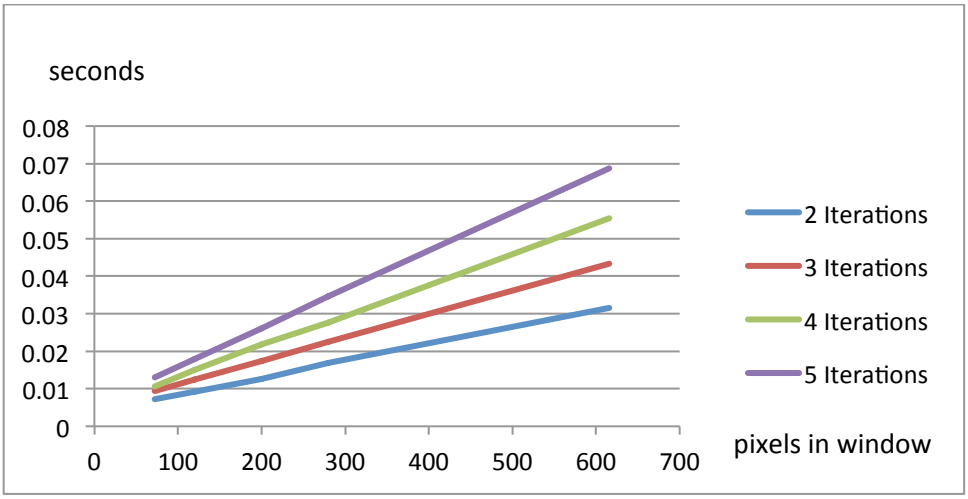

Figure 6.14 Window scalability at 640x120 band across image

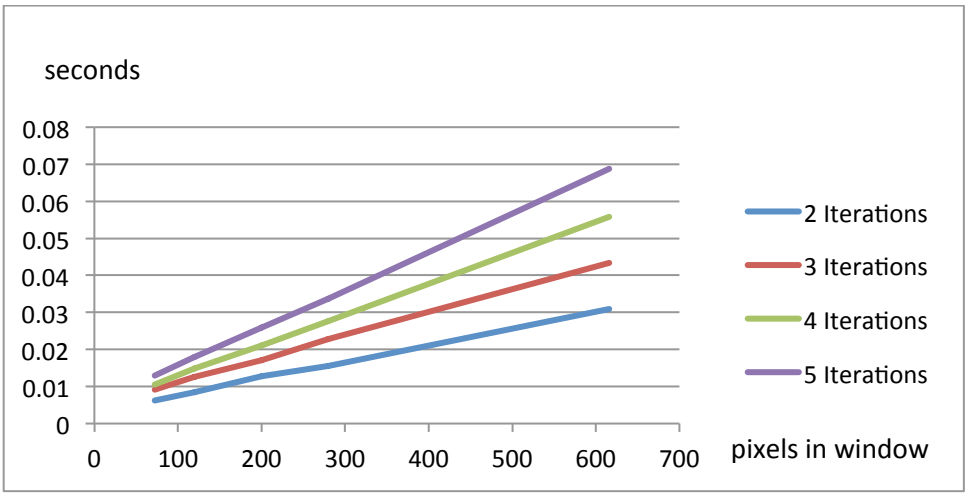

Figure 6.15 Window scalability at 320x240

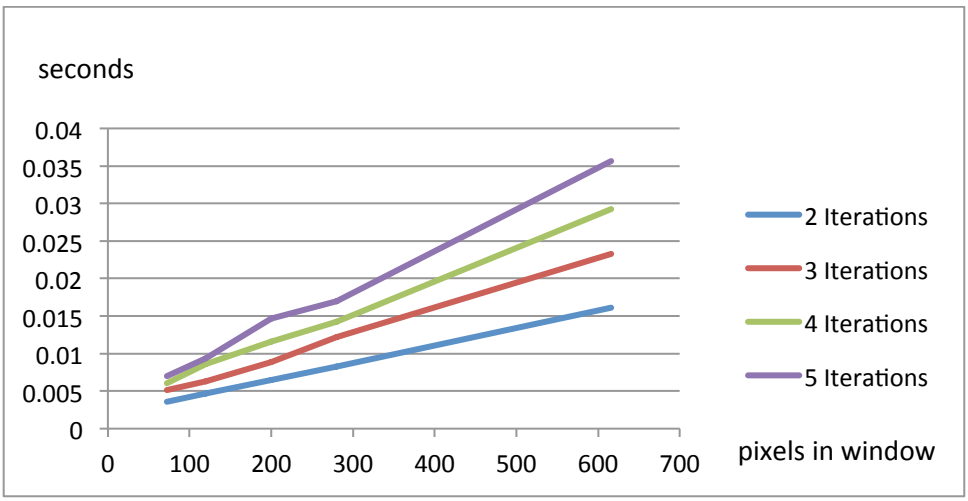

Figure 6.16 Window scalability at 160x120

These figures demonstrate certain properties. Firstly, the computational time of the non-linear optimization step scales linearly with regard to the number of pixels contained in the support window. This holds true for all resolutions and differing numbers of optimization iterations. Secondly, there is a correlation between the number of iterations and window size scalability. With increasing iterations the window scalability, although still linear, worsens, and this correlation is also linear.

6.6.2 Scalability with regard to Resolution

This section examines the scalability with regard to image resolution for both the initialization step and the non-linear optimization. Figure 6.17 illustrates the time taken for the initialization step for varying maximum disparity values. Since the previous section demonstrated that the window size has little effect on the performance of the initialization step, all the timings in Figure 6.17 were taken using 11x7x8 window sizes. This figure demonstrates the following properties:

- Initialization does not scale linearly with regard to resolution
- The scaling differs between the number of image pixels and the image width indicated by the dip in curves at 640x120 (for certain maximum disparity values the computational time is less for 640x120 images than it is for 320x240 images although they both contain the same number of pixels)
- As the image resolution increases so does the impact of using greater maximum disparity values

The non-linear optimization step scales in a similar fashion to the initialization step with regard to image resolution. There are however some subtle differences. The GPU non-linear optimization step has a maximum performance whereby reducing the input resolution no longer reduces the computational time (i.e. the computational time for 320x60 is similar or identical to the computational time for 160x30 images). The scaling characteristics (as illustrated by Figure 6.18 through Figure 6.23) are also very similar when using differing window sizes and varying numbers of iterations. Again with a slight amplification, with an increase in window size and number of iterations. Although

the non-linear optimization step scales in a similar fashion to the DP initialization step, once the input resolution exceeds a certain threshold, it is worth noting, that the computational speed of the non-linear solver is almost an order of magnitude faster than the DP initialization.

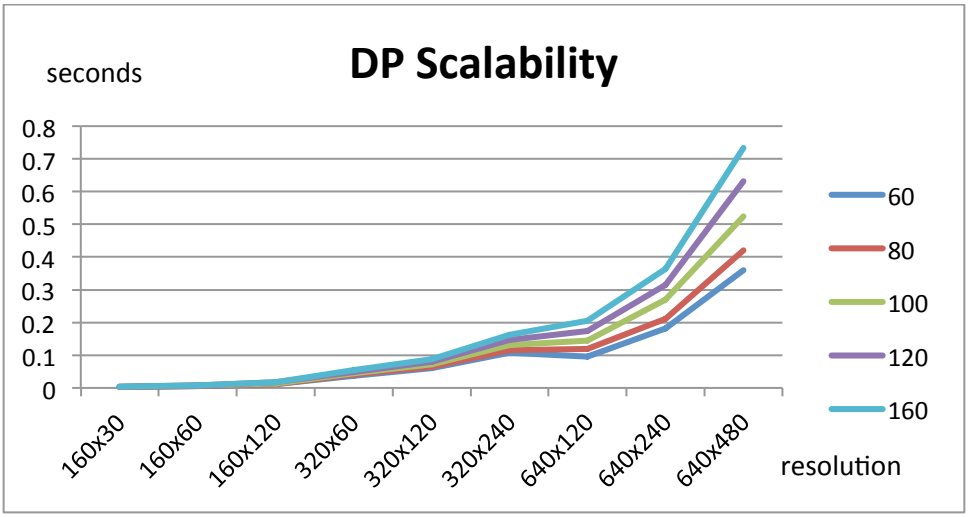

Figure 6.17 DP initialization time at different resolutions for various maximum disparity values

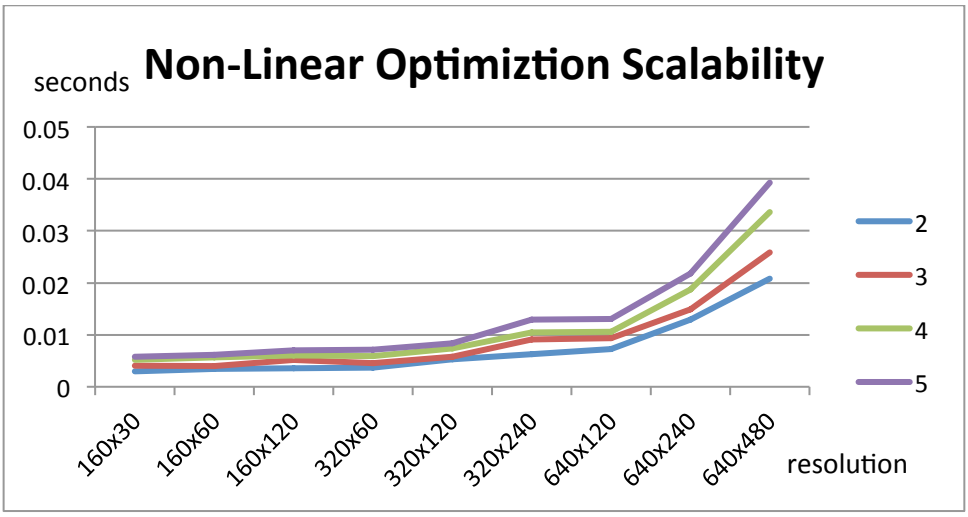

Figure 6.18 Non-linear optimization at different resolutions using 3x3x8 window

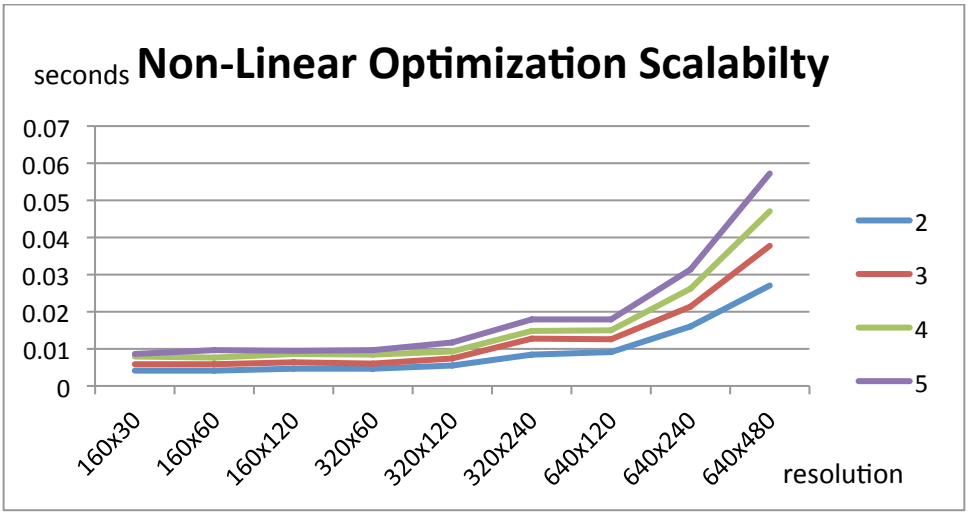

Figure 6.19 Non-linear optimization at different resolutions using 5x3x8 window

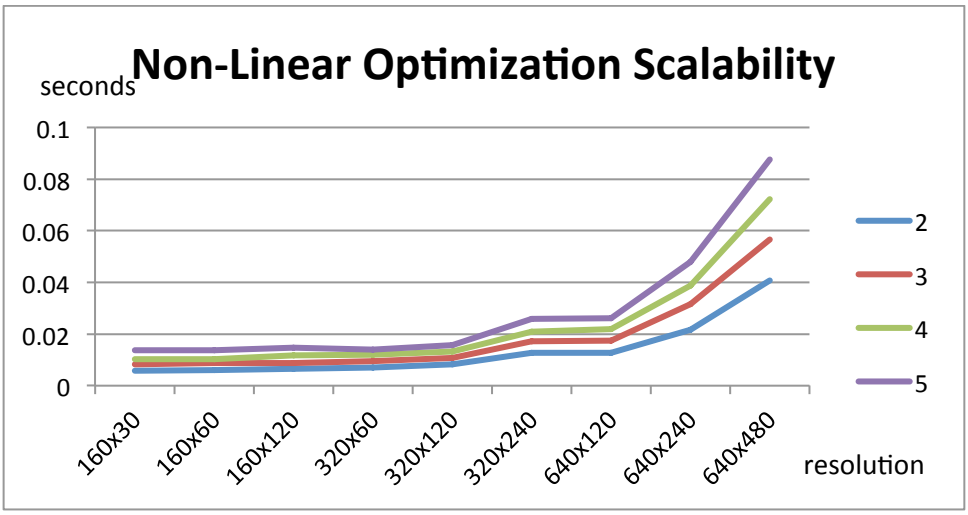

Figure 6.20 Non-linear optimization at different resolutions using 5x5x8 window

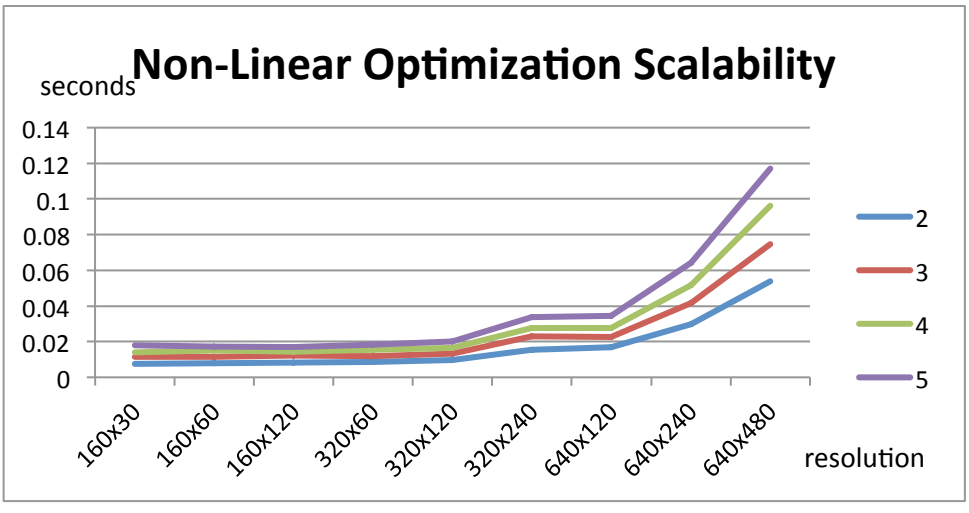

Figure 6.21 Non-linear optimization at different resolutions using 7x5x8 window

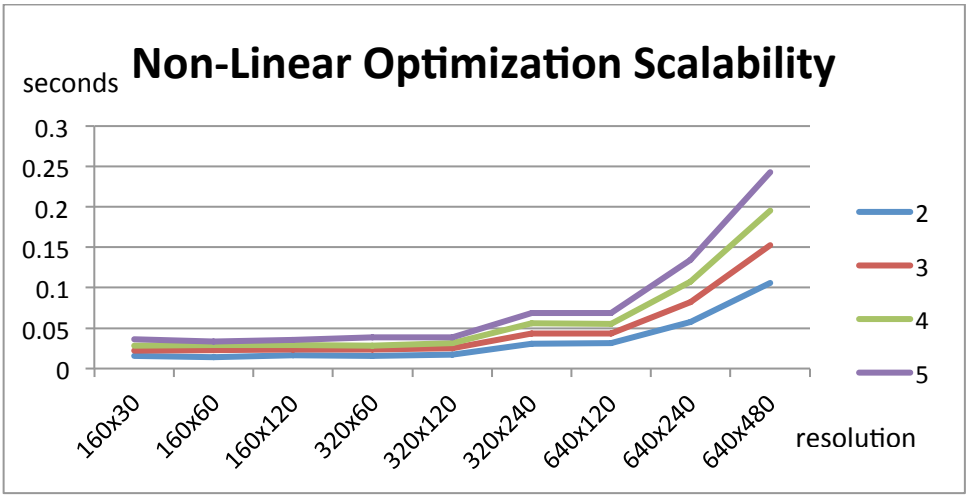

Figure 6.22 Non-linear optimization at different resolutions using 11x7x8 window

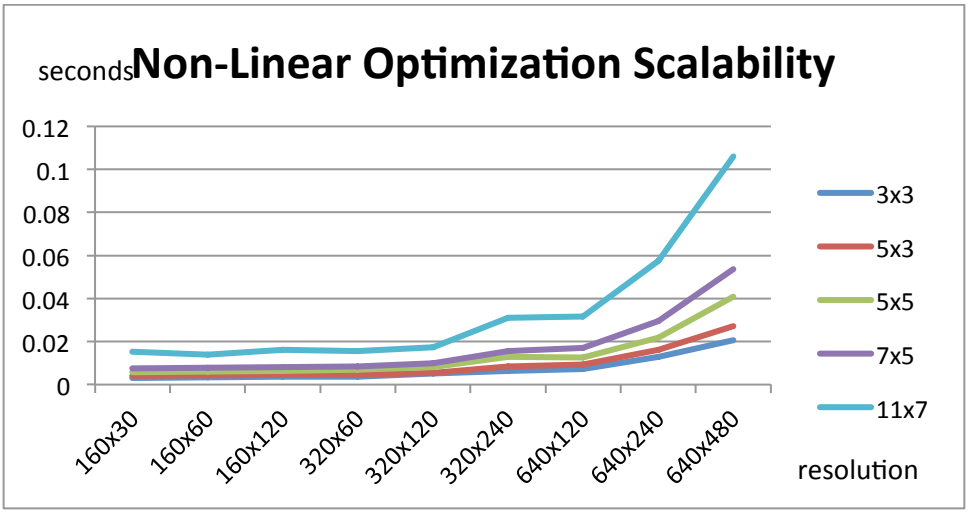

Figure 6.23 Non-linear optimization at different resolutions using 2 iterations

6.6.3 Scalability with regard to maximum disparity

The following Figure 6.24 illustrates the computational time of the initialization step for a range of maximum disparity values on three image sets of different resolutions. This figure shows there is a correlation between the scalability of the maximum disparity values and the image resolution. The resolutions of the images affect the scaling relative to the maximum disparity parameter, as the image resolution increases so does the gradient of this parameter. Although the scaling worsens it does still however remain close to linear.

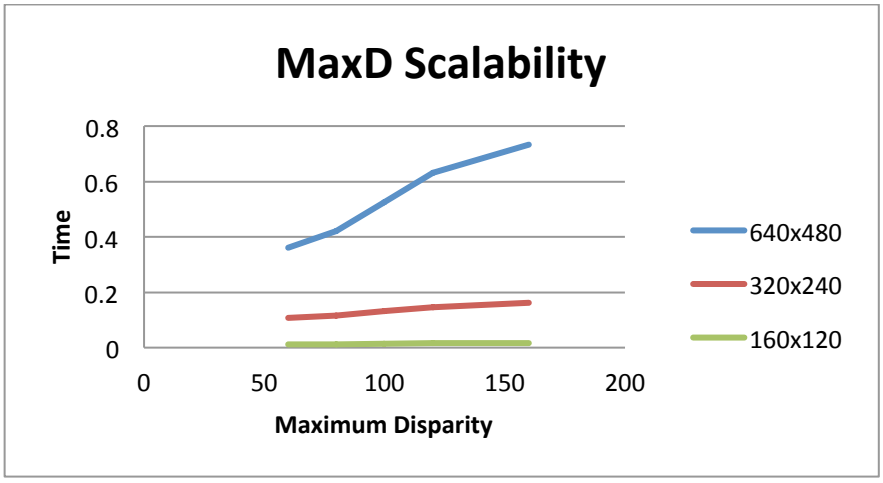

Figure 6.24 Time for DP at varying resolutions for different maximum disparity values

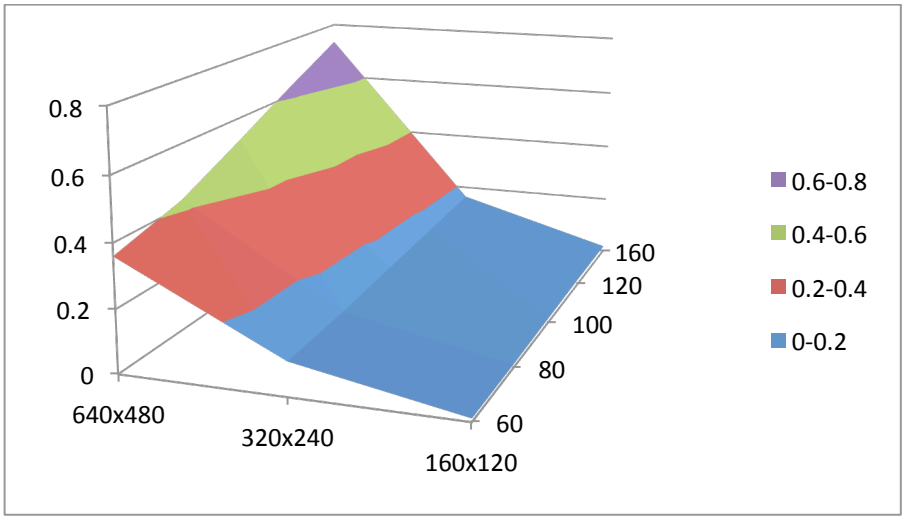

Figure 6.25 3D plot of computational against maximum disparity and image resolution

6.6.4 Scalability with regard to non-linear optimization iterations

The final parameter of the stereo reconstruction system presented in this chapter is the number of non-linear optimization iterations. Figure 6.26 illustrates the scaling of this parameter. It is clear that it scales in a linear fashion and its gradient is relative to window size.

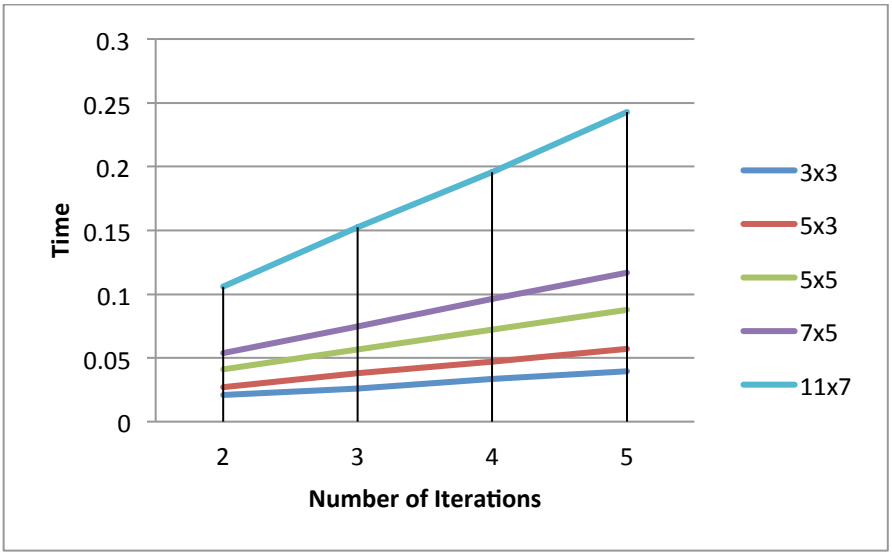

Figure 6.26 Time for non-linear optimization using increasing number of iterations for differing window sizes

Iterations	Width	Height	Win X	Win Y	Win Z	Total	Lower Half	Upper Half	Upper+Lower	OverHead
2	640	480	3	3	8	0.0207	0.01221	0.012888	0.025098	0.004331
3	640	480	3	3	8	0.0258	0.01484	0.014895	0.029735	0.003908
4	640	480	3	3	8	0.0335	0.01910	0.01867	0.037771	0.004203
5	640	480	3	3	8	0.0393	0.02102	0.021763	0.042784	0.003442
2	640	480	5	3	8	0.0270	0.01559	0.01605	0.031648	0.004611
3	640	480	5	3	8	0.0378	0.02111	0.021394	0.042509	0.00471
4	640	480	5	3	8	0.0469	0.02611	0.026233	0.052346	0.005356
2	640	480	5	5	8	0.0408	0.02245	0.021682	0.044137	0.003316
3	640	480	5	5	8	0.0565	0.03038	0.031489	0.061869	0.0053
4	640	480	5	5	8	0.0721	0.03821	0.03888	0.077095	0.004944
5	640	480	5	5	8	0.0875	0.04637	0.048038	0.094413	0.006826
2	640	480	7	5	8	0.0537	0.02857	0.02969	0.058267	0.004496
3	640	480	7	5	8	0.0747	0.04072	0.041534	0.082262	0.007471
4	640	480	7	5	8	0.0961	0.05039	0.051617	0.102013	0.005896
5	640	480	7	5	8	0.1170	0.06376	0.064171	0.127934	0.010924
2	640	480	11	7	8	0.1059	0.05688	0.057607	0.114492	0.008551
3	640	480	11	7	8	0.1527	0.08156	0.082319	0.163886	0.011167
4	640	480	11	7	8	0.1957	0.10474	0.107273	0.212013	0.016279
5	640	480	11	7	8	0.2427	0.13014	0.134223	0.264364	0.021609

Table 6.15 Timing for non-linear optimization complete image versus segmented image

6.6.5 Computational Load and Balancing

Having examined the impact of every parameter for this stereo reconstruction system on computational performance it would be advantageous to determine the potential gains to be achieved with regard to future hardware developments or increased computational

resources. By determining the computational load across the scan-lines it is possible to extrapolate potential speed increases gained with multiple GPU or more CPU cores.

Width	Height	WinX	WinY	WinZ	Max D	Total	Total Image	Lower	Upper	Overhead
640	480	3	3	8	60	0.30139	0.354361	0.18407	0.17542	0.005121
640	480	3	3	8	80	0.38302	0.435991	0.20068	0.22139	-0.013921
640	480	3	3	8	100	0.46761	0.520582	0.23808	0.26513	-0.017375
640	480	3	3	8	120	0.58947	0.642444	0.30462	0.32041	-0.017418
640	480	3	3	8	160	0.71999	0.772959	0.36904	0.38607	-0.017848
640	480	5	3	8	60	0.29787	0.350838	0.16338	0.17524	-0.012224
640	480	5	3	8	80	0.38408	0.437054	0.20209	0.22241	-0.012551
640	480	5	3	8	100	0.43764	0.490614	0.22532	0.24963	-0.01566
640	480	5	3	8	120	0.58483	0.637804	0.30724	0.31832	-0.012245
640	480	5	3	8	160	0.71262	0.765594	0.36363	0.38199	-0.019977
640	480	5	5	8	60	0.29948	0.352453	0.16261	0.17728	-0.012563
640	480	5	5	8	80	0.36698	0.419957	0.19347	0.21194	-0.014546
640	480	5	5	8	100	0.44569	0.498662	0.22722	0.25491	-0.016533
640	480	5	5	8	120	0.58396	0.636936	0.30223	0.3178	-0.016912
640	480	5	5	8	160	0.69322	0.746194	0.35379	0.37519	-0.017216
640	480	7	5	8	60	0.30333	0.356307	0.16399	0.17788	-0.014431
640	480	7	5	8	80	0.3686	0.421574	0.19296	0.21319	-0.01543
640	480	7	5	8	100	0.45833	0.511305	0.2385	0.25848	-0.014323
640	480	7	5	8	120	0.57774	0.630713	0.30035	0.31671	-0.013654
640	480	7	5	8	160	0.68913	0.742102	0.3727	0.36897	-0.000437
640	480	11	7	8	60	0.30756	0.360535	0.16646	0.18193	-0.012147
640	480	11	7	8	80	0.36834	0.421313	0.19484	0.21201	-0.01446
640	480	11	7	8	100	0.47102	0.523994	0.23798	0.26951	-0.0165
640	480	11	7	8	120	0.57892	0.631894	0.30335	0.31485	-0.013696
640	480	11	7	8	160	0.68013	0.7331	0.35228	0.36456	-0.016262

Table 6.16 Timing for DP initialization complete image versus segmented image

With regard to the non-linear optimization step I have already demonstrated that subdividing the input images yield little benefits at lower resolutions. However, at resolutions of 640x480 and above this is not the case. Table 6.15 shows the non-linear optimization timings for the complete images in the Total column as well as the timing for the lower half and upper half. When adding the time taken for each half together, one notices that this total is greater than the total reconstruction time for entire set of images. This indicates that should two GPUs be used for the non-linear optimization step, one would not get the theoretical maximum speed increase of 2x, no matter how well the system was load balanced. However, this difference shown in the Overhead

column is relatively negligible compared to the total computational time, indicating a potentially substantial performance increase obtained by using two GPUs.

Using the same analysis on the DP initialization as shown by Table 6.16 produces different results. In this particular case treating the image as two halves produces superior results, which indicates that further potential optimizations are possible for DP initialization implementation, possibly due to memory allocation overheads or cache misses. This part of the algorithm would gain performance from using more CPU cores.

6.7 Conclusion

This chapter has demonstrated how to further optimize the stereo system developed for this thesis. It has also presented a scalable framework for a hybrid CPU-GPU sub-pixel stereo reconstruction algorithm and has shown how the reconstruction system fits into this framework. An in depth analysis of the full system scalability has been carried out. From this, I conclude the parameters having the greatest to least impact on performance are listed as follows

- Initialization Resolution
- Maximum Disparity
- Final Reconstruction Resolution
- Non-Linear Optimization Window Size
- Number of non-linear optimization iterations
- Initialization Window Size

From this list one can determine which parameters should be tuned first to achieve the highest computational performance. Almost all parameters scale linearly and I have demonstrated the potential computational increase from using multiple GPU and more CPU cores. Finally, this gives us a clear indication of future potential speed increase with more hardware.

Chapter 7

Conclusion

The goal of the research presented in this thesis was to examine real-time 3D reconstruction in the context of tele-immersion. This body of work focused on stereo techniques but with the added constraints of making no assumptions with regard the underlying shapes being reconstructed and producing a cheap system using off the shelf components. It is clear throughout this dissertation that the quality of 3D reconstructions relies primarily on the quality of the disparity maps produced by the stereo solvers. The scope of the research was therefore narrowed down to examining the stereo correspondence problem, in the context of real-time implementations that run on parallel architectures with an emphasis on scalability.

The main goal of this research was to build a real-time stereo sub-pixel stereo correspondence solver that would scale well on massively parallel architectures such as those found in GPUs. This has been achieved and is clearly demonstrated in the previous chapter. Not only did the final system run in real-time but its performance was clearly demonstrated to scale across multiple processors and new architectures. Although the design of this system is a few years old, the fact that it scales to modern GPU architectures, is a testament to how well suited, the system is for these types of architectures. The lessons learnt and contributions presented in this thesis fall into the following categories:

- The initialization step (dynamic programming)
- Non-linear optimization (Gauss-Newton)
- Parallelization and scalability framework on GPUs

Chapter 3 examined the suitability and performance of various dynamic programming algorithms for solving the correspondence problem using structured light. These algorithms were not designed to be used with structured light and a space-time

support region. It was therefore necessary to determine their suitability in this context. This chapter clearly demonstrated that although using structured light and extending the support region into the time domain improve the correspondence results, the choice of dynamic programming algorithm has a far greater impact on the quality of the results. This chapter also demonstrated that the advantages of using structured light can only be obtained by tailoring the cost function appropriately to the structured light patterns. This was all achieved in context of a parallel implementation suitable for GPUs.

Chapter 4 demonstrated how to further increase the reconstruction detail with non-linear optimization methods. The advantages and disadvantages of various solvers were examined as well as a potential way of reducing certain artefacts with the use of simple Tikonov regularisation. The correlation between the conditioning of all the non-linear systems solved and some of the reconstruction artefacts, were also examined. This chapter has also shown that by using these non-linear methods one can get away with a highly sub-sampled initialization and therefore improve computational efficiency, while using a warped space-time window.

These iterative non-linear methods usually reserved for computationally expensive off-line reconstruction were shown to run in real-time on GPUs by using a multi-scale approach with a tailored Cholesky solver. As shown in Chapter 5, very computationally expensive algorithms that rely on linear algebra operators are well suited to highly parallel GPU architectures.

All the knowledge gained from the previous chapters was then used to create a framework and scalable system that achieves sub-pixel disparity reconstruction in real-time. Chapter 6 addressed certain performance issues with the initial GPU implementation to achieve greater speed, while the scalability of the system as a whole was analysed and profiled, in order to fine tune it to target platforms. Certain parameters were almost completely decoupled from the performance. The remaining parameters were then classified in terms of their performance implications. The result is a scalable system that can be tailored with respect to performance for real-time applications given limited computational resources as well as future hardware advances.

It is worth noting that although GPUs are very powerful computational resources, and can be leveraged to make certain types of offline algorithms run in real-time, they require significant investment in terms of development time and fine tuning. This situation is slowly changing with the advent of more powerful development and

analysis tools. This work has demonstrated that designing systems with parallelism can have many advantages, and be made to run a lot faster on GPUs. However, this is not a magic bullet to solve all problems.

7.1 Future work

The research carried out and presented here was narrowed down to focus on the correspondence problem in the context of tele-immersion. For this system to be transformed into a fully functional tele-immersive system further tasks need to be undertaken. In order to achieve this goal, one would need to extract texturing information. This could be achieved by capturing an extra frame that is not illuminated by structured light, or transforming the projector to project infra-red light and the camera setup to have some cameras capturing infra-red light while others would capture the texturing information. Once texturing information was acquired it would have to be registered with the disparity map and a compelling 3D representation of the scene would have to be rendered in real-time. This could either be done by using the disparity map and textures in an image based rendering system, or by performing a full geometric reconstruction with textures and shading. All of this heavy data load would also have to be communicated with another instance of this system, and would most probably require some form of compression. Each of these tasks represent a significant investment in time and provide opportunities to further forward the current state of the art.

With regard to the correspondence problem, further avenues of research would include looking at improving the quality of the non-linear optimization step. The work carried out by [90], minimized certain banding artefacts by applying a gradient constraint on all the optimization parameters. Although the method used was to reformulate the problem as a global optimization problem that was then solved using conjugate gradients. This approach could not be taken at the time and be made to run in real-time. However with the recent advances in GPU hardware this assumption is no longer valid and worth investigating, as well as potential alternatives that could be derived using a quasi-local optimization across neighbouring pixels. One could optimize sets of five or seven neighbouring pixels with applied constraints, an approach which could lead to superior results, and potentially a superior quality speed trade-off. Chapter

6 also indicated the further potential for yet further speed increases. The implementation could be optimized further for performance, although the trade-off between the development time needed versus the potential gains might not warrant it.

Appendix A

The following are all the tables representing the timing results for the experiments carried out in Chapter 6.

Width	Height	WinX	WinY	WinZ	Max Disparity	Total Time
160	60	3	3	8	60	0.019318
160	60	3	3	8	80	0.021846
160	60	3	3	8	100	0.025842
160	60	3	3	8	120	0.030126
160	60	3	3	8	155	0.03293
160	60	5	3	8	60	0.019314
160	60	5	3	8	80	0.021745
160	60	5	3	8	100	0.02584
160	60	5	3	8	120	0.030002
160	60	5	3	8	155	0.032984
160	60	5	5	8	60	0.018921
160	60	5	5	8	80	0.021754
160	60	5	5	8	100	0.025866
160	60	5	5	8	120	0.029991
160	60	5	5	8	155	0.032919
160	60	7	5	8	60	0.018929
160	60	7	5	8	80	0.021838
160	60	7	5	8	100	0.025689
160	60	7	5	8	120	0.029961
160	60	7	5	8	155	0.03316
160	60	11	7	8	60	0.018587
160	60	11	7	8	80	0.021887
160	60	11	7	8	100	0.026349
160	60	11	7	8	120	0.030167
160	60	11	7	8	155	0.032621

Table A.1 DP Initialization on lower half of sub-sampled images using single maximum disparity value

Width	Height	WinX	WinY	WinZ	Max Disparity	Total Time
160	60	3	3	8	60	0.019519
160	60	3	3	8	80	0.022231
160	60	3	3	8	100	0.026306
160	60	3	3	8	120	0.030257
160	60	3	3	8	155	0.033528
160	60	5	3	8	60	0.019334
160	60	5	3	8	80	0.021915
160	60	5	3	8	100	0.025877
160	60	5	3	8	120	0.030435
160	60	5	3	8	155	0.033245
160	60	5	5	8	60	0.018956
160	60	5	5	8	80	0.022216
160	60	5	5	8	100	0.025848
160	60	5	5	8	120	0.030268
160	60	5	5	8	155	0.03299
160	60	7	5	8	60	0.018996
160	60	7	5	8	80	0.022014
160	60	7	5	8	100	0.026246
160	60	7	5	8	120	0.029933
160	60	7	5	8	155	0.033287
160	60	11	7	8	60	0.018737
160	60	11	7	8	80	0.021733
160	60	11	7	8	100	0.025665
160	60	11	7	8	120	0.030276
160	60	11	7	8	155	0.032779

Table A.2 DP Initialization on upper half of sub-sampled images using single maximum disparity value

Width	Height	WinX	WinY	WinZ	Max Disparity	Total Time
160	30	3	3	8	60	0.012911
160	30	3	3	8	80	0.014835
160	30	3	3	8	100	0.017373
160	30	3	3	8	120	0.02058
160	30	3	3	8	155	0.02245
160	30	5	3	8	60	0.013179
160	30	5	3	8	80	0.014895
160	30	5	3	8	100	0.017538
160	30	5	3	8	120	0.020184
160	30	5	3	8	155	0.022689
160	30	5	5	8	60	0.012726
160	30	5	5	8	80	0.014875
160	30	5	5	8	100	0.01737
160	30	5	5	8	120	0.020119
160	30	5	5	8	155	0.022634
160	30	7	5	8	60	0.012839
160	30	7	5	8	80	0.014878
160	30	7	5	8	100	0.017481
160	30	7	5	8	120	0.020329
160	30	7	5	8	166	0.023754
160	30	7	5	8	155	0.02298
160	30	11	7	8	60	0.012849
160	30	11	7	8	80	0.014991
160	30	11	7	8	100	0.017717
160	30	11	7	8	120	0.020508
160	30	11	7	8	155	0.023263

Table A.3 DP Initialization on a band of pixels across centre of sub-sampled images using single maximum disparity value

Width	Height	WinX	WinY	WinZ	Max D	Total	Total With 1/4 Scale
640	240	3	3	8	60	0.165479	0.184066
640	240	3	3	8	80	0.182095	0.200682
640	240	3	3	8	100	0.219489	0.238076
640	240	3	3	8	120	0.286028	0.304615
640	240	3	3	8	160	0.350452	0.369039
640	240	5	3	8	60	0.144789	0.163376
640	240	5	3	8	80	0.183502	0.202089
640	240	5	3	8	100	0.206733	0.22532
640	240	5	3	8	120	0.288652	0.307239
640	240	5	3	8	160	0.345039	0.363626
640	240	5	5	8	60	0.144024	0.162611
640	240	5	5	8	80	0.174883	0.19347
640	240	5	5	8	100	0.208629	0.227216
640	240	5	5	8	120	0.283642	0.302229
640	240	5	5	8	160	0.335201	0.353788
640	240	7	5	8	60	0.145407	0.163994
640	240	7	5	8	80	0.174369	0.192956
640	240	7	5	8	100	0.219911	0.238498
640	240	7	5	8	120	0.281762	0.300349
640	240	7	5	8	160	0.354112	0.372699
640	240	11	7	8	60	0.14787	0.166457
640	240	11	7	8	80	0.176252	0.194839
640	240	11	7	8	100	0.219396	0.237983
640	240	11	7	8	120	0.284766	0.303353
640	240	11	7	8	160	0.333696	0.352283

Table A.4 DP Initialization on lower half of images using per-pixel maximum disparity values

Width	Height	WinX	WinY	WinZ	Max D	Total	Total With 1/4 Scale
640	240	3	3	8	60	0.156679	0.175416
640	240	3	3	8	80	0.202651	0.221388
640	240	3	3	8	100	0.246394	0.265131
640	240	3	3	8	120	0.301674	0.320411
640	240	3	3	8	160	0.367335	0.386072
640	240	5	3	8	60	0.156501	0.175238
640	240	5	3	8	80	0.203677	0.222414
640	240	5	3	8	100	0.230897	0.249634
640	240	5	3	8	120	0.299583	0.31832
640	240	5	3	8	160	0.363254	0.381991
640	240	5	5	8	60	0.158542	0.177279
640	240	5	5	8	80	0.193204	0.211941
640	240	5	5	8	100	0.236176	0.254913
640	240	5	5	8	120	0.299058	0.317795
640	240	5	5	8	160	0.356453	0.37519
640	240	7	5	8	60	0.159145	0.177882
640	240	7	5	8	80	0.194451	0.213188
640	240	7	5	8	100	0.239747	0.258484
640	240	7	5	8	120	0.297973	0.31671
640	240	7	5	8	160	0.350229	0.368966
640	240	11	7	8	60	0.163194	0.181931
640	240	11	7	8	80	0.193277	0.212014
640	240	11	7	8	100	0.250774	0.269511
640	240	11	7	8	120	0.296108	0.314845
640	240	11	7	8	160	0.345818	0.364555

Table A.5 DP Initialization on upper half of images using per-pixel maximum disparity values

Width	Height	WinX	WinY	WinZ	Max D	Total	Total With 1/4 Scale
640	120	3	3	8	60	0.088528	0.101377
640	120	3	3	8	80	0.109155	0.122004
640	120	3	3	8	100	0.131645	0.144494
640	120	3	3	8	120	0.166528	0.179377
640	120	3	3	8	160	0.203177	0.216026
640	120	5	3	8	60	0.083047	0.095896
640	120	5	3	8	80	0.109776	0.122625
640	120	5	3	8	100	0.127208	0.140057
640	120	5	3	8	120	0.168532	0.181381
640	120	5	3	8	160	0.203596	0.216445
640	120	5	5	8	60	0.083789	0.096638
640	120	5	5	8	80	0.103828	0.116677
640	120	5	5	8	100	0.126737	0.139586
640	120	5	5	8	120	0.166786	0.179635
640	120	5	5	8	160	0.197508	0.210357
640	120	7	5	8	60	0.082141	0.09499
640	120	7	5	8	80	0.105938	0.118787
640	120	7	5	8	100	0.133873	0.146722
640	120	7	5	8	120	0.164237	0.177086
640	120	7	5	8	160	0.196255	0.209104
640	120	11	7	8	60	0.082655	0.095504
640	120	11	7	8	80	0.10593	0.118779
640	120	11	7	8	100	0.131294	0.144143
640	120	11	7	8	120	0.160459	0.173308
640	120	11	7	8	160	0.192882	0.205731

Table A.6 DP Initialization on a band of pixels across centre of images using per-pixel maximum disparity values

Width	Height	WinX	WinY	WinZ	Max D	Total	Total With 1/4 Scale
320	240	3	3	8	60	0.07745	0.109555
320	240	3	3	8	80	0.088801	0.120906
320	240	3	3	8	100	0.102876	0.134981
320	240	3	3	8	120	0.119842	0.151947
320	240	3	3	8	160	0.141475	0.17358
320	240	5	3	8	60	0.077406	0.109511
320	240	5	3	8	80	0.08878	0.120885
320	240	5	3	8	100	0.09976	0.131865
320	240	5	3	8	120	0.119502	0.151607
320	240	5	3	8	160	0.140301	0.172406
320	240	5	5	8	60	0.075176	0.107281
320	240	5	5	8	80	0.083864	0.115969
320	240	5	5	8	100	0.100662	0.132767
320	240	5	5	8	120	0.117055	0.14916
320	240	5	5	8	160	0.136603	0.168708
320	240	7	5	8	60	0.076013	0.108118
320	240	7	5	8	80	0.082892	0.114997
320	240	7	5	8	100	0.100741	0.132846
320	240	7	5	8	120	0.117993	0.150098
320	240	7	5	8	160	0.134012	0.166117
320	240	11	7	8	60	0.075266	0.107371
320	240	11	7	8	80	0.083043	0.115148
320	240	11	7	8	100	0.099513	0.131618
320	240	11	7	8	120	0.11451	0.146615
320	240	11	7	8	160	0.130876	0.162981

Table A.7 DP Initialization on sub-sampled images using per-pixel maximum disparity values

Width	Height	WinX	WinY	WinZ	Max D	Total	Total With 1/4 Scale
320	120	3	3	8	60	0.039794	0.058381
320	120	3	3	8	80	0.044995	0.063582
320	120	3	3	8	100	0.051792	0.070379
320	120	3	3	8	110	0.053251	0.071838
320	120	3	3	8	120	0.060543	0.07913
320	120	3	3	8	160	0.069888	0.088475
320	120	5	3	8	60	0.039338	0.057925
320	120	5	3	8	80	0.044969	0.063556
320	120	5	3	8	100	0.050439	0.069026
320	120	5	3	8	120	0.058762	0.077349
320	120	5	3	8	160	0.069343	0.08793
320	120	5	5	8	60	0.038711	0.057298
320	120	5	5	8	80	0.042888	0.061475
320	120	5	5	8	100	0.050752	0.069339
320	120	5	5	8	120	0.059247	0.077834
320	120	5	5	8	160	0.068639	0.087226
320	120	7	5	8	60	0.038959	0.057546
320	120	7	5	8	80	0.044382	0.062969
320	120	7	5	8	100	0.051379	0.069966
320	120	7	5	8	120	0.061025	0.079612
320	120	7	5	8	160	0.068321	0.086908
320	120	11	7	8	60	0.039244	0.057831
320	120	11	7	8	80	0.043297	0.061884
320	120	11	7	8	100	0.050982	0.069569
320	120	11	7	8	120	0.060151	0.078738
320	120	11	7	8	160	0.067954	0.086541

Table A.8 DP Initialization on lower half of sub-sampled images using per-pixel maximum disparity values

Width	Height	WinX	WinY	WinZ	Max D	Total	Total With 1/4 Scale
320	120	3	3	8	60	0.041515	0.060252
320	120	3	3	8	80	0.046302	0.065039
320	120	3	3	8	100	0.053421	0.072158
320	120	3	3	8	120	0.062382	0.081119
320	120	3	3	8	160	0.072029	0.090766
320	120	5	3	8	60	0.041588	0.060325
320	120	5	3	8	80	0.046695	0.065432
320	120	5	3	8	100	0.052341	0.071078
320	120	5	3	8	120	0.060411	0.079148
320	120	5	3	8	160	0.071587	0.090324
320	120	5	5	8	60	0.041189	0.059926
320	120	5	5	8	80	0.045012	0.063749
320	120	5	5	8	100	0.053265	0.072002
320	120	5	5	8	120	0.061244	0.079981
320	120	5	5	8	160	0.070257	0.088994
320	120	7	5	8	60	0.040888	0.059625
320	120	7	5	8	80	0.045495	0.064232
320	120	7	5	8	100	0.053429	0.072166
320	120	7	5	8	120	0.062711	0.081448
320	120	7	5	8	160	0.069546	0.088283
320	120	11	7	8	60	0.0409	0.059637
320	120	11	7	8	80	0.045129	0.063866
320	120	11	7	8	100	0.052574	0.071311
320	120	11	7	8	120	0.061402	0.080139
320	120	11	7	8	160	0.069733	0.08847

Table A.9 DP Initialization on upper half of sub-sampled images using per-pixel maximum disparity values

Width	Height	WinX	WinY	WinZ	Max D	Total	Total With 1/4 Scale
320	60	3	3	8	60	0.025223	0.038072
320	60	3	3	8	80	0.028588	0.041437
320	60	3	3	8	100	0.032359	0.045208
320	60	3	3	8	120	0.037231	0.05008
320	60	3	3	8	160	0.042479	0.055328
320	60	5	3	8	60	0.025022	0.037871
320	60	5	3	8	80	0.028817	0.041666
320	60	5	3	8	100	0.03184	0.044689
320	60	5	3	8	120	0.036838	0.049687
320	60	5	3	8	160	0.042434	0.055283
320	60	5	5	8	60	0.024718	0.037567
320	60	5	5	8	80	0.027549	0.040398
320	60	5	5	8	100	0.032065	0.044914
320	60	5	5	8	120	0.036162	0.049011
320	60	5	5	8	160	0.040751	0.0536
320	60	7	5	8	60	0.024916	0.037765
320	60	7	5	8	80	0.027205	0.040054
320	60	7	5	8	100	0.032552	0.045401
320	60	7	5	8	120	0.03676	0.049609
320	60	7	5	8	160	0.041249	0.054098
320	60	11	7	8	60	0.025096	0.037945
320	60	11	7	8	80	0.027169	0.040018
320	60	11	7	8	100	0.031624	0.044473
320	60	11	7	8	120	0.036194	0.049043
320	60	11	7	8	160	0.040899	0.053748

Table A.10 DP Initialization on a band of pixels across centre of sub-sampled images using per-pixel maximum disparity values

Iterations	Width	Height	WinX	WinY	WinZ	Total
2	640	240	3	3	8	0.012888
3	640	240	3	3	8	0.014895
4	640	240	3	3	8	0.01867
5	640	240	3	3	8	0.021763
2	640	240	5	3	8	0.01605
3	640	240	5	3	8	0.021394
4	640	240	5	3	8	0.026233
5	640	240	5	3	8	0.031395
2	640	240	5	5	8	0.021682
3	640	240	5	5	8	0.031489
4	640	240	5	5	8	0.03888
5	640	240	5	5	8	0.048038
2	640	240	7	5	8	0.02969
3	640	240	7	5	8	0.041534
4	640	240	7	5	8	0.051617
5	640	240	7	5	8	0.064171
2	640	240	11	7	8	0.057607
3	640	240	11	7	8	0.082319
4	640	240	11	7	8	0.107273
5	640	240	11	7	8	0.134223

Table A.11 Non-Linear Optimization GPU Full Resolution upper half

Iterations	Width	Height	WinX	WinY	WinZ	Total
2	640	120	3	3	8	0.007224
3	640	120	3	3	8	0.009396
4	640	120	3	3	8	0.010624
5	640	120	3	3	8	0.013031
2	640	120	5	3	8	0.009185
3	640	120	5	3	8	0.012491
4	640	120	5	3	8	0.014988
5	640	120	5	3	8	0.017917
2	640	120	5	5	8	0.012637
3	640	120	5	5	8	0.01746
4	640	120	5	5	8	0.021851
5	640	120	5	5	8	0.026076
2	640	120	7	5	8	0.016958
3	640	120	7	5	8	0.022562
4	640	120	7	5	8	0.027577
5	640	120	7	5	8	0.034599
2	640	120	11	7	8	0.031533
3	640	120	11	7	8	0.043383
4	640	120	11	7	8	0.055518
5	640	120	11	7	8	0.068754

Table A.12 Non-Linear Optimization GPU Full Resolution band of pixels across centre

Iterations	Width	Height	WinX	WinY	WinZ	Total
2	320	240	3	3	8	0.006222
3	320	240	3	3	8	0.009119
4	320	240	3	3	8	0.010453
5	320	240	3	3	8	0.012873
2	320	240	5	3	8	0.008448
3	320	240	5	3	8	0.012672
4	320	240	5	3	8	0.014858
5	320	240	5	3	8	0.017909
2	320	240	5	5	8	0.012817
3	320	240	5	5	8	0.017166
4	320	240	5	5	8	0.021009
5	320	240	5	5	8	0.025942
2	320	240	7	5	8	0.015584
3	320	240	7	5	8	0.02285
4	320	240	7	5	8	0.027692
5	320	240	7	5	8	0.033684
2	320	240	11	7	8	0.030931
3	320	240	11	7	8	0.043415
4	320	240	11	7	8	0.055854
5	320	240	11	7	8	0.068816

Table A.13 Non-Linear Optimization GPU Half Resolution

Iterations	Width	Height	WinX	WinY	WinZ	Total
2	320	120	3	3	8	0.004655
3	320	120	3	3	8	0.005688
4	320	120	3	3	8	0.007181
5	320	120	3	3	8	0.008169
2	320	120	5	3	8	0.005585
3	320	120	5	3	8	0.007336
4	320	120	5	3	8	0.009345
5	320	120	5	3	8	0.011709
2	320	120	5	5	8	0.008179
3	320	120	5	5	8	0.010784
4	320	120	5	5	8	0.013186
5	320	120	5	5	8	0.015827
2	320	120	7	5	8	0.009807
3	320	120	7	5	8	0.013388
4	320	120	7	5	8	0.016531
5	320	120	7	5	8	0.02004
2	320	120	11	7	8	0.017377
3	320	120	11	7	8	0.025303
4	320	120	11	7	8	0.031435
5	320	120	11	7	8	0.038482

Table A.14 Non-Linear Optimization GPU Half Resolution Lower Half

Iterations	Width	Height	WinX	WinY	WinZ	Total
2	320	120	3	3	8	0.005291
3	320	120	3	3	8	0.005797
4	320	120	3	3	8	0.007321
5	320	120	3	3	8	0.008346
2	320	120	5	3	8	0.005699
3	320	120	5	3	8	0.007872
4	320	120	5	3	8	0.009463
5	320	120	5	3	8	0.011045
2	320	120	5	5	8	0.007847
3	320	120	5	5	8	0.010424
4	320	120	5	5	8	0.013622
5	320	120	5	5	8	0.016032
2	320	120	7	5	8	0.009916
3	320	120	7	5	8	0.013339
5	320	120	7	5	8	0.02036
2	320	120	11	7	8	0.018388
3	320	120	11	7	8	0.026093
4	320	120	11	7	8	0.032076
5	320	120	11	7	8	0.039708

Table A.15 Non-Linear Optimization GPU Half Resolution Upper Half

Iterations	Width	Height	WinX	WinY	WinZ	Total
2	320	60	3	3	8	0.003641
3	320	60	3	3	8	0.004503
4	320	60	3	3	8	0.00595
5	320	60	3	3	8	0.007158
2	320	60	5	3	8	0.004631
3	320	60	5	3	8	0.006028
4	320	60	5	3	8	0.008386
5	320	60	5	3	8	0.009724
2	320	60	5	5	8	0.006984
3	320	60	5	5	8	0.009393
4	320	60	5	5	8	0.011973
5	320	60	5	5	8	0.013988
2	320	60	7	5	8	0.008572
3	320	60	7	5	8	0.011809
4	320	60	7	5	8	0.015428
5	320	60	7	5	8	0.018227
2	320	60	11	7	8	0.015604
3	320	60	11	7	8	0.02325
4	320	60	11	7	8	0.028326
5	320	60	11	7	8	0.038265

Table A.16 Non-Linear Optimization GPU Half Resolution Band of Pixels across centre

Iterations	Width	Height	WinX	WinY	WinZ	Total
2	160	120	3	3	8	0.003559
3	160	120	3	3	8	0.00514
4	160	120	3	3	8	0.006056
5	160	120	3	3	8	0.007
2	160	120	5	3	8	0.004653
3	160	120	5	3	8	0.006306
4	160	120	5	3	8	0.00865
5	160	120	5	3	8	0.00942
2	160	120	5	5	8	0.006485
3	160	120	5	5	8	0.008835
4	160	120	5	5	8	0.011618
5	160	120	5	5	8	0.01466
2	160	120	7	5	8	0.008267
3	160	120	7	5	8	0.012209
4	160	120	7	5	8	0.014238
5	160	120	7	5	8	0.016974
2	160	120	11	7	8	0.01616
3	160	120	11	7	8	0.02327
4	160	120	11	7	8	0.029266
5	160	120	11	7	8	0.035637

Table A.17 Non-Linear Optimization GPU Quarter Resolution

Iterations	Width	Height	WinX	WinY	WinZ	Total
2	160	60	3	3	8	0.002989
3	160	60	3	3	8	0.004615
4	160	60	3	3	8	0.005012
5	160	60	3	3	8	0.00642
2	160	60	5	3	8	0.004557
3	160	60	5	3	8	0.005815
4	160	60	5	3	8	0.00778
5	160	60	5	3	8	0.009357
2	160	60	5	5	8	0.005649
3	160	60	5	5	8	0.00883
4	160	60	5	5	8	0.010816
5	160	60	5	5	8	0.013571
2	160	60	7	5	8	0.008017
3	160	60	7	5	8	0.010675
4	160	60	7	5	8	0.0145
5	160	60	7	5	8	0.019734
2	160	60	11	7	8	0.01526
3	160	60	11	7	8	0.022661
4	160	60	11	7	8	0.029733
5	160	60	11	7	8	0.036434

Table A.18 Non-Linear Optimization GPU Quarter Resolution Lower Half

Iterations	Width	Height	WinX	WinY	WinZ	Total
2	160	60	3	3	8	0.003404
3	160	60	3	3	8	0.003984
4	160	60	3	3	8	0.00561
5	160	60	3	3	8	0.006172
2	160	60	5	3	8	0.004195
3	160	60	5	3	8	0.005931
4	160	60	5	3	8	0.007641
5	160	60	5	3	8	0.009607
2	160	60	5	5	8	0.006146
3	160	60	5	5	8	0.008837
4	160	60	5	5	8	0.010298
5	160	60	5	5	8	0.01361
2	160	60	7	5	8	0.007813
3	160	60	7	5	8	0.011388
4	160	60	7	5	8	0.015079
5	160	60	7	5	8	0.017085
2	160	60	11	7	8	0.013919
3	160	60	11	7	8	0.022546
4	160	60	11	7	8	0.029243
5	160	60	11	7	8	0.033452

Table A.19 Non-Linear Optimization GPU Quarter Resolution Upper Half

Iterations	Width	Height	WinX	WinY	WinZ	Total
2	160	30	3	3		
3	160	30	3	3		
4	160	30	3	3		
5	160	30	3	3		
2	160	30	5	3		
3	160	30	5	3		
4	160	30	5	3		
5	160	30	5	3		
2	160	30	5	5		
3	160	30	5	5		
4	160	30	5	5		
5	160	30	5	5		
2	160	30	7	5		
3	160	30	7	5		
4	160	30	7	5		
5	160	30	7	5		
2	160	30	11	7		
3	160	30	11	7		
4	160	30	11	7		
5	160	30	11	7		

Table A.20 Non-Linear Optimization GPU Quarter Resolution Band of Pixels across centre

Bibliography

[1] 2D3. Boujou. http://www.2d3.com/.

[2] AMTEL. Avr stk500 development board. http://www.atmel.com/dyn/Products/-tools_card.asp?tool_id=2735.

[3] BAILLARD, C., AND ZISSERMAN, A. A plane-sweep strategy for the 3d reconstruction of buildings from multiple images. In *ISPRS Journal of Photogrammetry and Remote Sensing* (2000), pp. 56–62.

[4] BAKER, H. H., BHATTI, N. T., TANGUAY, D., SOBEL, I., GELB, D., GOSS, M. E., MACCORMICK, J., YUASA, K., CULBERTSON, W. B., AND MALZBENDER, T. Computation and performance issues in coliseum: an immersive videoconferencing system. In *ACM Multimedia'03* (2003), pp. 470–479.

[5] BAKER, S., GROSS, R., AND MATTHEWS, I. Lucas-kanade 20 years on: A unifying framework: Part 4. *International Journal of Computer Vision 56* (2004), 221–255.

[6] BAY, H., ESS, A., TUYTELAARS, T., AND VAN GOOL, L. Speeded-up robust features (surf). *Computer Vision and Image Understanding (CVIU) 110*, 3 (June 2008), 346–359.

[7] BIRCHFIELD, S. Derivation of kanade-lucas-tomasi tracking equation. Unpublished: http://www.ces.clemson.edu/ stb/klt/birchfield-klt-derivation.pdf.

[8] BIRCHFIELD, S., AND TOMASI, C. Depth discontinuities by pixel-to-pixel stereo. *International Journal of Computer Vision 35* (1996), 1073–1080.

[9] BORSHUKOV, G., PIPONI, D., LARSEN, O., LEWIS, J. P., AND TEMPELAAR-LIETZ, C. Universal capture - image-based facial animation for "the matrix reloaded". In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses* (New York, NY, USA, 2005), ACM.

[10] BOUGUET, J.-Y. Camera calibration toolbox for matlab.

[11] BOYKOV, Y., VEKSLER, O., AND ZABIH, R. A new algorithm for energy minimization with discontinuities. *In International Workshop on Energy Minimization Methods in Computer Vision and Pattern Recognition 1* (1999), 26–29.

[12] BRADSKI, G., AND KAEHLER, A. *Learning OpenCV: Computer Vision with the OpenCV Library*, 1st ed. O'Reilly Media, Inc., October 2008.

[13] CASPI, D., KIRYATI, N., AND SHAMIR, J. Range imaging with adaptive color structured light. *Pattern Analysis and Machine Intelligence, IEEE Transactions on 20*, 5 (may 1998), 470 –480.

[14] CHANG, J. Y., PARK, H., PARK, I. K., LEE, K. M., AND LEE, S. U. Gpu-friendly multi-view stereo reconstruction using surfel representation and graph cuts. *Comput. Vis. Image Underst. 115* (May 2011), 620–634.

[15] CHEN, C. H., AND KAK, A. C. Modelling and calibration of a structured light scanner for 3d robot vision. In *IEEE Conference on Robotics and Automation* (1987), pp. 807–815.

[16] CONDER, M. Explicit definition of the binary reflected gray codes. *Discrete Mathematics 195*, 1-3 (1999), 245 – 249.

[17] COTTING, D., NAEF, M., GROSS, M., AND FUCHS, H. Embedding imperceptible patterns into projected images for simultaneous acquisition and display. In *Third IEEE and ACM International Symposium on Mixed and Augmented Reality (ISMAR 2004)* (2-5 Nov. 2004), pp. 100–109.

[18] CRIMINISI, A., REID, I. D., AND ZISSERMAN, A. Single view metrology. In *ICCV* (1999), pp. 434–441.

[19] CRIMINISI, A., SHOTTON, J., BLAKE, A., AND TORR, P. Gaze manipulation for one-to-one teleconferencing. In *Proceedings. Ninth IEEE International Conference on Computer Vision* (2003), vol. 1, pp. 191–198.

[20] DEBEVEC, P. E., TAYLOR, C. J., AND MALIK, J. Modeling and rendering architecture from photographs: a hybrid geometry- and image-based approach. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1996), SIGGRAPH '96, ACM, pp. 11–20.

[21] EPPSTEIN, D. The farthest point delaunay triangulation minimizes angles. *Computational Geometry Theory & Applications 1*, 3 (March 1992), 143–148.

[22] FORSYTH, D. A., AND PONCE, J. *Computer Vision: A Modern Approach*, us ed ed. Prentice Hall, August 2002.

[23] FURUKAWA, R., AND KAWASAKI, H. Dense 3d reconstruction with an uncalibrated stereo system using coded structured light. In *Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05) - Workshops - Volume 03* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 107–.

[24] FUSIELLO, A., TRUCCO, E., AND VERRI, A. A compact algorithm for rectification of stereo pairs. *Machine Vision and Applications 12*, 1 (2000), 16–22.

[25] GARLAND, M., LE GRAND, S., NICKOLLS, J., ANDERSON, J., HARDWICK, J., MORTON, S., PHILLIPS, E., ZHANG, Y., AND VOLKOV, V. Parallel computing experiences with cuda. *Micro, IEEE 28*, 4 (july-aug. 2008), 13 –27.

[26] GENG, J. Structured-light 3d surface imaging: a tutorial. *Adv. Opt. Photon. 3*, 2 (Jun 2011), 128–160.

[27] GONG, M., AND YANG, Y.-H. Near real-time reliable stereo matching using programmable graphics hardware. In *Proc. IEEE Computer Society Conf. Computer Vision and Pattern Recognition CVPR 2005* (2005), vol. 1, pp. 924–931.

[28] GRAPHICS, O. C., SADAGIC, A., TOWLES, H., HOLDEN, L., DANIILIDIS, K., AND ZELEZNIK, B. Tele-immersion portal: Towards an ultimate synthesis. In *of Computer Graphics and Computer Vision Systems, Proceedings of 4th Annual International Workshop on Presence* (2001).

[29] GROSS, M., WÜRMLIN, S., NAEF, M., LAMBORAY, E., SPAGNO, C., KUNZ, A., KOLLER-MEIER, E., SVOBODA, T., GOOL, L. V., LANG, S., STREHLKE, K., MOERE, A. V., OLIVER, ZÜRICH, E., AND STAADT, O. blue-c: A spatially immersive display and 3d video portal for telepresence. In *ACM Transactions on Graphics* (2003), pp. 819–827.

[30] HALL-HOLT, O., AND RUSINKIEWICZ, S. Stripe boundary codes for real-time structured-light range scanning of moving objects. In *Computer Vision, 2001. ICCV 2001. Proceedings. Eighth IEEE International Conference on* (2001), vol. 2, pp. 359 –366 vol.2.

[31] HARRIS, C., AND STEPHENS, M. A combined corner and edge detection. In *Proceedings of The Fourth Alvey Vision Conference* (1988), pp. 147–151.

[32] HARTLEY, R., AND ZISSERMAN, A. *Multiple View Geometry in Computer Vision*. Cambridge University Press, March 2004.

[33] HASENFRATZ, J.-M., LAPIERRE, M., AND SILLION, F. A real-time system for full body interaction with virtual worlds. *Eurographics Symposium on Virtual Environments 1* (2004), 147–156.

[34] HELD, W., ABADI, A. K., AND WENDT, V. *3ds max 7*. bhv, Bonn, 2005.

[35] HERTZMANN, A., AND SEITZ, S. M. Shape and materials by example: A photometric stereo approach. *IEEE Computer Society Conference on Computer Vision and Pattern Recognition 1* (2003), 533.

[36] HORN, E., AND KIRYATI, N. Toward optimal structured light patterns. *Image and Vision Computing 17*, 2 (1999), 87 – 97.

[37] INOKUCHI, S., SATO, K., AND MATSUDA, F. Range imaging system for 3-d object recognition. In *International Conference on Pattern Recognition* (1984).

[38] INTEL COORPERATION. Documents intel math kernel library release 7.0.1 doc, 2004.

[39] JUNG, J. H., AND O'LEARY, D. P. Cholesky decomposition and linear programming on a gpu. In *Workshop on Edge Computing Using New Commodity Architectures (EDGE)* (Chapel Hill, North Carolina, May 2006).

[40] JUNG, S.-H., AND BAJCSY, R. A framework for constructing real-time immersive environments for training physical activities. *Journal of Multimedia 1*, 7 (2006), 9–17.

[41] KANADE, T., RANDER, P., VEDULA, S., AND SAITO, H. Virtualized reality: Digitizing a 3d time-varying event as is and in real time. In *Mixed Reality, Merging Real and Virtual Worlds*, H. T. Yuichi Ohta, Ed. Springer-Verlag, 1999, pp. 41–57.

[42] KLETTE, R., SCHHLUNS, K., AND KOSCHAN, A. *Computer Vision Three Dimensional Data from Images*. Springer, 1998.

[43] KOLMOGOROV, V., AND ZABIH, R. Computing visual correspondence with occlusions via graph cuts. In *International Conference on Computer Vision* (2001), pp. 508–515.

[44] KRUPPA, E. Zur ermittlung eines objektes aus zwei perspektiven mit innerer orientierung. *Other Journal* (1913), 1939–1948.

[45] KURASHIMA, C. S., YANG, R., AND LASTRA, A. Combining approximate geometry with view-dependent texture mapping - a hybrid approach to 3d video teleconferencing. In *in Proc. SIBGRAPI* (2002), pp. 112–119.

[46] KURILLO, G., VASUDEVAN, R., LOBATON, E., AND BAJCSY, R. A framework for collaborative real-time 3d teleimmersion in a geographically distributed environment. In *Tenth IEEE International Symposium on Multimedia (ISM2008)* (dec. 2008), pp. 111 –118.

[47] KUTULAKOS, K. N., AND SEITZ, S. M. A theory of shape by space carving. *International Journal of Computer Vision 38* (2000), 307–314.

[48] LONGUET HIGGINS, H., AND PRAZDNY, K. The interpretation of a moving retinal image. *Proceedings of the Royal Society of London. Series B, Biological Sciences B-208* (1980), 385–397.

[49] LOURAKIS, M. I. A. A brief description of the levenberg-marquardt algorithm implemented by levmar. foundation for research and technology, 2005.

[50] LOWE, D. Object recognition from local scale-invariant features. In *International Conference on Computer Vision* (1999), pp. 1150–1157.

[51] LUCAS, B., AND KANADE, T. An iterative image registration technique with an application to stereo vision. In *Proceedings of the 7th International Joint Conference on Artificial Intelligence (IJCAI '81)* (April 1981), pp. 674–679.

[52] MARROQUIM, R., AND MAXIMO, A. Introduction to gpu programming with glsl. *Tutorials of the Brazilian Symposium on Computer Graphics and Image Processing 1* (2009), 3–16.

[53] MATHWORKS. Matlab optimization toolbox user's guide. http://www.mathworks.co.uk/access/-helpdesk/help/pdf_doc/optim/optim_tb.pdf.

[54] MATUSIK, W., BUEHLER, C., RASKAR, R., GORTLER, S. J., AND MCMILLAN, L. Image-based visual hulls. In *SIGGRAPH'00* (2000), pp. 369–374.

[55] MOSLAH, O., VALLES-SUCH, A., GUITTENY, V., COUVET, S., AND PHILIPP-FOLIGUET, S. Accelerated multi-view stereo using parallel processing capababilities of the gpus. In *3DTV Conference* (Postdam, DE, 2009).

[56] MULLIGAN, J., AND DANIILIDIS, K. Real time trinocular stereo for tele-immersion. In *Image Processing, 2001. Proceedings. 2001 International Conference on* (2001), vol. 3, pp. 959 –962 vol.3.

[57] NAHMIAS, J.-D., STEED, A., AND BUXTON, B. Evaluation of modern dynamic programming algorithms for realtime active stereo systems. In *WSCG (Short Papers)* (2005), pp. 113–116.

[58] NAHMIAS, J.D., S. A. B. B. Analysis of cost functions and structured light patterns for modern dynamic programming stereo algorithms. In *IEE International Conference on Visual Information Engineering 05* (April 2005).

[59] NISTER, D. Preemptive ransac for live structure and motion estimation. In *Proc. Ninth IEEE Int Computer Vision Conf* (2003), pp. 199–206.

[60] NVIDIA CORPORATION. *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*. NVIDIA Corporation, 2007.

[61] OH, B. M., CHEN, M., DORSEY, J., DURAND, F., MAX, O., JULIE, C., AND DURAND, D. F. Image-based modeling and photo editing. In *Proceedings of ACM SIGGRAPH'01. ACM* (2001).

[62] OHTA, Y., AND KANADE, T. Stereo by two-level dynamic programming. In *Proceedings of the 9th international joint conference on Artificial intelligence - Volume 2* (San Francisco, CA, USA, 1985), Morgan Kaufmann Publishers Inc., pp. 1120–1126.

[63] OPENGL, SHREINER, D., WOO, M., NEIDER, J., AND DAVIS, T. *OpenGL(R) Programming Guide : The Official Guide to Learning OpenGL(R), Version 2 (5th Edition)*. Addison-Wesley Professional, August 2005.

[64] OUALI, M. H., LANGE, H., AND LAURGEAU, C. I. An energy minimization approach to dense stereovision. In *Proc. Conf. Int Image Processing* (1996), vol. 1, pp. 841–845.

[65] PARK, S.-Y., PARK, G.-G., AND ZHANG, L. An easy camera-projector calibration technique for structured light 3-d reconstruction. *The Kips Transactions:partb 17B* (2010), 215–226.

[66] PHARR, M., AND FERNANDO, R. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley Professional, March 2005.

[67] PING-SING, R. Z., ZHANG, R., SING TSAI, P., CRYER, J. E., AND SHAH, M. Shape from shading: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence 21* (1999), 690–706.

[68] PRESS, W. H., TEUKOLSKY, S. A., VETTERLING, W. T., AND FLANNERY, B. P. *Numerical Recipes in C "The Art of Scientific Computing" Second Edition*. Cambridge University Press, 1992.

[69] ROCCHINI, C., CIGNONI, P., MONTANI, C., PINGI, P., AND SCOPIGNO, R. A low cost 3d scanner based on structured light. *Computer Graphics Forum 20* (2001), 299–308.

[70] SALOM, P., MEGRET, R., DONIAS, M., AND BERTHOUMIEU, Y. Dynamic picking system for 3d seismic data: Design and evaluation. *Int. J. Hum.-Comput. Stud. 67*, 7 (2009), 551–560.

[71] SALVI, J., PAGÈS, J., AND BATLLE, J. Pattern codification strategies in structured light systems. *PATTERN RECOGNITION 37* (2004), 827–849.

[72] SCHARSTEIN, D., AND SZELISKI, R. A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. *International Journal of Computer Vision 47* (2001), 7–42.

[73] SEITZ, S., AND DYER, C. Photorealistic scene reconstruction by voxel coloring. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition* (17-19 June 1997), pp. 1067–1073.

[74] SEITZ, S. M., CURLESS, B., DIEBEL, J., SCHARSTEIN, D., AND SZELISKI, R. A comparison and evaluation of multi-view stereo reconstruction algorithms. In *Proceedings of the 2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition* (Washington, DC, USA, 2006), vol. 1, IEEE Computer Society, pp. 519–528.

[75] SHEWCHUK, J. R. An introduction to the conjugate gradient method without the agonizing pain. Tech. rep., Carnegie Mellon University, August 1994.

[76] SHREINER, D., AND BOARD, O. A. R. *OpenGL reference manual : the official reference document to OpenGL, version 1.4*. Addison-Wesley, 2004.

[77] SIN, C.-H., CHENG, C.-M., LAI, S.-H., AND YANG, S.-Y. Geodesic tree-based dynamic programming for fast stereo reconstruction. In *2009 IEEE 12th International Conference on Computer Vision Workshops (ICCV Workshops)* (sep. 2009), pp. 801 –807.

[78] SINHA, S., MORDOHAI, P., AND POLLEFEYS, M. Multi-view stereo via graph cuts on the dual of an adaptive tetrahedral mesh. In *Computer Vision, 2007. ICCV 2007. IEEE 11th International Conference on* (oct. 2007), pp. 1 –8.

[79] SKOCAJ, D., AND LEONARDIS, A. Range image acquisition of objects with non-uniform albedo using structured light range sensor. In *Pattern Recognition, 2000. Proceedings. 15th International Conference on* (2000), vol. 1, pp. 778 –781 vol.1.

[80] STREILEIN, A., AND VAN DEN HEUVEL, F. A. Potential and limitation for the 3d documentation of cultural heritage from a single image. In *Proceedings XVII CIPA Symposium* (1999).

[81] THE PIXEL FARM. Pftrack. www.thepixelfarm.com.

[82] TRUCCO, AND VERRI, A. *Introductory Techniques for 3-D Computer Vision*. Prentice Hall, March 1998.

[83] ULLMAN, S. Computational studies in the interpretation of structure and motion: Summary and extension. In *Human and Machine Vision* (1983), Academic Press.

[84] WANG, G. Implementation and experimental study on fast object modeling based on multiple structured stripes. *Optics and Lasers in Engineering 42* (2004), 627–638.

[85] WANG, L., LIAO, M., GONG, M., YANG, R., AND NISTER, D. High-quality real-time stereo using adaptive cost aggregation and dynamic programming. In *3D Data Processing, Visualization, and Transmission, Third International Symposium on* (june 2006), pp. 798 –805.

[86] WILLOUGHBY, R. A. Solutions of ill-posed problems (a. n. tikhonov and v. y. arsenin). *SIAM Review 21*, 2 (1979), 266–267.

[87] YANG, R., WELCH, G., AND BISHOP, G. Real-time consensus-based scene reconstruction using commodity graphics hardware. In *Proceedings of the 10th Pacific Conference on Computer Graphics and Applications* (Washington, DC, USA, 2002), PG '02, IEEE Computer Society, pp. 225–.

[88] ZACH, C., KLAUS, A., REITINGER, B., AND KARNER, K. Optimized stereo reconstruction using 3d graphics hardware. In *In Workshop of Vision, Modelling, and Visualization (VMV 2003* (2003), pp. 119–126.

[89] ZENG, G., PARIS, S., LHUILLIER, M., AND QUAN, L. Study of volumetric methods for face reconstruction. In *Proceedings of IEEE Intelligent Automation Conference* (2003).

[90] ZHANG, L. *Spacetime stereo and its applications*. PhD thesis, Washington University, Seattle, WA, USA, 2005. Chair-Seitz, Steven M.

[91] ZHANG, L., CURLESS, B., AND SEITZ, S. Spacetime stereo: shape recovery for dynamic scenes. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition* (18-20 June 2003), vol. 2, pp. II–367–74vol.2.

[92] ZHANG, L., SNAVELY, N., CURLESS, B., AND SEITZ, S. M. Spacetime faces: High-resolution capture for modeling and animation. In *ACM Annual Conference on Computer Graphics* (August 2004), pp. 548–558.

[93] ZHANG, S., AND HUANG, P. High-resolution, real-time 3d shape acquisition. In *Computer Vision and Pattern Recognition Workshop, 2004. CVPRW '04. Conference on* (june 2004), p. 28.

[94] ZHANG, S., AND HUANG, P. S. Novel method for structured light system calibration. *Optical Engineering 45* (2006).

[95] ZHANG, Z., AND ZHANG, Z. A flexible new technique for camera calibration. *IEEE Transactions on Pattern Analysis and Machine Intelligence 22* (1998), 1330–1334.

[96] ZITNICK, C. L., AND KANADE, T. A cooperative algorithm for stereo matching and occlusion detection. *IEEE Trans. Pattern Anal. Mach. Intell. 22*, 7 (2000), 675–684.