Parallel Patch-Based Volumetric Reconstruction from Images

Robert Jermy

July 7, 2014

In fulfillment of the degree Master of Science in Computer Engineering, College of Agriculture, Engineering and Science, University of KwaZulu-Natal.

Supervisor: Mr. B. Naidoo
Co-Supervisor: Prof. J.R. Tapamo
Examiner's Copy

COLLEGE OF AGRICULTURE, ENGINEERING AND SCIENCE

DECLARATION 1 - PLAGIARISM

I, ……., declare that

1. The research reported in this thesis, except where otherwise indicated, is my original research.

2. This thesis has not been submitted for any degree or examination at any other university.

3. This thesis does not contain other persons' data, pictures, graphs or other information, unless specifically acknowledged as being sourced from other persons.

4. This thesis does not contain other persons' writing, unless specifically acknowledged as being sourced from other researchers. Where other written sources have been quoted, then:

a. Their words have been re-written but the general information attributed to them has been referenced

b. Where their exact words have been used, then their writing has been placed in italics and inside quotation marks, and referenced.

5. This thesis does not contain text, graphics or tables copied and pasted from the Internet, unless specifically acknowledged, and the source being detailed in the thesis and in the References sections.

Signed

……

CMC Feb 2012

As the candidate's Supervisor I agree/do not agree to the submission of this thesis.

Acknowledgements

I would like to thank my supervisor for his guidance and input throughout this degree as well as providing the topic and giving constructive criticism and positive feedback as required. I would also like to thank my family and fiancée for proof reading and providing feedback on my dissertation in its various incarnations and for putting up with me pushing reconstructed models in their faces at fairly regular intervals.

Abstract

Three Dimensional (3D) reconstruction relates to the creating of 3D computer models from sets of Two Dimensional (2D) images. 3D reconstruction algorithms tend to have long execution times, meaning they are ill suited to real time 3D reconstruction tasks. This is a significant limitation which this dissertation attempts to address. Modern Graphics Processing Units (GPUs) have become fully programmable and have spawned the field known as General Purpose GPU (GPGPU) processing. Using this technology it is possible to offload certain types of tasks from the Central Processing Unit (CPU) to the GPU. GPGPU processing is designed for problems that have data parallelism. This means that a particular task can be split into many smaller tasks that can run in parallel, the results of which and are not dependent upon the order in which the tasks are completed. Therefore to properly make use of both CPU parallelism and GPGPU processing a 3D reconstruction algorithm with data parallelism was required. The selected algorithm was the Patch-Based Multi-View Stereopsis (PMVS) method, proposed and implemented by Yasutaka Furukawa and Jean Ponce. This algorithm uses small oriented rectangular patches to model a surface and is broken into four major steps: Feature detection; feature matching, expansion and filtering. The reconstructed patches are independent and as such the algorithm is data parallel. Some segments of the PMVS algorithm were programmed for GPGPU and others for CPU parallelism. Results show that the feature detection stage runs 10 times faster on the GPU than the equivalent CPU implementation. The patch creation and expansion stages also benefited from GPU implementation. Which brought an improvement in the execution time of two times for large images, and equivalent execution times for small images, when compared to the CPU implementation. These results show that the use of GPGPU and CPU parallelism can indeed improve the performance of this 3D reconstruction algorithm.

Contents

List of Tables

List of Figures

List of Algorithms

Nomenclature

3D	Three Dimensional
MVS	Multi-View Stereo
CPU	Central Processing Unit
CUDA	Compute Unified Device Architechture
DoG	Difference of Gaussians
FPS	Frames per Second
GPGPU	General Purpose computing on Graphics Processing Units
GPU	Graphics Processing Unit
GSL	GNU Scientific Library
GTM	Ground Truth Model
L-BFGS	Limited Memory Broyden-Fletcher-Goldfarb-Shanno
MUTEX	Mutual Exclusion
NCC	Normalised Cross Correlation
PC	Personal Computer
PMVS	Patch-Based Multi-View Stereopsis
SVD	Singular Value Decompisition
VSTL	Vertex Shading, Transform and Lighting

Chapter 1

Introduction

The topic of Three Dimensional (3D) reconstruction from images has many different methods. All with the same goal of creating an accurate reconstruction of a 3D model from a sequence of input images. This dissertation aims to improve upon an existing 3D reconstruction algorithm. This is achieved by making use of parallel processing on the Central Processing Unit (CPU) and the Graphics Processing Unit (GPU). There is not a single algorithm that is suitable for all problems, as such an algorithm should be chosen based on the specific requirements of the problem.

1.1 Three Dimensional Reconstruction

All 3D reconstruction algorithms aim to create an accurate and detailed 3D model of some object or scene, but how they go about this task, and their suitability to a particular task, varies greatly. For example voxel based methods provide a highly scalable method of creating a model of an object. These methods however are not well suited to large scene datasets where the large number of voxels required creates a prohibitively high memory and processing requirement.

Patch based methods such as the one that is the basis of this dissertation are suited to both object and scene datasets. These methods attempt to densely cover a surface in small oriented patches so that the patches provide an accurate point cloud reconstruction. This can then be used to create an accurate and complete mesh model [1, 2]. Patch based methods are suited to parallelisation because they create a model made of many unconnected patches. This disconnectedness is also the major drawback of patch based methods. Patches do not have any inherent connectivity information and thus some amount of processing is required to determine neighbourhood information [1, 3].

In general all 3D reconstruction algorithms take a series of calibrated input images and from these create a 3D model of the object or scene in the images. A calibrated image refers to an image with known extrinsic and intrinsic camera parameters. The extrinsic parameters store the translation and rotation of the camera relative to a reference point in space. The intrinsic parameters store the internal parameters of the camera: focal length, pixel size, and principal point.

A major issue with 3D reconstruction algorithms is that they tend to take a pro-

hibitively long time to create a 3D model from calibrated input images. This problem can be mitigated by making use of parallel processing, such as multi-core methods, typical of CPU threads, and many-core methods, typical of GPU threads.

1.2 Parallelism

The CPU can be thought of as a general purpose processor. It can perform complex calculations while also running a media player that allows one to listen to music or run a word processor. For these reasons CPU threads can be thought of as semi-independent programs. Most modern personal computers (PCs) have four physical cores and four virtual cores. This allows eight independent threads to be run in parallel, allowing for a significant speed-up to processing, although this speed-up falls short of an eight times increase.

The GPU on the other hand has a very specific purpose: it is dedicated to doing floating point calculations. Only recently though has this power become available to developers in the form of OpenCL and NVIDIA's Compute Unified Device Architecture (CUDA). Where the CPU runs up to eight simultaneous and complex threads the GPU can run thousands of simultaneous light-weight threads. This type of processing power is often called "Embarrassingly Parallel Processing" [4] due to the large number of simultaneous threads.

Both types of parallelism have their uses and they are used to improve the performance of the PMVS algorithm. CPU threads are used to "supervise" a particular reference image, while GPU threads are used for smaller specific tasks during the run time of the program.

1.3 Problem Statement

This dissertation aims to build on the work done by Furukawa and Ponce. Using the Patch-Based Multi-View Stereopsis (PMVS) [1, 5] algorithm as a starting point. What impacts can parallel processing make to the execution time of the program? If parallelism does make an impact on processing which methods should be used, and where?

The general problem statement is:

> Can parallel processing, both CPU and GPU based, be used to improve the execution time of three dimensional reconstruction algorithms?

1.4 Outline of this Thesis

Chapter 2 contains a literature review, covering the work done in the field of 3D reconstruction. It includes different algorithms that create a 3D model from input sequences of images.

Chapter 3 has a detailed description of the Patch-Based Multi-View Stereopsis [1] (PMVS) algorithm developed by Furukawa and Ponce. This is the method that forms the foundation of this thesis and upon which the implementation was built.

Chapter 4 provides information about modifications that were made to the algorithm to improve its efficiency. As well as images of reconstructed models and timing information for the primary dataset.

Chapter 5 concludes this dissertation and discusses what was achieved and possible extensions to the work.

Chapter 2

Literature Review

2.1 Multi-View Stereopsis

The objective of a multi-view stereo (MVS) reconstruction algorithm is to construct an accurate three-dimensional model of an object, or scene, from a set of calibrated input images in a reasonable amount of time. There are many different types of multi-view stereo algorithms that make use of a multitude of different methods to construct a model. Work by Seitz *et al.* has determined six categories that can be used to classify the various MVS algorithms. These are *Scene Representation*, *Photo-Consistency Measure*, *Visibility Model*, *Shape Prior*, *Reconstruction Algorithm* and *Initialisation Requirements* [3]. These classifications may not fit all methods of three-dimensional reconstruction, however they do allow one to break a particular algorithm into its component parts.

2.1.1 Scene Representation

The way that a MVS algorithm represents a scene is very important to the algorithm as a whole. Certain scene representations will work better for object data sets, for example voxel based methods. These methods often require some sort of initial bounding volume like a convex hull or visual hull. Other methods work on both scene and object data sets. For example patch based methods like the one presented in this dissertation do not require a bounding volume and are thus able to adequately model scenes and objects. Depth map based methods such as the depth map fusion method [6, 7] create depth maps from multiple views and do not require a bounding volume. This makes them acceptable for both object and scene datasets. These methods create scalable models of the object or scene as they run, rather than carving away at a fixed virtual block like voxel based methods.

Voxels based methods represent the 3D geometry of an object as a series of discrete voxels. The accuracy of the 3D reconstruction can be changed by altering the resolution of the voxel grid. Unfortunately for voxel based methods to work properly a bounding volume is required around the scene. As such voxel based methods will only work on object data sets, or scenes where a strict bounding volume can be defined [1, 3].

Polygonal Mesh methods represent surfaces as a series of connected planes which can

be deformed to create an accurate 3D model of the scene. These methods generally require some sort of initial model to refine such as a visual hull or convex hull [8]. This makes these methods only viable for object datasets, or scenes with a strict bounding volume [1, 3].

Depth Map Fusion methods create depth maps for each input view. These depth maps are then fused together to create a 3D model. These methods do not require an initial bounding volume and can therefore be used for objects and scenes [1, 3].

Patch based methods represent the surface of the scene with a number of patches that are oriented towards the camera that observes them. They require no bounding volume or visual hull and can model both objects and scenes. A final post-processing step is needed to transform the patch model into a mesh model. This is only necessary if a closed surface representation is required [1].

2.1.2 Initialisation Requirements

All MVS algorithms require an input set of calibrated images. Calibrated in this case means that the extrinsic and intrinsic parameters of the camera are known. The extrinsic parameters are the position and orientation of the camera that captured each image known relative to some to a global origin. The intrinsic parameters of the camera are the pixel size, focal length, and principal point. It is possible to generate the camera parameters using a set of uncalibrated images using a method known as Structure from Motion [9, 2] which uses common features to determine the relative positions of the cameras. For larger scene datasets a method such as bundle adjustment can be used to group images and cameras into smaller subsets [10]. Bundle adjustment attempts to group and optimise bundles of vectors based on their proximity which makes it suited to such problems. Some methods require more input information, such as a visual hull or bounding volume around the object that was modeled. While other algorithms require silhouette images to construct a visual hull [8, 3].

2.1.3 Photo-Consistency Measure

The photo-consistency measure is used to measure the visual compatibility between the different input images. Most of these measures compare a set of pixels in one image to a corresponding set of pixels in another. The photo-consistency measure can be defined as either a scene space or an image space method [3].

Scene space methods operate by taking a patch from one reference image and projecting it into the other input images to determine the amount of agreement between them. There are a number of methods that can be used to determine the agreement between images, such as the variance of projected pixels in the images [3].

Image space methods use an estimate of the scene geometry to warp an image from the reference viewpoint to predict a different viewpoint. The predicted image is then compared to the measured image to yield a photo-consistency measure called the prediction error [3].

2.1.4 Visibility Model

The purpose of a visibility model is to determine which views are considered when evaluating photo-consistency. This stage ensures that images that may not have a clear view of the object or scene are not used when measuring photo-consistency. The visibility model is also important for dealing with occlusions. This is a necessity for all modern MVS algorithms if they are going to be used on real world datasets. Three techniques for dealing with visibility are detailed below.

Geometric techniques try to model the image formation process as well as the shape of the scene to determine which parts of the scene are visible in the images [3, 11, 12].

Quasi-Geometric techniques attempt to estimate visibility relationships based on approximate geometric reasoning. A popular method is to ensure that photo-consistency analyses are done between cameras that have similar positions and orientations [3, 13].

Outlier Based methods just treat all occlusions as outliers, this technique is useful when there are a sufficient number of good views. One of the most commonly used methods is to avoid comparing views that are far apart [3].

2.1.5 Shape Prior

Shape priors are used to assist the MVS algorithm to converge to the correct surface. They do this by giving the algorithm a reconstruction bias. Some methods try to minimise the scene based photo-consistency and typically seek minimal surfaces with small surface area; however these methods tend to smooth over surfaces with high curvature [3].

There are other methods that favour maximal surfaces. These methods tend to be space carving methods that remove voxels only if they are not photo-consistent [3].

2.1.6 Evaluation of MVS Results

One of the major problems with MVS algorithms is that it can be difficult to compare the results of one algorithm to those of another. This means that more often than not the only measure of a 3D reconstruction algorithm's accuracy is qualitative rather than quantitative. This may have been one of the reasons that research in the field of multi-view stereo 3D reconstruction was fairly slow when compared to binocular 3D reconstruction [3]. Seitz *et al.* proposed a method for quantitatively comparing MVS algorithms in their paper "A Comparison and Evaluation of Multi-View Stereo Reconstruction Algorithms" [3]. Their paper covers a number of state-of-the-art MVS algorithms and classifies the various implementations, before finally offering a quantitative measure of the accuracy of the reconstruction.

The authors decided on two objects that could be used to test these algorithms. A sample of these objects as well as the ground truth models are shown in Figure 2.1a. A robotic arm, that could be accurately positioned on a one metre radius hemisphere, was used to capture images of the objects. The captured hemisphere is shown in Figure 2.1b.

The captured images had a resolution of 640x480 pixels. The arm photographed two sets of images of each object using two different arm configurations because the arm would cast shadows over the object in certain positions. The images without any shadows were selected for the final dataset. Each object was used to make three data sets with varying numbers of images. These are listed in decreasing order of size: a hemisphere containing more than 300 images, a ring containing between 40 and 50 images, and a sparse ring (referred to as SparseR. in Table 2.1) containing 24 images.

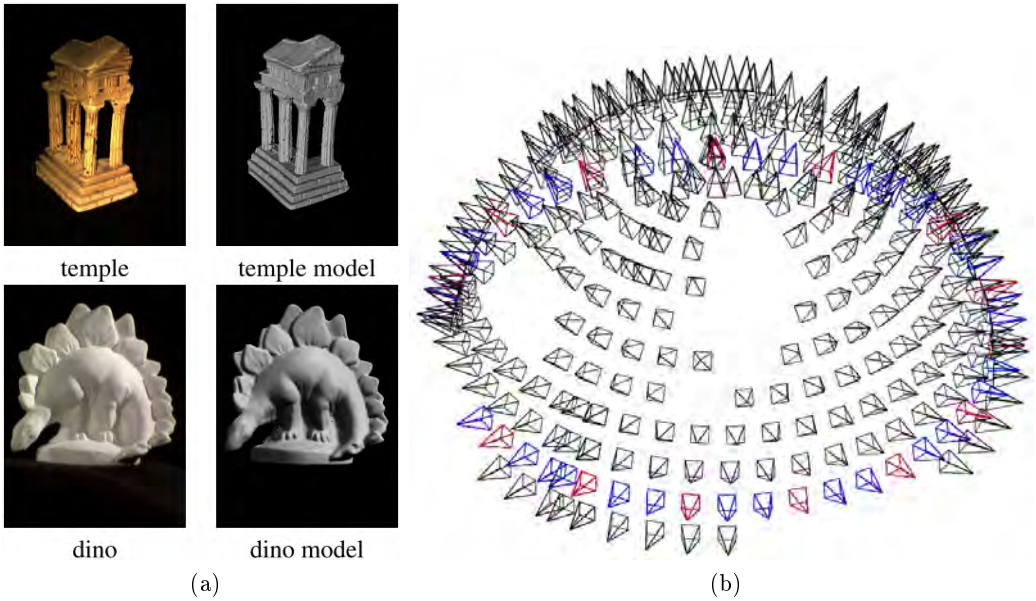

Figure 2.1: (2.1a) The objects that were used to create the image data and Ground Truth Models [3]. (2.1b) Hemisphere that the robot arm rotated around. The pyramids represent the viewpoints that the images are taken from [3].

Using this method Seitz *et al.* produced a high quality, calibrated set of input images. They also made use of a laser scanner to create Ground Truth Models (GTMs) of both objects. These models have an accuracy of 0.05mm to 0.2 mm [3]. From this data it is possible for an algorithm to be tested and then quantitatively rated according to two criteria. To test an algorithm first one must reconstruct the 3D models from the input data that was supplied, then the reconstructed model can be compared to the GTMs to determine how "good" the reconstruction is. The remaining question is how can one quantitatively measure goodness, which is by no means a scientific term?

Seitz *et al.* determined two metrics, accuracy and completeness, that would rate the goodness of an algorithm. The accuracy is *the distance d such that X% of the points on [the MVS reconstruction] are within distance d of [the GTM]* [3]; this metric shows how closely the MVS reconstruction models the GTM. In their paper the authors use an accuracy threshold of $X = 90\%$, so if $d = 0.65$ it means that 90% of the MVS reconstruction is within 0.65mm of the GTM. The completeness is defined as *the fraction of points of [the GTM] that are within an allowable distance d of [the MVS reconstruction]* [3]. In other words the completeness is the amount of the GTM that is modeled by the MVS reconstruction, so a reconstructed model with many holes will have a low completeness

score. The Middlebury evaluation is only relevant for closed mesh models, and does not provide relevant results for point clouds.

Table 2.1 shows the results of a number of MVS algorithms. The table is an amalgamation of the data collected by Seitz *et al.* [3] and Furukawa and Ponce [1]. The table lists the authors of a particular algorithm and their results on the Middlebury datasets. The Middlebury datasets contain two objects: a temple and a dinosaur. The dinosaur has a fairly uniform texture and the full dataset contains 363 images. The temple also has a fairly uniform texture and gaps between the pillars creating shadows and obscurations. The full temple dataset contains 317 images. These two datasets are each split into three sets: hemisphere, ring and sparse ring. The number of images in each dataset is shown in Table 2.1.

The PMVS algorithm by Furukawa and Ponce [1] has the highest accuracy and best completeness score for each set of the dinosaur dataset. Their algorithm also performs consistently well on the temple dataset but is outperformed on the hemisphere set by Campbell *et al.* [6]. More up-to-date results can be found on the Middlebury web page [14].

	Temple						Dino					
	Full (317)		Ring (47)		SparseR. (16)		Full (363)		Ring (48)		SparseR. (16)	
Bradley[15]			0.57	98.1%	0.48	93.7%			0.39	97.6%	0.38	94.7%
Campbell[6]	0.41	99.9%	0.48	99.4%	0.53	98.6%						
Furukawa[16]	0.65	98.7%	0.58	98.5%	0.82	94.3%	0.52	99.2%	0.42	98.8%	0.58	96.9%
Furukawa[1]	0.49	99.6%	0.47	99.6%	0.63	99.3%	0.33	99.8%	0.28	99.8%	0.37	99.2%
Goesele[17]	0.42	98.0%	0.61	86.2%	0.87	56.6%	0.56	80.0%	0.46	57.8%	0.56	26.0%
Hernandez[13]	0.36	99.7%	0.52	99.5%	0.75	95.3%	0.49	99.6%	0.45	97.9%	0.60	98.5%
Kolmogorov[18]			1.86	90.4%					2.80	85.7%		
Pons[19]			0.60	99.5%	0.90	95.4%			0.55	99.0%	0.71	97.7%
Vogiatzis[11]	1.07	90.7%	0.76	96.2%	2.77	79.4%	0.42	99.0%	0.49	96.7%	1.18	90.8%
Vogiatzis[12]	0.50	98.4%	0.64	99.2%	0.69	96.9%						
Zach[7]	0.51	98.8%	0.56	99.0%			0.55	98.7%	0.51	99.1%		

Table 2.1: Results from the Middlebury evaluation. The first value is the accuracy of the reconstruction (i.aa). The second value is the completeness (cc%), which is the percentage of points on the GTM that are within 1.25mm of the MVS reconstruction. The numbers in the parentheses at the top of the table are the number of images used in the reconstruction. Recreated from [1, 3]

2.2 Mathematical Background

2.2.1 Notation

Throughout this thesis there a number of different types of values are displayed, this section serves to describe the notations that are used.

- All matrices are shown as a capital letter, for example F is the fundamental matrix that is described later in this section.
- Scalar values are shown as lowercase letters, x or y.
- Vectors are shown as bold letters with an arrow over them, $\overrightarrow{\mathbf{x}} = (x, y)$.
- Points are shown as bold letters with a bar over them, $\overline{\mathbf{x}} = (x, y)$.
- Points in an image are referred to using the references (h, v), representing horizontal and vertical position.
- All vectors are considered to be column vectors unless otherwise stated. Therefore $\overrightarrow{\mathbf{x}^T}$ is a row vector.

2.2.2 Homogenous Coordinates

Homogenous coordinates represent the position of a point in 3D space using four parameters instead of three. They are often labeled as (x, y, z, w). When the w parameter is set to one, homogenous coordinates behave identically to normal 3D coordinates. Homogenous coordinates allow any point, including points at infinity, to be represented. They also simplify the mathematical calculations that are used in projective geometry.

When defining the position of a point on a two-dimensional plane a pair of coordinates (x, y) are used. If $\mathbb{R}^2$ is thought of as a vector-space then the point (x, y) can be thought of as a vector starting at the origin of $\mathbb{R}^2$ and ending at (x, y). Homogenous coordinates are used to represent points in two-dimensional space using a three-vector.

The equation of a line in two-dimensional space is $ax + by + c = 0$. This line can be represented as a vector $(a, b, c)^T$. However $(a, b, c)^T$ does not uniquely define a specific line because $ax + by + c = 0$ is the same line as $(ka)\,x + (kb)\,y + kc = 0$, for every non-zero value of k. This means that $(a, b, c)^T$ and $k\,(a, b, c)^T$ represent the same line for any non-zero value of k. This representation of lines and coordinates is known as the homogenous representation [20].

The point $\overline{\mathbf{x}} = (x, y)^T$ is on the line $\overrightarrow{\mathbf{L}} = (a, b, c)^T$ only if $ax + by + c = 0$. This is equivalent to $(x, y, 1)\,(a, b, c)^T = (x, y, 1)\,\overrightarrow{\mathbf{L}} = 0$, so the point $(x, y)^T$ can be represented as $(x, y, 1)^T$ in homogenous coordinates. It is evident that $(kx, ky, k)\,\overrightarrow{\mathbf{L}} = 0$ for any non-zero constant k. Therefore $(x, y, 1)^T \equiv (kx, ky, k)^T \equiv (x, y)^T$ [20].

2.2.3 Intrinsic and Extrinsic Parameters

Ideal pinhole cameras [20] can be characterised according to two matrices that contain their intrinsic and extrinsic parameters. The intrinsic parameters of a camera represent the internal properties of the camera such as the focal length, size of pixels in the image plane and position of the camera centre. The extrinsic parameters represent the position and orientation of the camera with reference to a set of world coordinates. The intrinsic parameters of a camera have the following form:

$$M_i = \begin{bmatrix} \frac{f}{h_x} & s & o_x \\ 0 & \frac{f}{h_y} & o_y \\ 0 & 0 & 1 \end{bmatrix} \tag{2.1}$$

The camera has a focal length of f, which is the distance from the aperture to the focal plane. The origin of the image is given by the point pair (o_x, o_y) and the physical dimensions of the pixels are given by (h_x, h_y). Using these data it is possible to transform points from the camera's coordinate system (x_c, y_c) into the image's coordinate system (h_{img}, v_{img}) and vice versa using the following equations.

$$\begin{aligned} x_c &= (h_{img} - o_x)\, h_x \\ y_c &= (v_{img} - o_y)\, h_y \end{aligned} \tag{2.2}$$

The aspect ratio of the image is given by aspect ratio $= \frac{h_y}{h_x}$, and s is called the skew parameter which can generally be set to zero and thus ignored.

The extrinsic parameters of a camera provide the means for moving the origin of the camera $\overline{\mathbf{O_c}}$ to the origin of the world $\overline{\mathbf{O_w}}$ using a transformation vector $\overrightarrow{\mathbf{T}}$, and then rotating the axis system of the camera such that it is aligned with the axes of the world using rotation matrix R. The rotation matrix and translation vector have the following form:

$$R = \begin{bmatrix} R^{1T} \\ R^{2T} \\ R^{3T} \end{bmatrix} = \begin{bmatrix} R_{11} & R_{12} & R_{13} \\ R_{21} & R_{22} & R_{23} \\ R_{31} & R_{32} & R_{33} \end{bmatrix} \tag{2.3}$$

$$\overrightarrow{\mathbf{T}} = \overline{\mathbf{O_w}} - \overline{\mathbf{O_c}} = \begin{bmatrix} T_1 \\ T_2 \\ T_3 \end{bmatrix} \tag{2.4}$$

The extrinsic parameter matrix is formed according to:

$$M_e = \left[R | \overrightarrow{\mathbf{T}} \right] = \begin{bmatrix} R_{11} & R_{12} & R_{13} & -R_1 T \\ R_{21} & R_{22} & R_{23} & -R_2 T \\ R_{31} & R_{32} & R_{33} & -R_3 T \end{bmatrix} \tag{2.5}$$

Finally the intrinsic and extrinsic parameters can be multiplied to form the camera matrix $P = M_i M_e$ which transforms 3D word coordinates into pixel coordinates in the image.

2.2.4 Epipolar Geometry

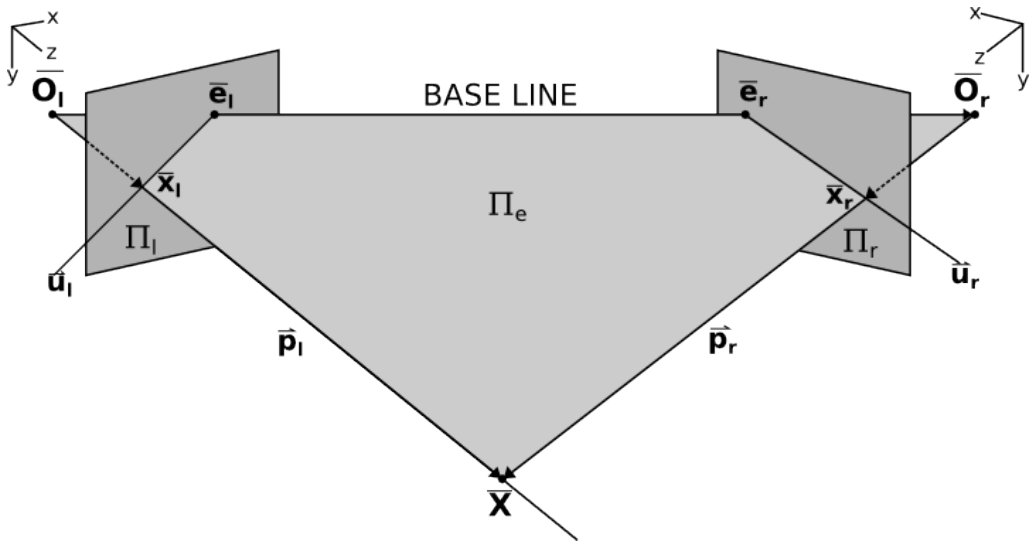

Figure 2.2: The main components of epipolar geometry. Adapted from [21].

Epipolar geometry can be used to find important geometric details about objects in a camera's field of vision. Figure 2.2 shows two pinhole cameras with optical centres $\overline{\mathbf{O_l}}$ and $\overline{\mathbf{O_r}}$. These cameras have image planes Π_l and Π_r respectively. The point of intersection of the line originating from an optical centre $\overline{\mathbf{O_i}}$ and passing through the image plane Π_i such that it is orthogonal to Π_i is called the principal point. The distance between the optical centre and the principal point is called the focal length and the line between them defines the optical axis of the camera. The line that joins $\overline{\mathbf{O_l}}$ and $\overline{\mathbf{O_r}}$ is called the base line. The points where the base line intersects the image planes are the epipoles, denoted as $\overline{\mathbf{e_l}}$ and $\overline{\mathbf{e_r}}$. These represent the projection of the camera centres into their respective image planes.

Given a 3D point $\overline{\mathbf{X}}$ that is visible in both Π_l and Π_r it is possible to define the epipolar plane, Π_e, that is the plane with vertices $\overline{\mathbf{X}}$, $\overline{\mathbf{O_l}}$ and $\overline{\mathbf{O_r}}$. The epipolar plane intersects the image planes and these intersections are called the epipolar lines, denoted as $\overrightarrow{\mathbf{u_l}}$ and $\overrightarrow{\mathbf{u_r}}$. An epipolar line $\overrightarrow{\mathbf{u_r}}$ describes the projection of the point $\overline{\mathbf{x_l}}$ onto the image plane Π_r.

Each camera has its own coordinate system such that the z-axis is co-linear with the optical axis. The points $\overline{\mathbf{x_l}} = [x_l, y_l, z_l]^T$ and $\overline{\mathbf{x_r}} = [x_r, y_r, z_r]^T$ represent the images of the 3D point $\overline{\mathbf{X}}$ in the image planes Π_l and Π_r respectively.

2.2.5 The Essential and Fundamental Matrix

By using a translation vector $\overrightarrow{\mathbf{T}} = \overline{\mathbf{O_r}} - \overline{\mathbf{O_l}}$, that translates the position of one camera centre to the other, and a rotation matrix R that aligns their systems of axes, it is possible to

change from one coordinate system to another. Given two vectors $\overrightarrow{\mathbf{x_l}}$ and $\overrightarrow{\mathbf{x_r}}$ both of which point towards $\overrightarrow{\mathbf{X}}$ from their respective camera centres, $\overrightarrow{\mathbf{O_r}}$ and $\overrightarrow{\mathbf{O_l}}$, then $\overrightarrow{\mathbf{x_r}} = R\left(\overrightarrow{\mathbf{x_l}} - \overrightarrow{\mathbf{T}}\right)$.

The epipolar plane Π_e in the coordinate system of the left camera is spanned by $\overrightarrow{\mathbf{T}}$ and $\overrightarrow{\mathbf{p_l}}$, thus $\overrightarrow{\mathbf{p_l}} - \overrightarrow{\mathbf{T}}$ also belongs in that plane, therefore $\left(\overrightarrow{\mathbf{x_l}} - \overrightarrow{\mathbf{T}}\right)\left(\overrightarrow{\mathbf{T}} \times \overrightarrow{\mathbf{x_l}}\right) = 0$.

$$\overrightarrow{\mathbf{T}} \times \overrightarrow{\mathbf{x_l}} = \begin{bmatrix} T_1 & T_2 & T_3 \\ p_{l1} & p_{l2} & p_{l3} \\ i & j & k \end{bmatrix} = \begin{bmatrix} 0 & -T_3 & T_2 \\ T_3 & 0 & -T_1 \\ -T_2 & T_1 & 0 \end{bmatrix} \begin{bmatrix} p_{l1} \\ p_{l2} \\ p_{l3} \end{bmatrix} = A \cdot \overrightarrow{\mathbf{x_l}} \tag{2.6}$$

Because R is orthogonal

$$\overrightarrow{\mathbf{x_r^T}} RA\overrightarrow{\mathbf{x_l}} = 0 \tag{2.7}$$
$$\overrightarrow{\mathbf{x_r^T}} E\overrightarrow{\mathbf{x_l}} = 0 \tag{2.8}$$

$E = RA$ is called the essential matrix, which is of rank two and stores encoded information about the extrinsic parameters of both the left and right cameras. The essential matrix transforms image plane coordinates from one camera to image plane coordinates of the other camera. With $\overrightarrow{\mathbf{x_r^T}} E\overrightarrow{\mathbf{x_l}} = 0$ where $\overrightarrow{\mathbf{x_l}}$ and $\overrightarrow{\mathbf{x_r}}$ are image points on the planes Π_l and Π_r respectively. Since these points lie on the epipolar lines of the other image plane the following equations hold:

$$\overrightarrow{\mathbf{u_r}} = E\overrightarrow{\mathbf{x_l}} \tag{2.9}$$
$$\overrightarrow{\mathbf{u_l}} = E^T\overrightarrow{\mathbf{x_r}} \tag{2.10}$$

The pixel coordinates $\overrightarrow{\mathbf{x_{pi}}}$ of a point $\overrightarrow{\mathbf{x_i}}$ are given by $\overrightarrow{\mathbf{x_{pi}}} = M_{ij}\overrightarrow{\mathbf{x_i}}$ where M_{ij} is the matrix of intrinsic parameters for camera j as defined in Equation 2.1. Using this relationship the following equations are found.

$$\left(M_{ir}^{-1}\overrightarrow{\mathbf{x_{pr}}}\right)^T EM_{il}^{-1}\overrightarrow{\mathbf{x_{pl}}} = 0 \tag{2.11}$$
$$\overrightarrow{\mathbf{x_{pr}^T}} M_{ir}^{-T} EM_{il}^{-1}\overrightarrow{\mathbf{x_{pl}}} = 0 \tag{2.12}$$
$$\overrightarrow{\mathbf{x_{pr}^T}} F\overrightarrow{\mathbf{x_{pl}}} = 0 \tag{2.13}$$

Where $F = M_{ir}^{-T} EM_{il}^{-1}$ is called the fundamental matrix, which is a matrix of rank two and stores encoded information of the intrinsic and extrinsic camera parameters. It describes the epipolar geometry in terms of pixel coordinates as opposed to the essential matrix which uses the image plane coordinates.

2.2.6 Triangulation

There are a number of methods that one can use to resolve a 3D point $\overrightarrow{\mathbf{P}}$ from two 2D image points $\overrightarrow{\mathbf{p_l}}$ and $\overrightarrow{\mathbf{p_r}}$ and their respective camera information. The naive triangulation

method is to project lines from the optical centres of the cameras through their respective points in the image plane out into the focal plane. In Figure 2.2 these lines are given by $\overrightarrow{\mathbf{P_l}}$ and $\overrightarrow{\mathbf{P_r}}$. This 3D point is the intersection of these two lines. In real multi-view stereo problems these two lines often do not intersect exactly and as such a better method is required. Three such methods are considered here.

2.2.6.1 Midpoint Triangulation

The midpoint triangulation method creates a line between the two rays $\overrightarrow{\mathbf{P_l}}$ and $\overrightarrow{\mathbf{P_r}}$ that is orthogonal to both. A numerical optimiser is then used to minimise its length. Finally the position of the 3D point $\overline{\mathbf{P}}$ is the midpoint of this joining line. This method is not projective invariant, and although simple to compute, often performs poorly [22].

Let $\overline{\mathbf{P}} = (M_i \,|{-M_i}\mathbf{c}_i)$ and the centre of a camera be given by $\overline{\mathbf{O_i}} = \begin{pmatrix} \mathbf{c}_i \\ 1 \end{pmatrix}$ using homogenous coordinates. The point at infinity that is seen on the image plane as point $\overline{\mathbf{x_i}}$ is given by $\begin{pmatrix} M^{-1}\overline{\mathbf{x_i}} \\ 0 \end{pmatrix}$. Therefore the equation for any point on a ray that projects onto $\overline{\mathbf{x_i}}$ is $\begin{pmatrix} \mathbf{c}_i + \alpha M_i^{-1}\overline{\mathbf{x_i}} \\ 1 \end{pmatrix}$. Given any two images their respective rays may cross in space, according to $\alpha_l M_l^{-1}\overline{\mathbf{x_l}} - \alpha_r M_r^{-1}\overline{\mathbf{x_r}} = \overrightarrow{\mathbf{c_r}} - \overrightarrow{\mathbf{c_l}}$. A numerical minimisation method can be used to change the values of α_l and α_r such that the squared distance between the two rays and the midpoint given by $\left(\overrightarrow{\mathbf{c_l}} + \alpha_l M_l^{-1}\overline{\mathbf{x_l}} + \overrightarrow{\mathbf{c_r}} + \alpha_r M_r^{-1}\overline{\mathbf{x_r}}\right)/2$ is minimised.

2.2.6.2 Linear Triangulation

One of the most common methods of triangulation is the method of linear triangulation [5, 20, 22]. This method makes use of the camera matrix and some geometric properties to accurately triangulate a 3D point. If $\overline{\mathbf{x_i}} = P\overline{\mathbf{X}}$, in homogenous coordinates $\overline{\mathbf{x_i}} = w\,(h, v, 1)^T$, where h and v are the coordinates of the points, in pixels, and w is an unknown scaling factor. Using P^{iT} to denote the i^{th} row of the camera matrix P the following equations can be found:

$$wh = P^{1T}\overline{\mathbf{x_i}} \tag{2.14}$$

$$wv = P^{2T}\overline{\mathbf{x_i}} \tag{2.15}$$

$$w = P^{3T}\overline{\mathbf{x_i}} \tag{2.16}$$

Then eliminate w from the first and second equations:

$$hP^{3T}\overline{\mathbf{x_i}} = P^{1T}\overline{\mathbf{x_i}} \tag{2.17}$$

$$vP^{3T}\overline{\mathbf{x_i}} = P^{2T}\overline{\mathbf{x_i}} \tag{2.18}$$

When looking at the 3D point from two views a total of four linear equations are found

that can then be written in the form of $A\overline{\mathbf{x}} = \overline{0}$. Equations of the form $A\overline{\mathbf{x}} = \overline{0}$ can be solved using a method such as singular value decomposition (SVD) or another method for solving equations of the form $A\overline{\mathbf{x}} = \overline{\mathbf{b}}$ which is described in Section 2.2.7.2.

2.2.6.3 Optimal Triangulation Method

The optimal triangulation method can be used to find a 3D point in space that is the exact intersection of two rays [20, 22]. The optimal triangulation method was initially implemented as an alternative to linear triangulation in the dissertation. However its long processing time made it unsuitable for this particular implementation. Briefly the method uses the known fundamental matrix for a pair of cameras, as well as corresponding feature points that have been located in each of the input images. The method first changes the position of the feature points in the image. This small change ensures that the points in the left and right images project out to the same 3D point.

2.2.7 Numerical Methods

2.2.7.1 Limited Memory Broyden-Fletcher-Goldfarb-Shanno Optimisation Algorithm

The Limited Memory Broyden-Fletcher-Goldfarb-Shanno [23] (L-BFGS) algorithm is an implementation of the standard BFGS algorithm, with an extra variable m introduced. This variable is the maximum number of corrections that can be stored, so for the first m iterations of the algorithm it is identical to the normal BFGS method. Which is itself an approximation to Newton's method that uses an approximation of the Hessian matrix computed from first derivatives. Thereafter the oldest correction is pushed out and replaced with the second oldest correction.

L-BFGS makes use of matrix-vector and matrix-matrix multiplication. For this reason it was considered for a GPU implementation. Fei *et al.* [24] created a GPU implementation of this algorithm which they showed could outperform the CPU implementation when computing Voronoi tessellation with one thousand vertices and over. The authors made use of NVIDIA's CUDA to implement the matrix operations as well as the CUBLAS library for solving of equations.

The L-BFGS algorithm is outlined in Algorithm 2.1 below:

Algorithm 2.1 L-BFGS Algorithm [23].

Inputs:

Choose x_0 as a starting point.
Choose m, the maximum number of corrections to store.
Choose two constants $0 < \beta' < 0.5$ and $\beta' < \beta < 1$.
Choose a symmetric, positive-definite matrix H_0.

Outputs:

Minimal solution to the smooth nonlinear function f with known gradient g.

1) Set $k = 0$ and $q = g_0 \equiv \nabla f(x_0)$.
2)

$$\begin{aligned} d_k &= -H_k g_k \\ x_{k+1} &= x_k + \alpha_k d_k \\ where: & \\ \alpha_k &= p_k s_k^T q \\ g_k &\equiv \nabla f(x_k) \\ p_k &= \frac{1}{y_k^T s_k} \\ s_k &= x_{k+1} - x_k \\ q &= q - \alpha_k y_k \\ y_k &= g_{k+1} - g_k \end{aligned}$$

where α_k satisfies the Wolfe conditions:

$$\begin{aligned} f(x_k + \alpha_k d_k) &\leq f(x_k) + \beta' \alpha_k g_k^T d_k \\ g(x_k + \alpha_k d_k) &\geq \beta g_k^T d_k \end{aligned}$$

3) Let $\hat{m} = \min\{k, m-1\}$. Then

$$\begin{aligned} H_{k+1} &= \left(V_k^T \cdots V_{k-\hat{m}}^T\right) H_0 \left(V_{k-\hat{m}} \cdots V_k\right) \\ &+ p_{k-\hat{m}} \left(V_k^T \cdots V_{k-\hat{m}+1}^T\right) s_{k-\hat{m}} s_{k-\hat{m}}^T \left(V_{k-\hat{m}+1} \cdots V_k\right) \\ &+ p_{k-\hat{m}+1} \left(V_k^T \cdots V_{k-\hat{m}+2}^T\right) s_{k-\hat{m}} s_{k-\hat{m}}^T \left(V_{k-\hat{m}+2} \cdots V_k\right) \\ &\vdots \\ &+ p_k s_k s_k^T \\ where: & \\ V_k &= I - p_k y_k s_k^T \end{aligned}$$

4) Set $k = k + 1$ and go to 2.

2.2.7.2 Solving Systems of Linear Equations

When attempting to triangulate a point the triangulation algorithm requires the solution of an equation in the form $A\overline{\mathbf{x}} = \overline{\mathbf{b}}$ where $\overline{\mathbf{x}}$ is the homogenous 3D coordinate of the triangulated point. Finding a least squares solution to equations of this form is possible using the equation:

$$\overline{\mathbf{x}} = \left(A^T A\right)^{-1} A^T \overline{\mathbf{b}} \tag{2.19}$$

The reason this equation is useful is because the matrix $\left(A^T A\right)$ may be invertible. This gives a least squares error solution to the problem [25] and the least squares error can be found by calculating the distance between $A\overline{\mathbf{x}}$ and $\overline{\mathbf{b}}$.

2.3 Parallel Processing

One of the major problems that this thesis aims to tackle is the relevance of parallel processing to a particular 3D reconstruction algorithm and possibly reconstruction algorithms in general. Two types of parallel processing are discussed and implemented. The first type uses multiple CPU threads to run specific parts of the program in parallel. The second type makes use of the relatively new method of general purpose processing on GPUs (GPGPU), specifically NVIDIA's CUDA. These two methods of parallel processing are very different from one another and require different programming approaches. This can be attributed to the fact that the CPU and GPU have very disparate architectures as they were designed for different tasks. The CPU was designed as a general purpose processor that can run an operating system as well as perform complex floating point problems. The GPU on the other hand has a design much more focused on running thousands of small threads in parallel. It is essentially a very powerful mathematics processor.

2.3.1 Parallelism on the CPU

The CPU is a general purpose processor that is suited to run multiple instructions in a pipeline allowing for near simultaneous operation of instructions. The pThreads library [26] is a C and C++ open source library that allows one to create independent CPU threads. The library provides important functionality for CPU based parallel processing.

When processing data in parallel one needs to be mindful of how the final result of a thread may effect the results of other threads that are currently running. If there is not a sufficient amount of data parallelism then MUTual EXclusions (MUTEXs) can be used. A MUTEX will temporarily lock a shared resource when a thread writes to it. This way other threads cannot access the resource until the thread with the MUTEX releases it. This allows data to be written to shared memory without multiple threads attempting to gain simultaneous access.

2.3.2 The Graphics Card as a Massively Parallel Processor

2.3.2.1 Introduction

Massively parallel processors are designed to take advantage of the parallelism that is present in some problems in an attempt to improve the performance of a program by several orders of magnitude. These sorts of programs used to be run on very large, and very expensive, computing clusters that were only available to researchers with large grants or corporations with the necessary capital to afford such a cluster. The high running cost offset the performance gains for many years.

This started to change as the video game industry required computers that could generate 3D computer graphics in real time. The fast evolving video game industry drove hardware manufacturers to create affordable graphics cards that could be attached to any personal computer [4]. These graphics processing units (GPUs) were many-core processors. Current generation GPUs that can be bought by any user have hundreds of compute cores, for example the GTX550Ti which has 192 Compute cores. The top of the range GPUs have thousands of compute cores, for example the NVIDIA GTX Titan which has 2688 cores.

Researchers and programmers soon realised that GPUs could be used to run highly parallel programs without the need of an expensive compute cluster; this became the field known as GPGPU. GPGPU is so called because it makes use of the GPU, which is essentially a numeric computing engine which specialises in 3D graphic computations, to solve many general problems in wide ranging fields from mathematics and physics to computer vision and medical imaging. There were still many limitations with the GPGPU framework and it was a non-trivial task to create a GPGPU program because all mathematical operations had to be cast as graphics operations that the fixed-function processors on the GPU could understand.

More recent developments in the field, such as NVIDIA's compute unified device architecture (CUDA) and OpenCL, have made programming on the GPU much simpler. CUDA extends the C++ language and allows programs to run on the GPU and CPU simultaneously. Furthermore because of a number of recent changes to the hardware of GPUs their processors no longer require problems to be cast as graphical operations and they can be programmed almost as one would program a CPU.

2.3.2.2 Graphics Processor Architecture

The graphics processing unit began its life cycle as a fixed function processing pipeline however this pipeline could not be programmed [4]. Through the use of Application Programming Interfaces (APIs) it was possible for game developers and designers to use the fixed function GPU to create 3D worlds in a video game. The fixed function pipeline of a GPU is shown in Figure 2.3.

In the fixed function pipeline the CPU, known as the host, oversees all operations passed to the GPU, or the device. The host interface receives instructions from the host

and is able to interpret and execute these instructions on the device. The vertex control stage converts triangle data from the host interface into a form that is understood by the device. This is then placed in the vertex cache which can be accessed by the Vertex Shading, Transform and Lighting (VS/T & L) stage of the pipeline. This stage transforms the vertices and assigns values to them, such as colours, normals, and texture coordinates. The triangle setup stage creates equations for the edges of each triangle which are then used to interpolate colours and other vertex data. The raster stage then determines which pixels are within each triangle, then for each of these pixels it interpolates vertex values that are necessary for shading the pixel. The shader stage finally applies colour to each of the vertices. The Raster OPeration (ROP) stage is used to determine which objects are adjacent or overlapping for transparency type effects and for getting rid of occluded pixels. Finally the Frame Buffer Interface (FBI) writes the final result to the frame buffer memory [4].

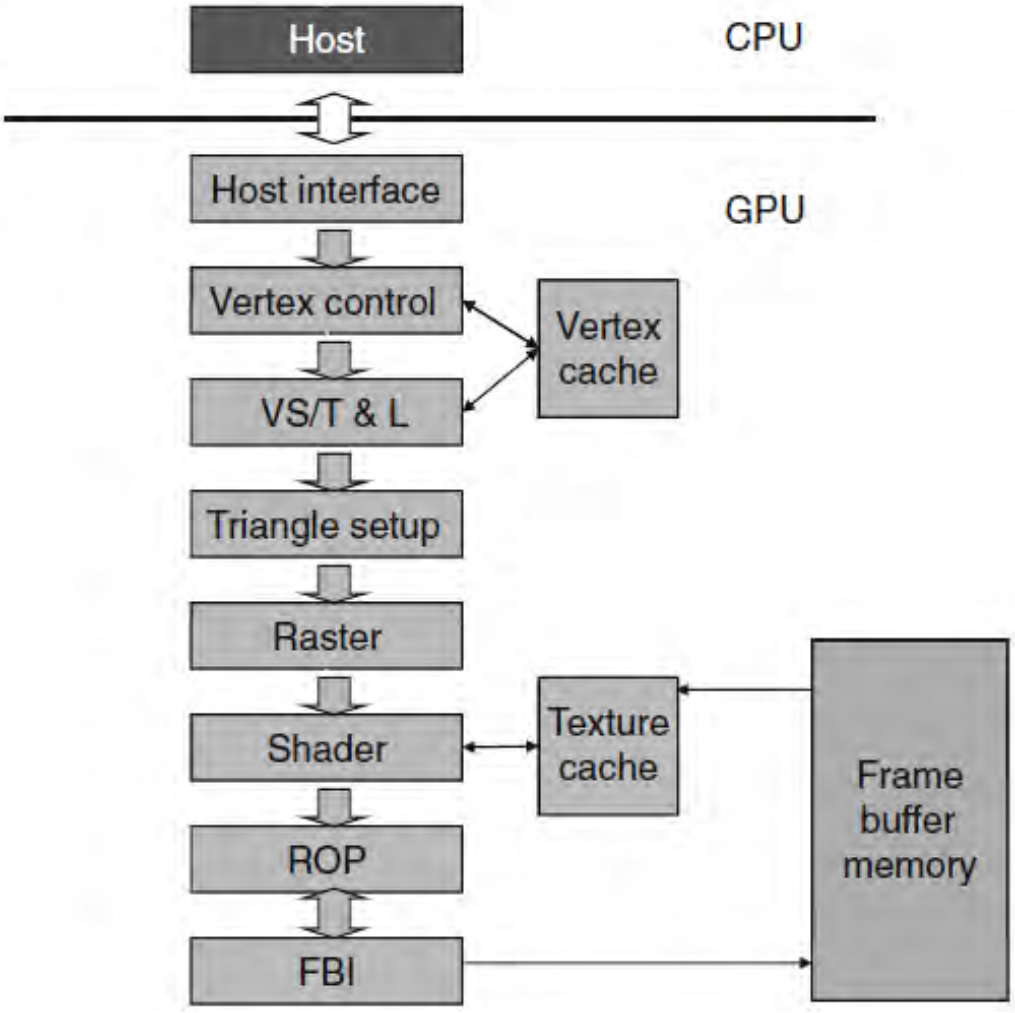

Figure 2.3: The fixed function pipeline of an old NVIDIA GeForce GPU [4].

2.3.2.3 General Purpose Graphics Processing Units

The topic of General Purpose computing on GPUs relates to using the GPU as a general purpose processor. Originally GPUs were used simply to calculate, process, and draw

geometry and graphics. However modern GPUs have begun to make their internal pipeline available to the programmer, making them very useful for computationally intensive tasks. GPUs tend to have many cores suited to small, but highly parallel, operations. For this reason GPUs can be used for tasks that are data parallel in nature (see Section 2.3.2.4) such as matrix multiplication.

2.3.2.4 Parallel Programs and Data Parallelism

The GPU is a very powerful tool when it comes to improving the execution time of a program. It does however have a number of limitations and it is by no means a panacea for every slow program. Therefore it is important to use the GPU only where it is relevant. It performs best on problems that display a large amount of data parallelism.

Data parallelism is when calculations are largely independent of one another, in other words the result of a particular thread A will have no impact on the result of thread B, regardless of order of execution. The GPU does not run instructions sequentially in the same way that a CPU does. For example if a particular problem is running in 5000 threads on the GPU there is a chance that thread 3000 will execute before thread 20. If the result of thread 3000 is based on the result of thread 20 the GPU will return an incorrect answer. If this were the case one would say that this particular problem does not display data parallelism and is probably not suitable for an implementation on the GPU.

The reason that data parallelism is such an important aspect of GPU programming is based on the primary reason GPUs were made: to process graphics and display them on a computer monitor. If a video game is running at 60 frames per second (FPS) the GPU has to process an entire frame in $\frac{1}{60}^{th}$ of a second, and a single frame may be composed of millions of triangles and several million pixels. This meant that there was a great opportunity for the GPU to make use of hardware parallelism to exploit the nature of the parallel data.

A very good example of a problem that does display data parallelism is matrix multiplication. Two matrices, M and N, of size $n \times n$ are multiplied together to obtain result P. Figure 2.4 shows what a sample calculation would look like. A row in M is multiplied by a column in N to find the value of a single element in P. This calculation shows that the value of an element of P has no impact on the value of any other element of P. The only values that are required are from the input matrices M and N. Therefore the problem of matrix multiplication displays data parallelism and would be suitable for implementation on a GPU.

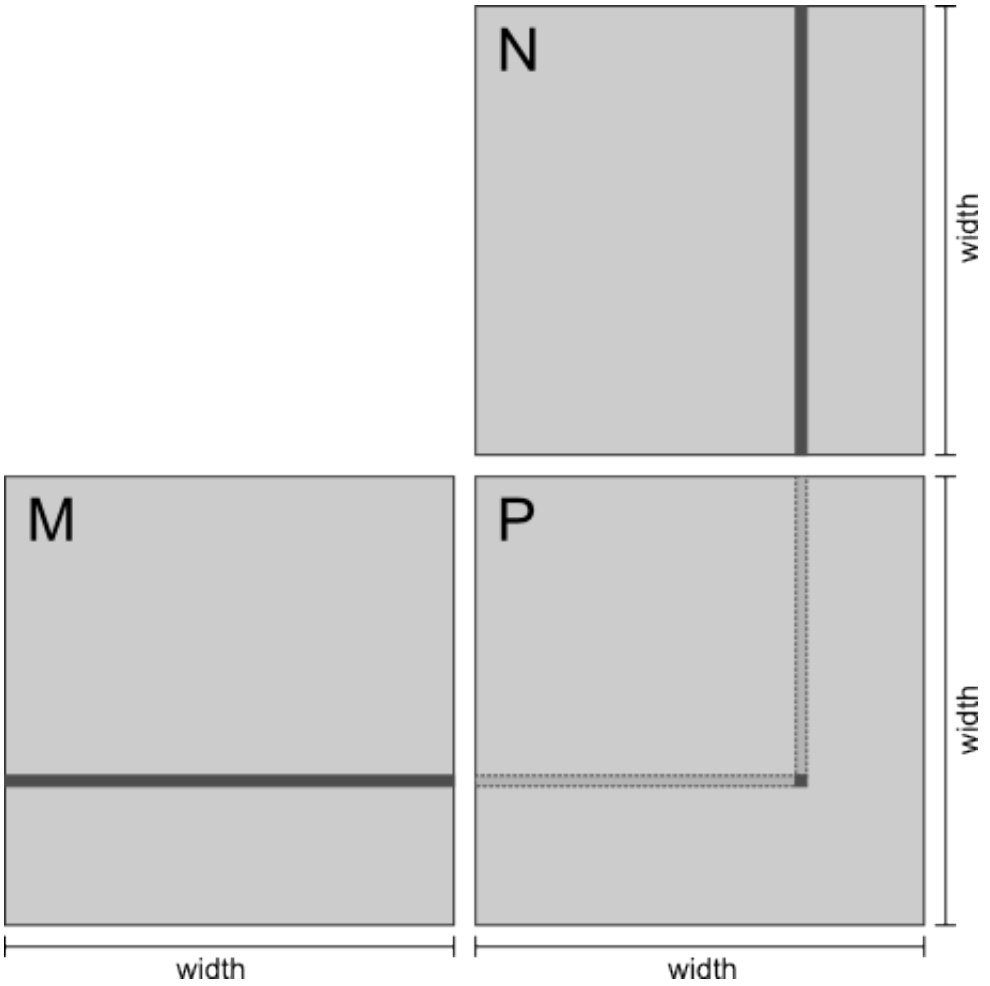

Figure 2.4: Matrix multiplication. Adapted from [4].

2.3.3 Compute Unified Device Architecture

The compute unified device architecture (CUDA) was developed by NVIDIA. The first graphics card that was based on this architecture was the GeForce 8800. This was a large step in the evolution of graphics processing, moving away from discrete vertex and pixel shaders towards a unified shader architecture. This refined architecture allowed NVIDIA to create a set of additions to the standard C++ library that allowed programs to have portions of their code executed on a NVIDIA CUDA capable card.

2.3.3.1 Programming Structure

In many ways programming a CUDA device is very similar to programming a normal CPU. However there are a few important differences to be considered. The first major difference is that a CUDA GPU cannot allocate its own memory. It relies on the CPU, referred to as the host, to allocate memory on the GPU. The host is used to supervise many aspects of GPU computing using CUDA. It will allocate and free memory for the device; it will call the code that runs on the GPU, referred to as a kernel, and define the number of blocks and threads on which that kernel will run. The next major difference between parallelism on CUDA devices and that of the CPU is that CUDA kernels are run in what is called a

grid of blocks. Each block contains a number of threads that are extremely light-weight and it is assumed that it takes negligible time to initialise a thread. As has been mentioned CUDA threads are designed to be lightweight and all threads will run the exact same piece of code. Therefore a method is required to differentiate between threads. This is possible because each block, and each thread in a block, is given an ID. Using this ID it is possible for a thread A running the same code as thread B to process different data. These blocks are further divided into units called warps which are groups of threads with continuous ID values. The purpose of warps is to hide high latency operations such as memory accesses. This is achieved by detecting if a thread is waiting for some high latency operation to complete. If it is waiting the next warp is selected for execution while the previous warp waits for the high latency operation to finish. Another difference is that CUDA makes use of a number of different memories with different access levels, which is discussed in the next section.

2.3.3.2 CUDA Memories

CUDA capable GPUs have a number of different types of memories that a programmer can use. All of these types of memories are faster than CPU memory. However when compared with one another there can be dramatic speed differences. The four types of memory, listed in increasing order of speed, are: global memory, shared memory, texture memory, and local (or register) memory.

1. Global memory is the largest memory available to the GPU, it is usually Graphics Double Data Rate 5 (GDDR5) Synchronous Dynamic Random Access Memory (SDRAM). It is available to all blocks running on the GPU and can be accessed by all threads running within the blocks. This is the slowest memory available, due in part to its large size. There are some internal optimisations that have been made for this memory, the major optimisation is that when a thread loads a block of memory determine which views are considered for that warp. This means that any threads in the same warp as the thread that requested the memory block will not have to make a call to global memory but rather to local memory, which is much faster.

2. Shared memory is shared by all threads in a single block, but is not shared between blocks. Shared memory can be declared dynamically or statically by the host. It can be accessed much faster than the global memory but is limited in the fact that it is smaller and is only accessible by the threads in a block.

3. Texture memory is used by the GPU to store texture information. This is generally used for video gaming and video processing tasks where many pixels have to be processed. Fortunately this memory can be used in GPGPU as a faster form of global memory. Texture memory can be accessed by all blocks in a grid and thus all threads in a block. This is beneficial for some programs, however texture memory can be slower than shared memory.

4. Local or register memory can only be accessed by single threads and is the fastest

form of memory being composed of high speed registers. This memory is very useful when a thread needs to run some small calculations very fast.

2.3.4 An Example of Parallel Processing

An example of how parallel processing can be used is outlined below. This example is relevant because it was used in the main program described in this dissertation during the feature detection stage that is discussed in Section 4.4.1. Image convolution is used frequently in almost all aspects of image processing. It involves convolving an image with a discrete function, called a convolution kernel, to create an output image. There are a number of different types of convolution kernels and they perform many different functions. For example the Gaussian convolution kernel, which was used in the program, when convolved with an image creates a blurred version of the input image, thus removing its the high frequency components. This has uses in problems such as feature detection where high frequency noise may be erroneously detected as a feature.

The convolution operation takes an input image such as the left image of Figure 2.5 and blurs it so that it looks like the right image of Figure 2.5. The input image as well as the convolution kernel are loaded into the texture memory of the GPU. The CUDA kernel then uses a one dimensional Gaussian kernel to blur the image in the h-direction. It then takes the blurred image and blurs it in the v-direction. This has the cumulative effect of a 2D Gaussian kernel (Appendix A.1).

(a) Input image of a skull. (b) Blurred version of the input image. ($\sigma = 4.0$)

Figure 2.5: The input and output of a Gaussian blurring kernel.

A simple example of a 2D Gaussian kernel is shown in Figure 2.6. The right hand matrix represents the input image that was convolved with the Gaussian function. The Gaussian function is the smaller matrix in the centre. It averages the neighbouring pixels of the central pixel in the input image to change the value of the central pixel. The output image is shown in the matrix on the right with a new value for the central pixel.

Image convolution can make use of the highly parallel nature of the GPU because the value of an output pixel relies only on the values of the input pixels and the Gaussian

kernel.

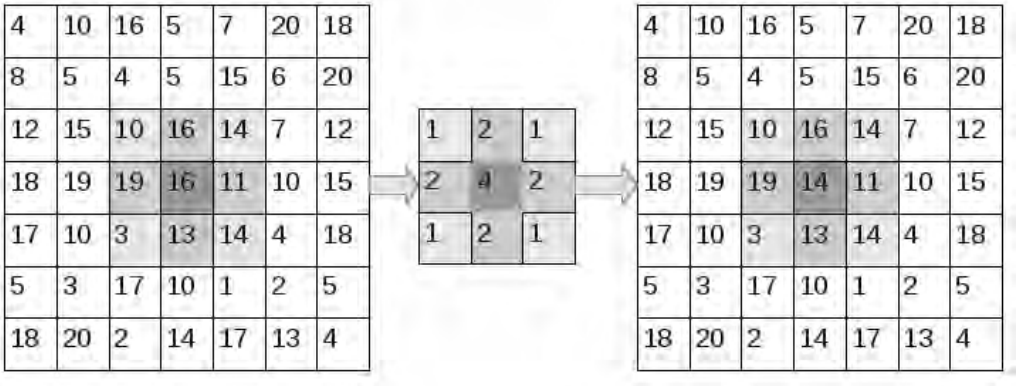

Figure 2.6: Example of a 2D Gaussian kernel blurring a segment of an image.

A CPU and GPU implementation of this convolution were written with the CPU algorithm taking 3.764 seconds to create the blurred image. The GPU took 0.367 seconds to create the same output which is approximately 10 times faster. These steps are shown in Algorithm 2.2

Algorithm 2.2 Gaussian smoothing on the GPU.

Input: Image I and Gaussian kernel G
Output: Smoothed image I'

$(h, v) \leftarrow$ Unique h and v ID of the current pixel in I
radius $\leftarrow$ Radius of the Gaussian kernel G
// First the h-direction derivative is found and saved
// Then the v-direction derivative of the h-derivative
// image is found. This is the resultant image. The
// GPU pseudocode for both is shown below for simplicity.
For each Gaussian value G_k in $G(k)$

{// For h direction
$n = h + k - radius$
$s = s + I(n, v) G_k$
}
{// For v direction
$n = v + k - radius$
$s = s + I(h, n) G_k$
}

end for
$I'(h, v) = s$

2.4 Selection of Modern MVS Algorithms

2.4.1 Accurate, Dense, and Robust Multi-View Stereopsis

"Accurate, Dense, and Robust Multi-View Stereopsis" by Furukawa and Ponce [1] presents a novel patch-based MVS reconstruction algorithm that outputs a dense collection of small rectangular patches that cover the surfaces that are visible in the input images. A final post-processing stage creates a closed mesh model.

Figure 2.7: Sample input image to be analysed by the PMVS algorithm [1].

The algorithm operates by detecting and matching features between input images. A sample image is shown in Figure 2.7. These matched features are used to create a sparse set of patches that form the starting point which is shown in Figure 2.8a. The expansion stage takes a set of patches as inputs and uses them to seed the creation of new patches nearby to create a more dense collection of patches. The expansion stage can produce many erroneous patches which need to be removed in a final filtering stage. This filtering stage applies three filters to the set of patches. The first filter removes patches whose projections are not visible in at least three input images (Figure 2.8b). The second filter enforces a strong form of visibility consistency by removing patches that are not visible in enough images (Figure 2.8c). The final filter enforces a weak form of regularisation by removing outlying patches that do not have enough patches that are their neighbours (Figure 2.8d). This filtering stage is very aggressive and removes a large number of points. The expansion and filtering steps are iterated three times to ensure dense coverage of the object.

(a) Initial patches (b) Filter 1 (c) Filter 2 (d) Filter 3

Figure 2.8: The initial patches found after feature detection and matching are shown in 2.8a, the expansion step is not shown. Next the initial patches are passed through the three filters: (2.8b) Weak Visibility; (2.8c) Strong Visibility and (2.8d) Weak Regularisation [1].

The PMVS algorithm produces very accurate results and is able to operate on object and scene data sets. Furthermore because the patches are independent of one another it is highly parallel. The final mesh model of the face data set is shown in Figure 2.9.

Figure 2.9: Final Mesh Model [1].

2.4.2 Multiple Hypotheses Depth-Maps for Multi-View Stereo

Campbell *et al.* propose a depth-map based MVS algorithm in the paper "Using Multiple Hypotheses to Improve Depth-Maps for Multi-View Stereo" [6] which overcomes two of the major issues common to these types of algorithms. The two problems that the authors have identified are:

- Spurious matches due to repeated texture.
- Matching failure due to occlusion, distortion and lack of texture.

The authors overcome these problems by making additions to the normal depth-map based MVS algorithm. The first change was to create good depth estimates for each pixel, and

then select the optimal depth of the pixel by enforcing consistent depth between neighbouring pixels. This helps to remove spurious results from repeated texture because pixels representing the same portion of an object in different images are expected to have similar depths to their neighbouring pixels. Whereas the pixels in repeated textures will have different depths to neighbouring pixels. The second addition was to allow the depth-map optimisation procedure to return an unknown state when it was unable to find an optimal depth. This unknown state allowed the authors to overcome the problem of matching failure since the unknown state adds no weight to the final depth of the pixel. Because of this the depth of the pixel can be determined from other images.

These additions have allowed the authors to create a very accurate depth-map based MVS algorithm, the performance of which is shown in Table 2.1. Their approach also gives very good results when there are few input images, which can cause many depth-map based algorithms to fail [3, 6]. Figure 2.10 shows the results of the algorithm by Campbell *et al.* on an input of only three images.

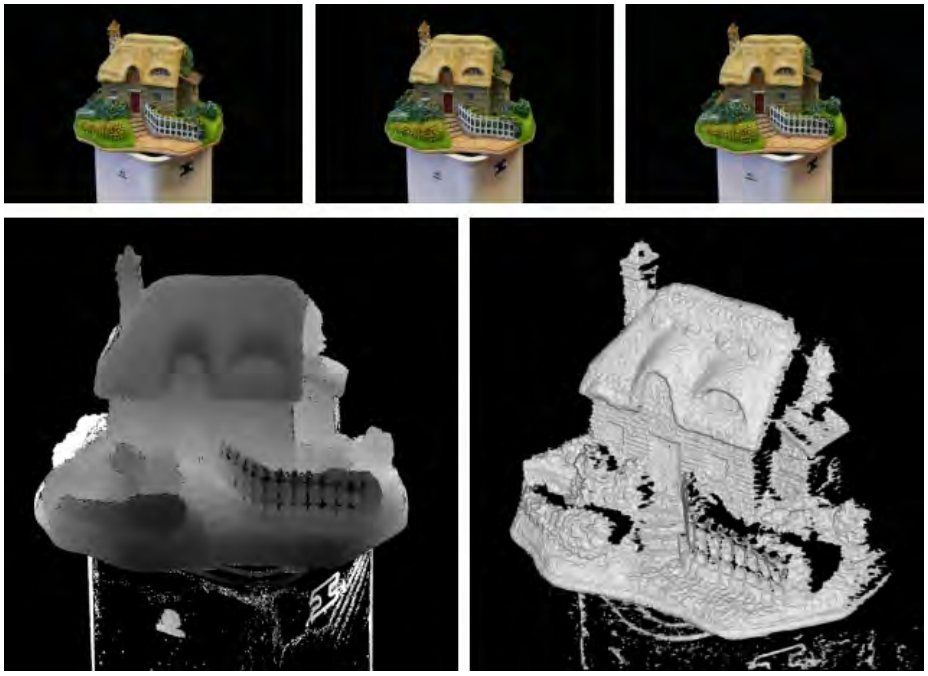

Figure 2.10: Results of the depth map MVS algorithm by Campbell *et al.* Only three input images are used, shown on the top; the recovered depth map is shown on the bottom left and the final rendering created from the depth map is shown on the bottom right [6].

Chapter 3

Patch-Based Multi-View Stereo

3.1 Introduction

The patch-based multi-view stereo reconstruction algorithm, described in the paper "Accurate, Dense, and Robust Multiview Stereopsis" [1] by Furukawa and Ponce shows a method for creating dense 3D models of objects, scenes, and crowded scenes. Their algorithm makes use of 3D patches that are used to densely model the surface of the object.

The technique makes use of a match, expand, and filter approach. The matching stage takes a sequence of calibrated images and creates a sparse set of 3D patches. The expansion and filtering stage are iterated three times, with the expansion stage using existing patches to create many more patches, some of them erroneous. The filtering stage makes use of three fairly strict filtering algorithms that removes patches that may not be visible or are outliers. This ensures that the reconstruction is mostly constructed of correct and accurate patches.

The output of the algorithm is a dense set of oriented rectangular patches that closely model the surface that they cover. This algorithm was chosen as the starting point for this dissertation since the patches are inherently disconnected, and can therefore be processed somewhat separately and in parallel.

3.2 Model Representation

3.2.1 Patch Model

Patch-based multi-view stereo reconstruction makes use of a dense set of small rectangular patches to accurately model a surface. These patches are oriented such that they are normal to the surface they lie on. Geometrically a patch, p, is defined by its centre $\overline{\mathbf{c}}(p)$, a three-dimensional position in space, and its unit normal vector $\overrightarrow{\mathbf{n}}(p)$ which originates from $\overline{\mathbf{c}}(p)$ and initially points towards the camera observing it (Figure 3.1). A patch is optimised such that the entire patch is normal to the surface it is modeling. Furthermore each patch stores its reference image $R(p)$, in which p is visible. The orientation of the patch is set such that one of its edges is parallel to the x-axis of the reference camera and the initial size of the patch is chosen such that when the patch is projected into the

reference image $R(p)$ it is seen as a $\mu \times \mu$ pixel square.

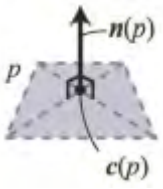

Figure 3.1: Patch Model [1].

3.2.2 Photometric Discrepancy Function

The photometric discrepancy function is used to determine whether the projections of p onto two images I_1 and I_2 match. If the patch projections in the images match closely then the patches most likely represent the same point on the surface. This function is used to determine whether a patch will accurately model the surface it is placed on. If the photometric discrepancy score between the proposed patch projections in I_1 and I_2 is low it implies that the image portions that the patches project on to are approximately the same. This means that the patch projections will most likely represent the same point.

The photometric discrepancy function for a patch p is defined as:

$$g(p) = \frac{1}{|V(p) \setminus R(p)|} \sum_{I \in V(p) \setminus R(p)} h(p, I, R(p)) \tag{3.1}$$

where:

$g(p)$ = is the photometric discrepancy score of p,

$V(p)$ = is the set of images in which $p$is visible,

$R(p)$ = is the reference image of p, and

$h(p, I_1, I_2)$ = is the pairwise photometric discrepancy of the patch projection in images $I_1$and I_2.

$V(p)$ is first estimated by checking the angle between the camera viewing $R(p)$ and the cameras of all the other input images. The images that are within a certain angle are assumed to be in $V(p)$.

The pairwise photometric discrepancy function, $h(p, I_1, I_2)$ is calculated using the following procedure:

1. Create a $\mu \times \mu$ grid for p that is centred at $\overline{\mathbf{c}}(p)$ and aligned with the image axes

2. Initialise the normal $\vec{\mathbf{n}}(p)$ such that it points into the optical centre of $O(R(p))$

3. Project the grid points into the images in $V(p)$.

4. Using bilinear interpolation sample the pixel colour at the projection of the grid points in image I_1 and I_2. The calculated colours in these images are denoted as

$\mathbf{q}(p, I_1)$ and $\mathbf{q}(p, I_2)$ respectively.

5. The pairwise photometric discrepancy $h(p, I_1, I_2)$ is given by one minus the normalised cross correlation (NCC) between $\mathbf{q}(p, I_1)$ and $\mathbf{q}(p, I_2)$.

Because the photometric discrepancy function makes use of pixel colours to determine correlation it can be sensitive to non-uniform lighting conditions. It may therefore not perform well when there are specular highlights in an image. This is dealt with simply by regarding images that are non-Lambertian as outliers and discarding patches with poor photometric discrepancy scores. The photometric discrepancy function is also sensitive to uniform texture. This leads to false positive matches in areas that are the same colour. However a large portion of these erroneous patches are removed in the filtering stages. With this in mind an improved set of visible images, $V^*(p)$ is defined, which can be considered as the set of images in which p is truly visible.

$$\begin{aligned} V^*(p) &= \{I | I \in V(p), h(p, I, R(p)) \leq \alpha\} \qquad (3.2) \\ \text{where:} & \\ V^*(p) &= \text{the set of images in which the patch is truly visible,} \\ I &= \text{the image currently being compared, and} \\ \alpha &= \text{the minimum photometric consistency value.} \end{aligned}$$

Using this new set of images in which p is truly visible an improved photometric discrepancy score can be calculated for a patch.

$$g^*(p) = \frac{1}{|V^*(p) \setminus R(p)|} \sum_{I \in V^*(p) \setminus R(p)} h(p, I, R(p)) \qquad (3.3)$$

By definition both $V(p)$ and $V^*(p)$ contain the reference image $R(p)$ as the first image in the set.

3.2.3 Image Model

One of the problems with using patches to model scenes or objects is the lack of connectivity information. For this reason each image is divided into cells that monitor which patches project into them. Every image is divided into a grid of $\beta_1 \times \beta_1$ pixel cells denoted as $C_i(x, y)$ where the subscript i refers to an image and x and y denote the position within the cell grid. Each cell stores the set of patches, $Q_i(x, y)$, that project into them. A patch p is reprojected into each image in $V(p)$. The patch will fall into a particular cell $C_i(x, y)$ then this cell will add p to $Q_i(x, y)$. Similarly a patch that is reprojected into $V^*(p)$ will fall in a cell $C_i^*(x, y)$ and this cell will store the set of patches $Q_i^*(x, y)$.

3.2.4 Patch Optimisation

The optimisation step is used to refine the geometric parameters, the patch centre $\overline{\mathbf{c}}(p)$ and patch normal $\overrightarrow{\mathbf{n}}(p)$ of a patch p, by minimising the photometric discrepancy score $g^*(p)$. Some assumptions are made to simplify the optimisation procedure. First the patch centre is constrained to lie on a line such that its reprojection into the reference image does not change. In other words $\overline{\mathbf{c}}(p)$ lies on the line between the optical centre of the camera corresponding to $R(p)$ and the position of $\overline{\mathbf{c}}(p)$ before the start of optimisation, denoted as $\overline{\mathbf{c_0}}(p)$. Secondly the patch normal can only have its pitch and yaw modified where pitch is rotation around the axis defined by the projection of the negative h-axis onto the patch plane and yaw is defined similarly using the projection of the negative v-axis of the image. In other words the patch may not rotate in its own plane. The assumptions make the optimisation problem have only three degrees of freedom. The optimisation is then performed using the conjugate gradient method.

If a patch does not have at least γ images in $V(p)$ before optimisation it is deleted. After the optimisation the number of visible images in $V(p)$ as well as in $V^*(p)$ is recomputed. If there are fewer than γ images in $V^*(p)$ the patch fails the optimisation stage and is deleted.

3.3 Initial Feature Matching

The initial feature matching step is used to create a sparse set of patches which were used to seed the expansion and filtering steps. First features are detected using Harris and difference of Gaussian detectors, which find corner and blob features. These features are ordered according to the strength of their response before they are matched. Features are matched across input images using the known position of the cameras and epipolar geometry. Initially an estimated position of a new patch is found using triangulation techniques. The patch is initialised and then optimised. If the optimisation procedure is successful the patch is created and stored using the image model defined in Section 3.2.3; if the optimisation fails then the patch and the primary feature used to generate it are discarded.

3.3.1 Feature Detection

3.3.1.1 Gaussian Kernels

A two-dimensional Gaussian kernel with standard deviation σ is defined as $G_\sigma(x, y)$ [27]. The zero-mean two-dimensional Gaussian kernel has the form $G_\sigma(x, y) = e^{-\frac{x^2-y^2}{2\sigma^2}}$. A useful property of the Gaussian kernel is that it is separable which allows the two-dimensional kernel to be split into two one-dimensional kernels, $G_\sigma(x) = e^{\frac{-x^2}{2\sigma^2}}$ and $G_\sigma(y) = e^{\frac{-y^2}{2\sigma^2}}$. See Appendix A for the derivation. The separated kernels can be applied sequentially to the image to get the same results as one would get when applying the two-dimensional kernel to the same image. The Gaussian kernel is a continuous function which needs to be

discretised to work with images. This is achieved by sampling the continuous function at regular intervals.

3.3.1.2 Harris Feature Detector

The response of the Harris filter [28] is defined in Equation 3.4 as:

$$\begin{aligned} H &= \det(M) - \lambda \text{trace}^2(M) \qquad (3.4) \\ \text{where:} & \\ M &= G_{\sigma_1} * \left(\nabla I \nabla I^T\right), \\ \nabla I &= \begin{bmatrix} \frac{\partial I}{\partial x} & \frac{\partial I}{\partial y} \end{bmatrix}^T, \text{ and} \\ * &= \text{the 2D convolution operator.} \end{aligned}$$

The standard deviation for the Gaussian kernel is $\sigma_1 = 1$ and the value of lambda is $\lambda = 0.06$. See Appendix A.2 for full calculation of the response of the Harris filter.

3.3.1.3 Difference of Gaussians Feature Detector

The difference of Gaussians feature detector [29] is used to identify corner and blob features in a grayscale image by subtracting two differently blurred versions of the same image. The response of the difference of Gaussians detector at a point in the image is defined in Equation 3.5 as:

$$\begin{aligned} D &= \left|\left(G_{\sigma_0} - G_{\sqrt{2}\sigma_0}\right) * I\right| = \left|G_{\sigma_0} * I - G_{\sqrt{2}\sigma_0} * I\right| \qquad (3.5) \\ \text{where:} & \\ G_{\sigma_0} * I &= \text{the image } I\text{convolved with a Gaussian} \\ & \quad \text{kernel with standard deviation } \sigma_0, \text{ and} \\ G_{\sqrt{2}\sigma_0} * I &= \text{the image } I\text{convolved with a Gaussian} \\ & \quad \text{kernel with standard deviation } \sqrt{2}\sigma_0. \end{aligned}$$

3.3.1.4 Detecting Features in Windows

When using feature detectors it is important to ensure uniform coverage of the image as seen in Figures 4.3 and 4.4. This is done by using a $n \times n$ pixel sliding window. The window is shifted over the image a pixel at a time and the η local maxima or minima with the strongest responses, that have not been selected before, are saved from the current window. The feature matching window is used to ensure that there is an even distribution of feature points over the entire image.

3.3.2 Feature Matching

3.3.2.1 Feature Matching Procedure

Once features have been detected, the next step is to attempt to match multiple features across the set of input images to create an initial sparse set of patches. To do this consider an image I_i with a camera that views it with optical centre $\overline{\mathbf{O}}(I_i)$. For each feature f detected in image I_i we collect all features, f', of the same type in another image, I_j, that lie within two pixels of the epipolar line that is created when f is projected into image I_j. Then using the feature pair (f, f') it is possible to triangulate a three-dimensional point that will become the centre of a candidate patch. These points are sorted in order of increasing distance from $\overline{\mathbf{O}}(I_i)$. From this list of feature pairs the following procedure is followed to create a candidate patch:

$$\begin{aligned}
\overline{\mathbf{c}}(p) &\leftarrow \text{Triangulation}\left(f, f'\right) && (3.6)\\
\overrightarrow{\mathbf{n}}(p) &\leftarrow \overrightarrow{\overline{\mathbf{c}}(p)\,\overline{\mathbf{O}}(I_i)} \Big/ \left|\overrightarrow{\overline{\mathbf{c}}(p)\,\overline{\mathbf{O}}(I_i)}\right| && (3.7)\\
R(p) &\leftarrow I_i && (3.8)
\end{aligned}$$

where:

$$\begin{aligned}
\overline{\mathbf{c}}(p) &= \text{the 3D centre point of the patch,}\\
\left(f, f'\right) &= \text{is the feature pair from images } (I_i, I_j),\\
\overrightarrow{\mathbf{n}}(p) &= \text{the 3D vector defining the patch normal,}\\
\overline{\mathbf{O}}(I_i) &= \text{the optical center of the camera viewing image } I_i, \text{ and}\\
R(p) &= \text{the reference image of the patch.}
\end{aligned}$$

The initial set of visible images $V(p)$ is created by assuming that the patch is visible in an image only if the angle between the patch normal and the line joining the patch centre and the optical centre of the camera is above a certain threshold.

$$V(p) \leftarrow \left\{ I_i \middle| \overrightarrow{\mathbf{n}}(p) \cdot \overrightarrow{\overline{\mathbf{c}}(p)\,\overline{\mathbf{O}}(I_i)} \Big/ \left|\overrightarrow{\overline{\mathbf{c}}(p)\,/\overline{\mathbf{O}}(I_i)}\right| > \cos(\iota) \right\} \quad (3.9)$$

where:

$$\begin{aligned}
V(p) &= \text{the set of images in which } p\text{should be visible,}\\
I_i &= \text{the image that is being tested to see if it is in } V(p),\\
\overline{\mathbf{O}}(I_i) &= \text{the 3D optical centre of the camera viewing } I_i, \text{ and}\\
\iota &= \text{the angular threshold, set to } \frac{\pi}{3}.
\end{aligned}$$

Next $V^*(p)$ is initialised using Equation 3.3. Finally $\overline{\mathbf{c}}(p)$ and $\overrightarrow{\mathbf{n}}(p)$ are optimised using the optimisation procedure described in Section 3.2.4. Once the geometric parameters of the patch have been refined, $V(p)$ and $V^*(p)$ are recalculated using the same methods as before. Finally, if $|V^*(p)| \geq \gamma$ (in other words if there are at least γ images with low photometric discrepancy) the patch p is successfully created. The image model (Section

3.2.3) is updated by projecting p into its image cells $C_i(x,y)$ and $C_i^*(x,y)$, and $Q_i(x,y)$ and $Q_i^*(x,y)$ are updated. Any features that lie in cells now occupied by the patch are deleted, since only one patch per cell is required for the feature matching step.

3.4 Patch Expansion

The goal of the expansion procedure is to create at least one patch in every image cell $C_i(x,y)$. The procedure for patch expansion is to use existing patches, and expand them outwards into neighbouring cells. During expansion it is assumed that adjacent image cells correspond to roughly the same area on the object or scene. As such expansion does not account for depth discontinuities which are resolved in the filtering stage. Expansion is divided into two steps, selection and expansion.

In the selection step the adjacent and immediately diagonal cells to a patch are selected as possible expansion areas. If one of these cells is currently occupied there is no reason for it to be considered for expansion.

The set of cells are then checked to see if they are valid for expansion. If the patch is a neighbour of one of the cells then the cell is an expansion candidate. A new patch is created in each candidate cell and initialised to be identical to the expanding patch except for the fact that the new patch centre projects into the centre of the candidate image cell. Using this starting point the new patch is optimised and if it passes the optimisation step the expansion into that cell is successful.

3.4.1 Selecting Cells for Expansion

The set of cells that are neighbours of a patch p are stored in $\mathbf{C}(p)$.

$$\mathbf{C}(p) = \left\{C_i\left(x',y'\right) | p \in Q_i(x,y), \left|x-x'\right| + \left|y-y'\right| = 1\right\} \tag{3.10}$$

where:

$$\begin{aligned} \mathbf{C}(p) &= \text{the set of cells which are neighbours of } p, \\ C_i\left(x',y'\right) &= \text{the cell that may be a neighbour of } p, \text{ and} \\ Q_i(x,y) &= \text{the set of patches that project into } C_i(x,y). \end{aligned}$$

Once the neighbouring cells have been identified it is possible to determine in which cells the expansion is required. There is no need to expand into a cell if that cell already contains a patch that is a neighbour of p. If an image cell $C_i(x',y') \in \mathbf{C}(p)$ contains a patch p' which is a neighbour of p, $C_i(x',y')$ can be removed from $\mathbf{C}(p)$. Once again the subscript i of a cell $C_i(x,y)$ represents the image I_i that contains that cell and x and y represent the position in the cell grid. Patches p and p' are said to be neighbours if:

$$2\rho_1 > \left|\left(\overline{\mathbf{c}}(p) - \overline{\mathbf{c}}\left(p'\right)\right) \cdot \vec{\mathbf{n}}(p)\right| + \left|\left(\overline{\mathbf{c}}(p) - \overline{\mathbf{c}}\left(p'\right)\right) \cdot \vec{\mathbf{n}}\left(p'\right)\right| \tag{3.11}$$

where:

$$\rho_1 = \text{the distance in 3D space that corresponds to an image displacement of } \beta_1 \text{pixels in } R(p).$$

The distance above is calculated on the plane defined by the patch centres and normals. It is unnecessary to expand into an empty cell if there is a depth discontinuity when viewed from the camera that corresponds with I_i. In practice it is difficult to judge if a depth discontinuity is present, as such an expansion is deemed unnecessary if $Q_i^*(x,y)$ already contains patches.

3.4.2 Expansion Procedure

The procedure to generate a more dense collection of patches begins by considering each image cell $C_j(x,y)$ in $\mathbf{C}(p)$. A new patch p' is created in $C_j(x,y)$. The patch centre $\overline{\mathbf{c}}(p')$ is initialised by projecting a line from $\overline{\mathbf{O}}(I_j)$ through the centre of $C_j(x,y)$ until it intersects the plane defined by $\overline{\mathbf{c}}(p)$ and $\overrightarrow{\mathbf{n}}(p)$. This point of intersection is the expanded patch's centre. Then $\overrightarrow{\mathbf{n}}(p')$, $R(p')$, and $V(p')$ are initialised from the corresponding parameters of p. The improved set of visible images $V^*(p')$ is created using Equation 3.2.

The optimisation procedure explained in Section 3.2.4 is run on p'. If, after the optimisation, the set of truly visible images, $V^*(p')$, contains less than γ images the patch fails the optimisation and is discarded. Any images in which p' should be visible according to a depth map test are added to the set $V(p')$, then $V^*(p')$ is updated once again. Finally the number of images in the set $V^*(p')$ is used to determine whether the patch exists. In other words if $|V^*(p')| \geq \gamma$ the patch p' is created and projected into $C_j(x,y)$ and $C_j^*(x,y)$ and $Q_j(x,y)$ and $Q_j^*(x,y)$ are updated accordingly.

3.5 Patch Filtering

After the patch expansion step the number of patches has increased greatly. However many of these patches may be erroneous and should therefore be removed. Since the expansion step is only concerned with creating new patches, and not with ensuring that they fit with nearby patches, it is necessary to perform filtering to ensure that the patches accurately model the surface.

To this end four filters are used to make sure the generated patches provide an accurate representation of the surface which they are modeling.

The first filter enforces visibility consistency by removing patches that are not neighbours of other patches that project into the same image cell. $U(p)$ is the set of patches that are not consistent with the current visibility information of the patch p. This means that patch p and another patch p' are not neighbours (where neighbours are defined as in Equation 3.11) but are stored in the same cell of an image where p is visible. The patch p is considered an outlier and removed if:

$$|V^*(p)|\,(1 - g^*(p)) < \sum_{p_i \in U(p)} (1 - g^*(p_i)) \tag{3.12}$$

In other words if the sum of photometric consistency scores of the patches in $U(p)$ is greater than the photometric consistency score of p multiplied by the number of images from which this score is calculated then p is an outlier and is removed.

The second filter enforces a more strict visibility consistency. For each patch p a depth map test is used to determine in how many images in $V^*(p)$ the patch should be visible. If this number, denoted as $|V_D^*(p)|$, is too small then the patch is discarded as an outlier.

$$|V_D^*(p)| < \gamma \tag{3.13}$$

The third filter enforces a weak form of regularisation. For each patch p collect the patches p' that lie in its own and adjacent cells in all images of $V(p)$. This set of patches is called $A(p)$. Next the set of patches in $A(p)$ that are neighbours of p according to Equation 3.11 are added to $N(p)$. Patches that are neighbours can be thought of as being connected to one another. Therefore if there are many patches that are close to p, but not neighbours of p, then p can safely be labeled as an outlier. If the following inequality holds then p is removed.

$$\frac{|N(p)|}{|A(p)|} < 0.25 \tag{3.14}$$

The fourth filter works in much the same way as the third filter, except instead of operating on a single patch it operates on connected groups of patches. If the group of patches have a discontinuity with the surrounding model they are removed as outliers.

3.6 Polygonal Mesh Reconstruction

The polygonal mesh reconstruction stage uses the dense point cloud created after the final iteration of expansion and filtering to create a closed mesh model of the surface. Once this mesh is created it is optimised such that the vertices are closer to the surface that it is modeling.

3.6.1 Mesh Creation

The mesh is created using the Poisson surface reconstruction library written by Kazhdan *et al.* [30]. Their method treats the problem as a spatial Poisson problem. This allows the method to create a mesh that is the global best fit since it does not need to divide the patches into smaller segments. It creates a triangular mesh from the reconstructed patches with smaller triangles in densely covered areas and larger triangles in sparsely covered areas.

3.6.2 Mesh Optimisation

The closed mesh created using the method outlined in Section 3.6.1 is optimised to create a final closed mesh. This is achieved by optimising an energy function that optimises the 3D position of the vertices with respect to patches that are generated from each vertex.

For each vertex $\overline{\mathbf{v}}_i$ that has been created by Poisson surface reconstruction create the set of images, $V(\overline{\mathbf{v}}_i)$, in which the vertex should be visible according to a depth map test. For each unique pair of images (I_j, I_k) in $V(\overline{\mathbf{v}}_i)$ create a patch that is centred at $\overline{\mathbf{v}}_i$ with a normal that is orthogonal to the reconstructed surface. This patch is then optimised according to the procedure discussed in Section 3.2.4 but with only two images. All patches are then added to the set $P(\overline{\mathbf{v}}_i)$. This set of patches is used to optimise the position of the vertex according to Equation 3.15 below:

$$E'_p(\overline{\mathbf{v}}_i) = \zeta_3 \sum_{p \in P(\overline{\mathbf{v}}_i)} 1 - \exp\left(-\left(\frac{\overrightarrow{\mathbf{n}}(p) \cdot (\overline{\mathbf{c}}(p) - \overline{\mathbf{v}}_i)}{\tau/4}\right)^2\right) \tag{3.15}$$

where:

ζ_3 = the linear combination weight (set to 1.0 or 4.0),

$\overrightarrow{\mathbf{n}}(p)$ = the patch normal,

$\overline{\mathbf{c}}(p)$ = the patch centre,

$\overline{\mathbf{v}}_i$ = the position of the vertex, and

τ = the average edge length of the mesh.

Chapter 4

Methodology and Results

4.1 Introduction

The modified PMVS algorithm was tested using the skull dataset which is described in Section 4.8.1. Eighteen experiments were conducted with three variables: image size, cell size, and patch size. By changing the values on these variables the overall quality of the reconstruction can be changed. With higher quality settings (full size images, one pixel cells, and large patches) the best models are created, however these models also incur the longest running time.

The number of patches, and thus the density of the reconstruction of 3D model, are directly related to image and cell size. The algorithm runs by making sure at least one patch exists in every image cell which means that a single cell may contain many patch projections. However if a patch projection exists in a cell the expansion step will not create a new patch in that cell. For this reason there is not a noticeable difference in the patch density of the outputs of the original and new algorithms.

The original PMVS algorithm [5] made use of the GNU Scientific Library (GSL) to optimise patches (Section 3.2.4). It also used a CPU based algorithm to detect features in the input sequence of images (Section 3.3.1). The mesh creation stage of the original algorithm could use a slightly modified version of Furukawa's visual hulls [31] or Poisson surface reconstruction [30]. The original mesh optimisation stage used each generated vertex to create a new set of patches to optimise the vertex.

The modified PMVS algorithm makes use of a CUDA based L-BFGS [32] optimiser to optimise patches. It also used a CUDA texture memory based feature detector to detect difference of Gaussian and Harris features. The mesh creation stage of the modified algorithm used the Poisson surface reconstruction [30] library to create a closed mesh. The mesh optimisation stage was changed to use the existing image model to find patches that correspond to a vertex and optimise that vertex.

4.2 Specifications

4.2.1 System

The system that was used to implement and test this program was a desktop PC running Ubuntu 11.10 64 bit. It had an Intel i7 3.6 GHz CPU with eight computational cores, 16 GB of 1600MHz DDR3 RAM, and two NVIDIA GeForce GTX 550Ti 1GB GPUs with 192 CUDA cores per card. The operating system and libraries were installed to a 60GB solid state drive.

4.2.2 Programs

The following programs and libraries were used in this implementation. NVIDIA's CUDA library was used to port some processing onto the GPU. The pThreads [26] library was used to create multiple CPU threads. The boost library [33] was used for its shared pointer functionality. The L-BFGS [24] library was used to run optimisation on the GPU. The PMVS-2 [5] program implemented by Furukawa and Ponce was used as the basis for the implementation. Finally the Poisson surface reconstruction [30] library is used to create the initial mesh model.

4.3 Test Implementations

Initial tests were performed using the OpenCV library [34]. The difference of Gaussians and Harris detectors were implemented on both CPU and GPU. An initial implementation of the PMVS algorithm was implemented using OpenCV up to the end of the initial matching stage. However the execution time of this stage was extremely slow and Furukawa's original PMVS implementation was used as a starting point.

4.4 Initial Feature Matching

The initial feature matching step, discussed in Section 3.3, creates a sparse set of three-dimensional patches that are passed to the next stage of the program. The pipeline of this step is shown in Figure 4.1. First features are found in pairs of images using the Harris and Difference of Gaussian operators. The features detected using these operators are defined to either have the type Harris or DoG. Features of the same type are compared to one another using epipolar geometry. If a pair of features passes the epipolar geometry test then they are used to triangulate a three-dimensional position which is used as a candidate patch centre. Finally if the patch is visible in at least three input images and the image segments that the patch projects into have an NCC score as defined in Section 3.3, the patch is confirmed and stored, causing the internal models to be updated according to the method in Section 3.2.3.

Algorithm 4.1 Initial Feature Matching Procedure. Adapted from [1].

```
Input: Features detected in input images.
Output: Initial sparse set of patches P.
P ← φ
For each image I_i with optical centre O(I_i) do

    For each feature f detected in I_i do
        For each feature f' in every other image I_j do

            F ←{Features f' that satisfy epipolar consistency with f};
            CC ←{Triangulated candidate centres from (f, f')};
            Sort CC in order of increasing distance from O(I_i);
            For each candidate centre in CC do
              Create a patch candidate p;
              Initialise c(p), n(p), and R(p); // Equations(3.6, 3.7, 3.8)
              Initialise V(p) and V*(p); // Equations(3.9, 3.2)
              Optimise the geometric parameters of p;
              Recompute V(p) and V*(p); // Equations(3.9, 3.2)
              If |V*(p)| < γ then

                Try next candidate centre; //failure
              Else
                Update cells with p;
                Remove features from cells where p was stored;
                Add p to P; //success
```

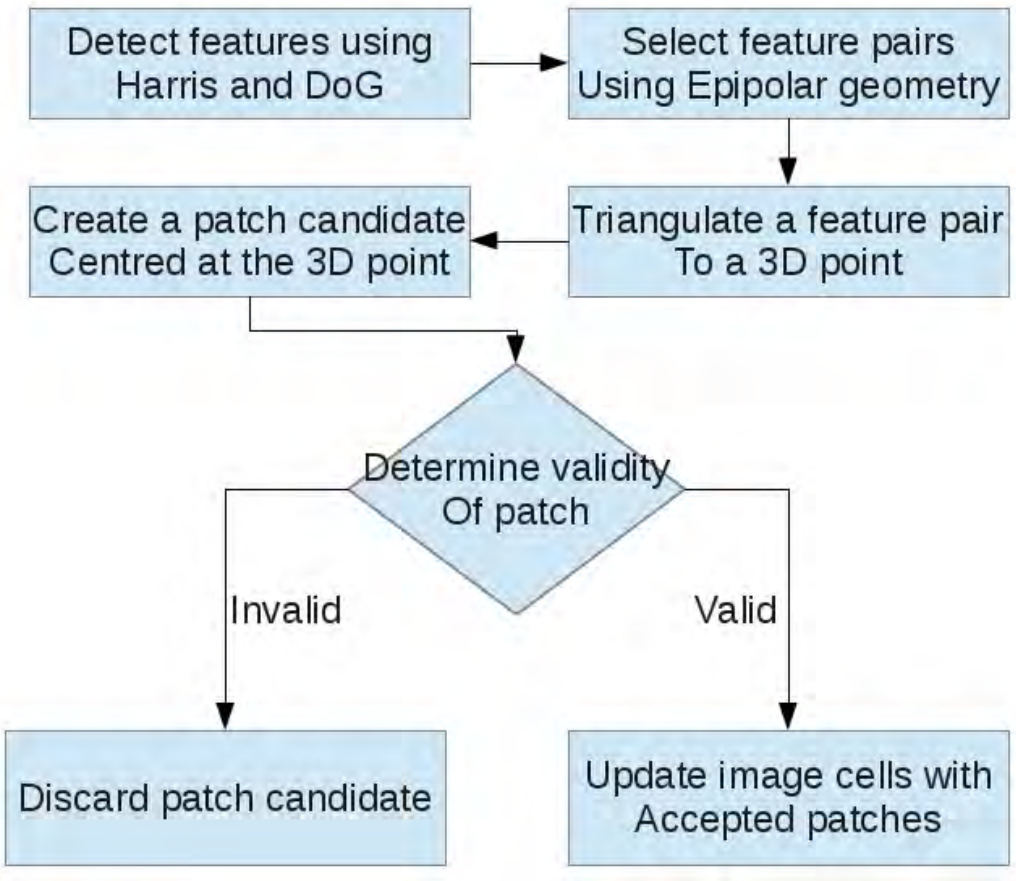

Figure 4.1: Feature Matching Pipeline.

4.4.1 Feature Detection

In the feature matching step two feature detection methods are used to find blob, corner, and edge features. The feature detectors that were implemented were the Harris, described in Section 3.3.1.2, and the difference of Gaussians (DoG), described in Section 3.3.1.3. Both of these filters were run over small windows within the image to get uniformly distributed features across all input images. The feature detector passes the set of feature points to the next stage in the pipeline, as shown in Figure 4.1. Both feature detectors make use of Gaussian smoothing to remove noise and find features. This part was implemented on the GPU using CUDA and the execution time of the detection stage was sped up by two times when the standard deviation of the Gaussian function was in the range of $\sigma = 1.0$ to $\sigma = 2.0$. These are the values that are used for the datasets discussed in this thesis. The feature detection was sped up by ten times when the standard deviation of the Gaussian function was set to $\sigma = 4.0$ (Shown in Figure 2.5).

This increase in execution time was because of the highly parallel nature of Gaussian smoothing. Gaussian smoothing averages the value of the pixels in the neighbourhood of a particular pixel. This means that each GPU thread is responsible for the averaging of a single output pixel, meaning that many output pixels can be computed simultaneously.

Figure 4.2: Sample images of the skull data set.

4.4.1.1 Gaussian Smoothing

The Gaussian smoothing algorithm was implemented on the GPU using CUDA. The two-dimensional Gaussian kernel was split into its one-dimensional components as explained in Section 3.3.1.1. The pixel data from a single input image was loaded into the GPU texture memory by the CPU. Then the kernel was loaded into a different texture in the GPU texture memory. This allows the size of the kernels to change during run time. The GPU algorithm performs smoothing in the h- and v-directions before returning the smoothed image to the CPU.

An image is loaded into the texture memory of the GPU in grayscale. Then the first and second derivatives of this image are computed in parallel on the GPU. This portion of the detection stage is common to both DoG and Harris and as such it was used as the primary area to improve the detection stage.

4.4.1.2 Harris Feature Detector

The Harris feature detector was implemented according Section 3.3.1.2. Initially the first derivatives of an image, $\frac{dI}{dh}$ and $\frac{dI}{dv}$, are computed as described in Section 4.4.1.1. These first derivative images are then differentiated to find the second derivatives, $\frac{d^2I}{dh^2}$, $\frac{d^2I}{dv^2}$, and $\frac{d^2I}{dhdv}$. The second derivatives are then used to create the response image $\mathbf{H}$. The following values are calculated for each pixel (h, v):

$$D(h,v) = \frac{d^2I}{dh^2} \cdot \frac{d^2I}{dv^2} - \left(\frac{d^2I}{dhdv}\right)^2 \tag{4.1}$$

$$\text{trace}(h,v) = \frac{d^2I}{dh^2} + \frac{d^2I}{dv^2} \tag{4.2}$$

The results of Equations 4.1 and 4.2 are used to create the response image $\mathbf{H}(h,v) = D(h,v) - 0.06 \cdot \text{trace}(h,v)$.

All values in $\mathbf{H}$ that are either local minima or local maxima are used to create the final response image. Finally a small sliding window is moved across the image and the four features with the strongest response in a window are saved. Figure 4.3 shows three images with Harris features highlighted in blue. The features in these images are distributed evenly across the entire image however the actual response value is not shown in these images.

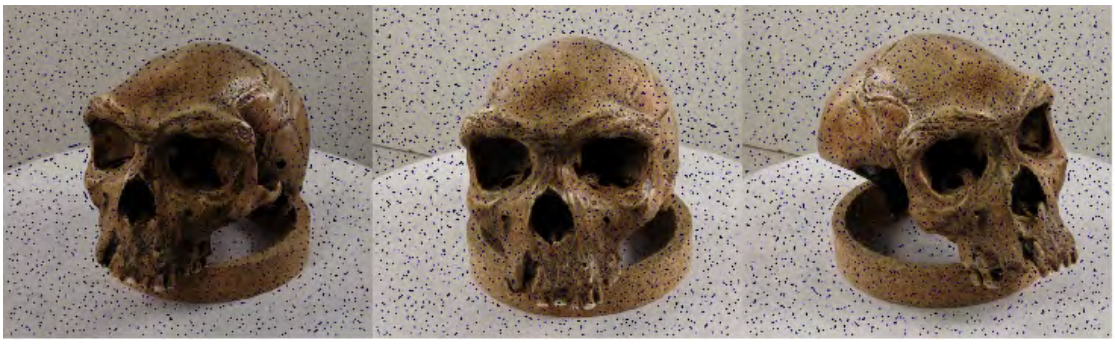

Figure 4.3: Sample output of the Harris Feature Detector.

4.4.1.3 Difference of Gaussians Feature Detector

The Difference of Gaussians feature detector was implemented as described in Section 3.3.1.3. It locates features by comparing an image to versions of itself at different blur levels. It is an approximation of the Laplacian of Gaussian detector [35]. The detector that was implemented compares two different versions of the same image, referred to as $\mathbf{I}_C$ and $\mathbf{I}_N$, which denote the current and next levels of blurred images respectively. These images are then compared to create a DoG image:

$$\mathbf{D}(h, v) = \mathbf{I}_N(h, v) - \mathbf{I}_C(h, v)$$

The resultant difference image $\mathbf{D}(h, v)$ is then searched for local maxima and minima which become the detected features. Robust feature points tend to survive the Gaussian blurring and are therefore highlighted as local minima or maxima. Figure 4.4 shows three images with difference of Gaussian features in red.

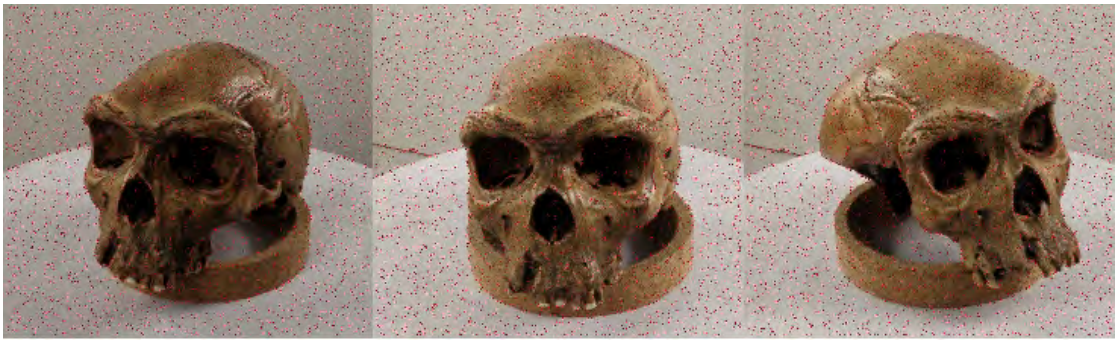

Figure 4.4: Sample output of the Difference of Gaussians feature detector.

4.4.2 Candidate Creation

Candidate creation was run in parallel on the CPU such that each thread is responsible for a single reference image I. All other images are added to the set $\mathbf{I}$. No mutexes are required because it does not matter if approximately the same patch is created twice, since one of them will be removed in the filtering step later in the process.

4.4.2.1 Finding Feature Pairs

A pair of images is used to identify two features that correspond to the same 3D point. An image I is used as the reference image. Next a secondary image I' is selected from the full set of images $\mathbf{I}$. Using the known camera parameters P and P', which correspond to images I and I' respectively, the angle between the cameras is found. Each camera has an optical axis denoted as $\overrightarrow{\mathbf{O}_p}$ and $\overrightarrow{\mathbf{O}'_p}$ respectively.

$$\cos\theta = \left(\overrightarrow{\mathbf{O}_p} \cdot \overrightarrow{\mathbf{O}'_p}\right) \tag{4.3}$$

If θ is less than 60° then I' is kept in the set $\mathbf{I}$ otherwise it is removed. This step reduces the number of secondary images that need to be searched for features, thereby reducing the processing required.

Features from the detection stage are loaded into image cells $C(h, v)$. The size of the image cells is an experimental parameter and is set to either 1×1 pixels or 2×2 pixels. The features are then sorted in descending order of response. For each feature, f, in image I an epipolar line is drawn in image I'. This is done by drawing a line that starts at $\overline{\mathbf{O}}(I)$ and passes through $f(h, v)$ in the virtual image plane. When this 3D line is reprojected into image I' it creates an epipolar line, which intersects the left and right epipole, and should contain the point that corresponds to f. If there are any features f' in I' within two pixels of the epipolar line then the feature pair (f, f') is used to create a patch candidate. Several such pairs are found and false positives will be discarded in the next stage.

4.4.2.2 Initialising a Three Dimensional Patch

A 3D patch p is created and defined as follows. Using 3D triangulation as described in Section 2.2.6.2 the patch centre $\overline{\mathbf{c}}(p)$ can be calculated. The feature pair (f, f') is projected into 3D space, the point of closest intersection is then said to be $\overline{\mathbf{c}}(p)$. A rough estimate of the patch normal $\overrightarrow{\mathbf{n}}(p)$ is generated by creating a unit vector that originates at $\overline{\mathbf{c}}(p)$ and points towards the optical centre of the camera which corresponds to the reference image I. This patch normal is used to orient the patch in such a way as to approximate the orientation of the surface. The normal will be optimised to provide a better orientation later. The reference image of the patch is defined as $R(p)$ and is initially set to I. Next the set of visible images for the patch is created. I and I' are added to $V(p)$, with I as the first and I' as the second image in the set. The angle between the patch normal and the other images in $\mathbf{I}$ is found. If this angle is less than 60° the image is added to $V(p)$. If there are at least three images in $V(p)$ the patch is refined.

4.4.2.3 Refining a Three Dimensional Patch

A 2D rectangle is created and centred at the point where $\overline{\mathbf{c}}(p)$ reprojects into $R(p)$. This rectangle is then projected out to the plane defined by $\overline{\mathbf{c}}(p)$ and $\overrightarrow{\mathbf{n}}(p)$. A $\eta \times \eta$ grid (where η is set to either five, seven, or nine) is created on this rectangle with the centre grid point corresponding to $\overline{\mathbf{c}}(p)$ and aligned with the axes of $R(p)$. By comparing this grid between

all of the images in $V(p)$ it is possible to find the set $V^*(p)$. The set $V^*(p)$ contains the images in which the values of the grid projections are very similar. The first image of $V^*(p)$ is set to $R(p)$. Then for each other image in $V(p)$ the normalised cross correlation between the projection of the grid points into $R(p)$ and the image in $V(p)$ is found. If the NCC score is greater than 0.7 (this initial NCC value was empirically determined by Furukawa and Ponce [1]) the image is added to $V^*(p)$. If the size of $V^*(p)$ is greater than three the patch is reserved for optimisation, otherwise it is removed.

4.4.2.4 Optimising a Three Dimensional Patch

The patch optimisation procedure described in Section 3.2.4 is used to improve the patch position such that it more closely models the surface of the object or scene. The objective function being optimised is the normalised NCC score as shown in Equation 3.3. The following geometric constraints are placed upon the system: $\overline{\mathbf{c}}(p)$ is constrained to lie on the line that joins it and optical centre of the camera viewing the reference image, $C(R(p))$. The patch is constrained to rotate using only pitch and yaw. This creates a three degree of freedom optimisation procedure which is solved using the Limited-memory Broyden-Fletcher-Goldfarb-Shanno (L-BFGS) method [23, 24], which is implemented using CUDA [32]. Every evaluation of the optimisation function requires several image loads since the patch is repeatedly projected into the images of $V^*(p)$ to determine the NCC score. The optimisation is run until the average NCC score reaches a maximum value, meaning that the sub-images that the patch projects into correspond very highly with one another. Now $V(p)$ and $V^*(p)$ are cleared and reinitialised as described in Section 3.2.4. If $V^*(p)$ still has more than three images the patch generation is deemed a success, and the patch p is created. It is then projected into image cells $Q(x, y)$ and $Q^*(x, y)$ and any features that were in those cells are deleted.

At the end of the initial feature creation step a noisy and sparse point cloud has been created as shown in Figure 4.5. This point cloud shows the approximate shape of the object being modeled with noise surrounding the object. All of these patches will be expanded to create more patches. Thereafter the filtering step will remove many of the noisy patches as well as some salient patches.

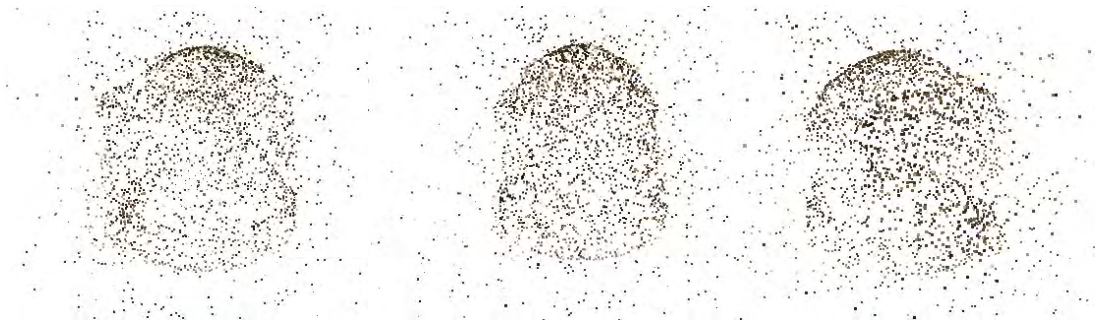

Figure 4.5: Model after initial feature matching step.

4.5 Patch Expansion

The patch expansion procedure runs a total of three times. Its purpose is to create a more dense set of patches that closely match the object that is being modeled. The patch expansion step does not check for any sort of occlusion or depth discontinuities because this is left to the filtering step.

The expansion procedure works by iterating over the set of existing patches, denoted as $\mathbf{P}$. First a patch p is selected from $\mathbf{P}$. A search is done for any patches that are neighbours of p. First p is reprojected into the images in $\mathbf{V}^*(p)$. At each of these locations a check is done to see if there are any other patches in neighbouring cells (see Section 3.2.3). If there are no neighbouring patches in a particular cell then the expansion begins.

A new patch, p', is created at the centre of the empty image cell. Specifically it is set to be equal to the 3D intersection point of the ray that projects from the optical centre of the camera viewing $R(p)$, through the centre of the empty image cell and the plane defined by the patch p. All the other patch parameters are set to be equal to the corresponding values of p. This is a very rough approximation and the patch is optimised to improve its position.

The optimisation step is then run on this new patch. If after the optimisation there are at least three images in $\mathbf{V}^*(p')$ then the patch expansion was successful and the patch p' is added to the list of patches. It should be noted that patches created during one step of the expansion procedure will not themselves be expanded in that same step, only in subsequent steps. Algorithm 4.2 shows the expansion process as explained above.

The expansion process runs three times over the course of the execution of the program. Figures 4.6, 4.7 and 4.8 show the reconstructed model at the end of the three expansion steps. Figure 4.6 shows the first call to the expansion step that runs after initial feature matching (shown in Figure 4.5). At this stage there are still many noises patches but the object being reconstructed is clearly visible. Figure 4.7 shows the second call to the expansion step that occurs after the first filtering step. The majority of the noisy patches have been removed leaving a clear view of the object being reconstructed. Figure 4.8 shows the final call to the expansion step that occurs after the second filtering step. Very few noisy patches remain and the surface of the object is densely covered in patches.

Algorithm 4.2 Patch Expansion Algorithm. Adapted from [1].

```
Input: Set of patches P from the feature matching step.
Output: Expanded set of reconstructed patches.
While P is not empty

    Select and remove a patch p from P;
    For each image cell C_i(x, y) containing p
        Collect a set C of image cells for expansion;
        For each cell C_i(x', y') in C
            Create a patch candidate p'
            Set c(p') according to the method described in...
            n(p') ← n(p);
            R(p') ← R(p);
            V(p') ← V(p);
            Compute V*(p');
            Optimise the geometric components of p';
            Add visible images to V(p') using a depth-map test;
            Recompute V*(p');
            If |V*(p')| < γ
              Select a new cell from C;
            Add p' to P;
            Add p' to Q_j(x, y) and Q*_j(x, y);
```

Figure 4.6: Model after the first expansion step.

Figure 4.7: Model after the second expansion step.

Figure 4.8: Model after the third expansion step.

4.6 Patch Filtering

The patch filtering step is run after the expansion step and this filtering culls the number of patches significantly. This strict filtering serves a purpose here as the patch expansion step does not check for any occlusions or depth discontinuities. The patch filtering step makes use of four filters. The purpose of the filtering step is to make sure that reconstructed patches match other patches in their neighbourhood.

4.6.1 Filtering of Outliers

The first filter in the pipeline removes patches that do provide a response that is as strong as the best neighbouring patch. First patches are collected and divided into a number of groups with this number set to the number of threads that are run (in this case eight). A patch is selected and its NCC score is computed as described in Section 3.2.2. Then each patch that is in a cell that is a neighbour of $C(p)$ is tested to see if it is a neighbour of p. If p' is a neighbour of p the NCC score of p' is stored and denoted as ncc_1. This is repeated for all patches in neighbouring cells. Each time if the NCC score of the tested patch p' is greater than ncc_1, then ncc_1 is replaced with the NCC score of p'.

The second stage runs in a similar fashion to the first, except only images in $V^*(p)$ are considered. Patches that are in cells that are neighbours of the cell containing p are stored in $\mathbf{C}(p)$. Then for each patch p' in $\mathbf{C}(p)$ the depth of p and p' are computed. If p and p' are neighbours and the depth of p is less than the depth of p' then p' occludes p. The NCC score of the patch p' with the highest NCC is stored in ncc_2.

Finally a value $gain = \text{NCC}(p) - ncc_1 - ncc_2$ is calculated where $\text{NCC}(p)$ is the average normalised cross correlation of the patch in the images of $V^*(p)$. If this gain is less than zero the patch is considered an outlier and is removed from the set of patches. The NCC score of a patch represents how closely it models the surface. While the gain value provides a relative measurement of how closely the patch p models the surface. If $\text{NCC}(p)$ is not as high as the sum of the best NCC scores of two neighbouring patches, then p may be an outlier and is removed.

4.6.2 Remove Occluded Patches

The second filter removes a patch p if it is occluded by another patch p' that is a neighbour of p. Equation 3.11 is used to determine if patches are indeed neighbours. If the ray from the centre of the camera intersects one of the neighbouring patches before intersecting p then p is said to be occluded and is removed. Threads for this filter select a single image as reference and use this image to determine visibility. A patch p is said to be occluded if there is another patch p' in its cell, or its neighbouring cells, that is closer to the camera viewing p. Any occluded patches are discarded.

4.6.3 Remove Patches that Lack Neighbours

The third filter removes patches that either do not have enough neighbouring patches, or that are not consistent with their neighbours. Each thread selects a single reference image to operate upon. For each patch p that projects into the image, the neighbouring cells of p are found using Equation 3.10. These neighbouring cells are checked for patches, which are stored in $\mathbf{C}(p)$. Then for each patch p' in $\mathbf{C}(p)$ Equation 3.11 is used to determine whether p and p' are neighbours if this is the case then p' remains, otherwise it is removed from $\mathbf{C}(p)$. If there are at fewer than six patches in $\mathbf{C}(p)$ then p is removed for being inconsistent.

If the patch p is consistent, i.e. it has enough neighbouring patches, then the depth of p is checked in the images of $V^*(p)$. If the depth of the patch is consistent with the depth of its neighbouring patches in these images the patch p is kept, otherwise it is removed.

4.6.4 Remove Small Groups of Erroneous Patches

The final filter selects a patch p and finds all of the patches that are connected to it. A patch p' is said to be connected to p if it is a neighbour of p according to Equation 3.11. If the size of this group of patches is determined to be too small the entire group of patches is removed as an outlier.

The previous three filters ensure that a single patch is consistent with the information of the patches in its neighbourhood. However if several erroneous neighbouring patches are created they will not be removed by these filters. Thus the filter to remove connected components is used to remove these small groups of erroneous patches.

Figures 4.9, 4.10 and 4.11 show samples of the output of the filtering step which is called immediately after the expansion. Figure 4.9 shows the results of the first call to the filtering step, the majority of the noisy patches are removed as well as some salient patches. Figure 4.10 shows the second call to the filtering step. There are now very few noisy patches many of the patches being removed in this step are correct patches with low photometric consistency scores. Figure 4.11 shows the final call to the filtering stage with almost all noisy patches removed and a dense set of patches that closely model the surface of the skull.

Figure 4.9: Model after the first filtering step.

Figure 4.10: Model after the second filtering step.

Figure 4.11: Model after the third filtering step.

4.7 Polygonal Mesh Reconstruction

The polygonal mesh reconstruction step takes the dense patch model that was created and turns it into a closed mesh model. This process is performed in two steps: a mesh reconstruction algorithm is used to turn the point cloud into a closed mesh model. A mesh optimisation stage is used to pull the vertices of the mesh closer to the reconstructed surface of the model.

4.7.1 Mesh Reconstruction

For the mesh reconstruction stage the Poisson Surface Reconstruction algorithm by Kazhdan *et al.* [30] is used to create a closed mesh model from the dense set of patches. The Poisson surface reconstruction algorithm is extensible. The parameters used for the creation of the meshes are shown below:

- Octree depth of 10
- At least γ samples per node (γ is the minimum image requirement from Section 3.3.2.1)
- Patch normals are used to refine vertices

4.7.2 Mesh Optimisation

The mesh optimisation stage aims to move the vertices of the closed mesh, created in Section 4.7.1, closer to the surface of the object. Two implementations of the energy function were written.

The first was as described in Section 3.6.2. This method took between 50 and 150 seconds to execute depending on the number of vertices in the input mesh even when run in parallel on eight cores.

The second method does not generate new patches for each vertex but rather uses the existing patches that were generated by the modified PMVS algorithm. For each vertex $\overline{\mathbf{v}}_i$ a depth map test is used to determine in which images it is visible, these images are added to the set $V(\overline{\mathbf{v}}_i)$. For each image in $V(\overline{\mathbf{v}}_i)$ determine which cell $\overline{\mathbf{v}}_i$ projects into. If this cell is occupied by a patch p add this patch to the set $P'(\overline{\mathbf{v}}_i)$. From this point the algorithm works as described in Section 3.6.2 but optimises over the set of patches $P'(\overline{\mathbf{v}}_i)$. The output of the mesh optimisation stage is shown below in Figure 4.12.

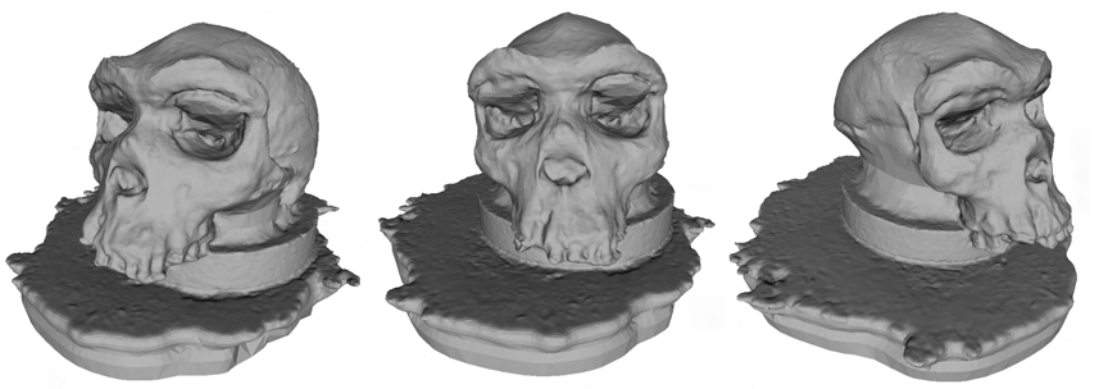

Figure 4.12: Final mesh model of the skull.

4.8 Results

4.8.1 Skull dataset

The skull dataset contains 24 high resolution images with an average resolution of 1950×1750 pixels. For this data set 18 experiments were run. The size of the image cells were set to either 1×1 pixels or 2×2 pixels; the patch size was set to 5×5, 7×7, or 9×9; and the image size was set to quarter-size, half-size, or full size. Sample images are shown for each reconstruction, as well as timing information and the number of patches reconstructed. Each image has a label in Table 4.1 given as follows: LβCγPδ, where β is the image level and two corresponds to quarter-size, one corresponds to half-size, and zero corresponds to

full-size. The size of a square image cell is denoted by γ and δ is the size of the square patch.

The experimental values of β, γ, and δ were determined empirically and each of these effect the quality of the reconstruction. Both the cell size and image size control the density of patches in the reconstructed output. When the size of the image is halved the number of cells is quartered. The values for cell size and image size were determined such that the reconstruction was still subjectively recognisable.

The size of the patch correlates to the observed quality of the reconstruction. When a larger patch is used the images that a patch projects onto need to match closely in a larger area. This results in reconstructions that more closely model the surface. The maximum patch size of 9×9 was chosen due to RAM limitations on the PC running the experiments. Reconstructions using patches smaller than 5×5 did not produce dense reconstructions and were therefore not considered. This dataset is available from Furukawa's website [36].

4.8.1.1 Reconstruction Results

The reconstructions can be roughly grouped based on the three changing parameters, and changing these parameters changes the reconstruction quality significantly. Full size images provide a better final reconstruction than images that have been downsampled, however larger images also take longer to process. The timing results show that the processing time decreases three-fold with a halving of image size (one quarter as many pixels). When the cell size is set to one a patch is reconstructed for every pixel and when it is two a patch is reconstructed for every four pixels. With smaller cell sizes the final reconstruction is more dense. The reconstruction time decreases by approximately three times upon doubling the cell size. The final variable is the size of the patch which was set to either five, seven, or nine. Increasing the patch size by two increases the processing time by 1.5 times as shown in Table 4.1.

Each figure shows three pictures of the same reconstruction from different orientations and each subsection groups reconstructions based on their image size.

The size of the input images alters the number of cells, and thus the reconstruction density of the model, by a factor of n^2. When the size of the image is halved the total number of cells in the image is reduced to a quarter. Further by reducing the size of the image neighbouring pixels are averaged to find a new pixel value. This new pixel value is thus less representative of the object being modeled which lowers the quality of the reconstruction.

The cell size determines the total number of cells in the image where a cell size of 1×1 means there are $h \times v$ cells in an image that are 1×1 pixels in size. When the size of the cell is doubled to 2×2 the total number of cells in the reconstruction is reduced to a quarter. This change effects the density of the output model with smaller cells creating denser reconstructions.

The patch size determines the neighbourhood that is used to check the correlation between patch projections in the input images. Larger patches project onto a larger area in the images. This means that patch projections provide a more accurate representation

of the surface they are modeling because larger patches project onto more pixels.

4.8.1.2 Reconstructions from Full Size Images

The reconstructions from the full size images with cell size one provide high density reconstructions with very minor differences in reconstruction quality over the varying patch sizes. The reconstructions using a patch size of nine (Figure 4.13) and of seven (Figure 4.14) are very similar. The reconstruction with patch size five in Figure 4.15 displays some holes above the eyebrows and below the left eye.

When the size of the cells is set to two the quality of the reconstructions changes noticeably with patch size. The reconstruction with the smallest holes is shown in Figure 4.16. When the patch size is set to seven as shown in Figure 4.17 larger holes are visible above the eyebrows and between the eyes. Finally when the patch size is set to five as shown in Figure 4.18 the model contains many holes over the surface of the reconstruction.

Figure 4.13: Full size images, cell size of one, patch size of nine.

Figure 4.14: Full size images, cell size of one, patch size of seven.

Figure 4.15: Full size images, cell size of one, patch size of five.

Figure 4.16: Full size images, cell size of two, patch size of nine.

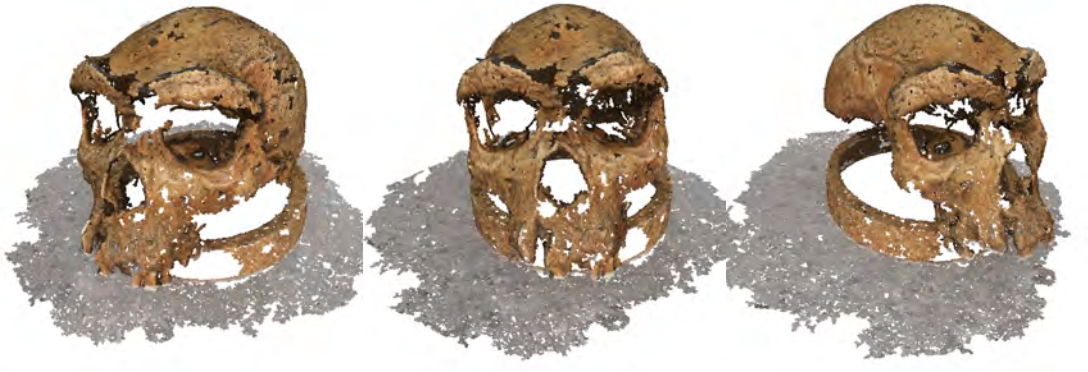

Figure 4.17: Full size images, cell size of two, patch size of seven.

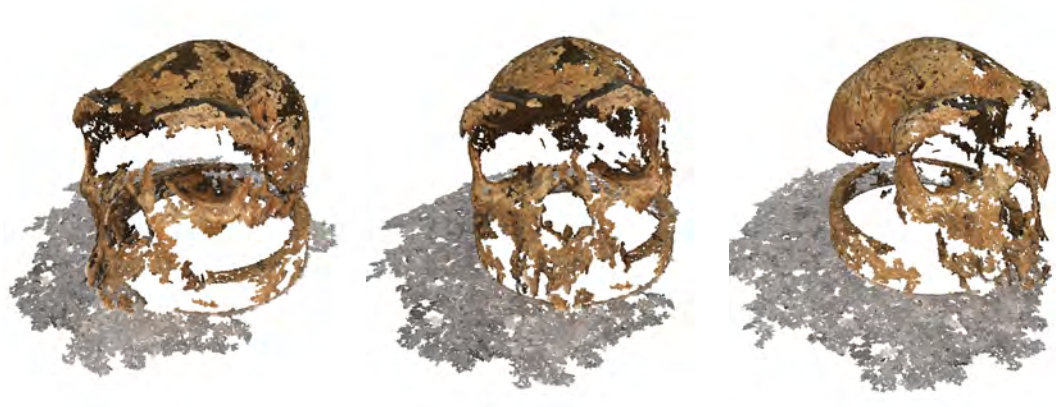

Figure 4.18: Full size images, cell size of two, patch size of five.

4.8.1.3 Reconstructions from Half Size Images

The reconstructions from half size images with cell size one are shown in Figures 4.19, 4.20 and 4.21. The differences between these images are very minor. The reconstruction with patch size five have several small holes distributed over the surface.

When the cell size is set to two the difference between the quality of the reconstructions is more noticeable than with a cell size of one. A patch size of five (Figure 4.24) and seven (Figure 4.23) produce models with a more sparse distribution of patches when compared to a patch size of nine (Figure 4.22).

Figure 4.19: Half size images, cell size of one, patch size of nine.

Figure 4.20: Half size images, cell size of one, patch size of seven.

Figure 4.21: Half size images, cell size of one, patch size of five.

Figure 4.22: Half size images, cell size of two, patch size of nine.

Figure 4.23: Half size images, cell size of two, patch size of seven.

Figure 4.24: Half size images, cell size of two, patch size of five.

4.8.1.4 Reconstructions from Quarter Size Images

The reconstructions from quarter size images with a cell size of one have more noisy patches on top of the skull. The reconstructions are less dense due to the lower number of image cells. The difference between the reconstructions with patch sizes of nine (Figure 4.25) and seven (Figure 4.26) are not significant. When the patch size is reduced to five (Figure 4.27) there are more holes visible on the front of the skull.

Figure 4.25: Quarter size images, cell size of one, patch size of nine.

Figure 4.26: Quarter size images, cell size of one, patch size of seven.

Figure 4.27: Quarter size images, cell size of one, patch size of five.

Figure 4.28: Quarter size images, cell size of two, patch size of nine.

Figure 4.29: Quarter size images, cell size of two, patch size of seven.

Figure 4.30: Quarter size images, cell size of two, patch size of five.

4.8.1.5 Timing Results

Table 4.1 shows the timing results for the skull dataset. Each column shows the timing for one of the major parts of the reconstruction algorithm. From the tables one can see that the time for feature detection and the initial feature matching step remain approximately constant for images of the same size. Full size images typically take ~ 330 seconds. The first expansion step typically takes the most time, a maximum of 68% in L2C1P9 (Figure 4.25), and a minimum of 21% for L0C1P5 (Figure 4.15). The reason the first expansion takes so long is because the initial feature matching step creates a very sparse set of patches, sparse in this case meaning that very few of the image cells contain a patch. This first expansion step tries to create a patch in every cell and since so few cells are populated this amounts to the creation of many patches. The second and third expansion steps take a much shorter time to run because at this stage the number of reconstructed patches is much higher due to the initial expansion. This leads to more cells being occupied and therefore fewer cells are expanded into. Filtering tends to take more time as many patches have been created during expansion. The filters will check all patches to make sure they are consistent with nearby patches causing them to have longer run times.

The total processing time decreases as reconstruction quality is decreased. This makes sense because smaller images have fewer image cells, meaning that fewer patches are created. As the cell size increases the number of cells in the reconstruction decreases meaning that fewer patches are created. The final thing that will decrease processing time is decreasing the size of a patch. The reason this decreases processing time is that there are fewer patch grid points and thus fewer pixel comparisons between images visible to a patch.

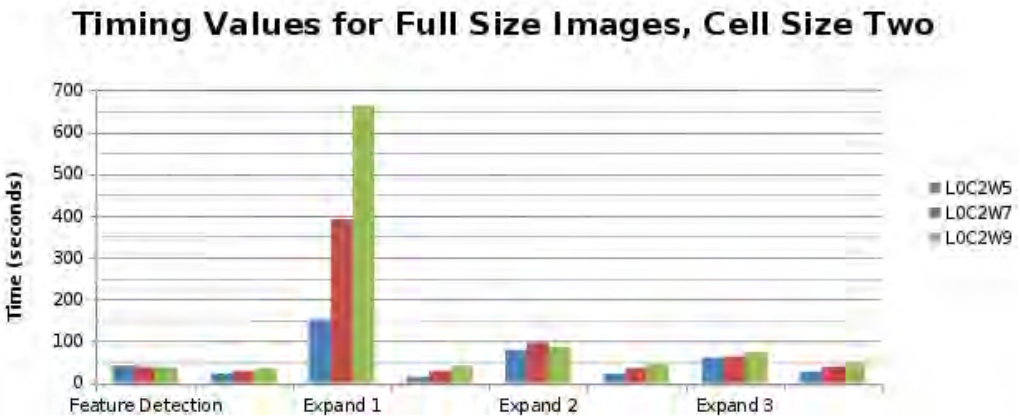

Figure 4.31: Timing graph for full size images with a cell size of two.

ID	Detection (s)	Matching (s)	Expand 1 (s)	Filter 1(s)	Expand 2(s)	Filter 2(s)	Expand 3(s)	Filter 3(s)	Total(s)	Number of Patches
L0C1P9	34.793	38.5568	2977.61	187.412	662.263	216.664	361.123	236.065	4956.0618	5576044
L0C1P7	34.947	33.9941	1938.01	143.776	582.283	178.887	359.743	196.003	3709.0571	5543764
L0C1P5	37.702	33.3517	954.917	93.2487	534.713	135.363	412.905	165.418	2627.7184	5245860
L1C1P9	9.088	7.58714	951.535	48.4225	183.756	53.0888	155.698	56.575	1528.70754	1332465
L1C1P7	9.239	6.80783	625.388	39.9571	139.201	45.7285	73.2985	47.3703	1050.75643	1259472
L1C1P5	10.061	6.29833	366.21	31.1595	89.0603	36.5498	40.4472	38.7909	688.46973	1169604
L2C1P9	2.548	1.99074	290.696	12.5436	38.7997	13.2612	34.0659	13.4354	424.72974	329249
L2C1P7	2.288	1.5676	170.345	10.6053	21.9293	10.7679	19.1527	11.0935	263.5565	304176
L2C1P5	2.306	1.12194	86.2461	7.81705	9.97001	8.4471	6.17696	8.68122	146.62698	268196
L0C2P9	38.915	33.7621	662.773	41.0561	85.5044	44.9883	73.8356	48.4073	1298.1468	1209982
L0C2P7	39.6	28.1422	391.966	28.3913	96.0115	35.6397	63.0811	38.9978	996.0086	1112735
L0C2P5	41.188	22.673	151.556	15.14	78.9546	21.4822	59.7598	26.5631	703.2667	858542
L1C2P9	10.119	5.50704	205.253	10.886	26.2487	11.6659	13.193	12.0171	364.85844	285666
L1C2P7	10.124	4.72869	131.552	8.34577	26.3634	9.36198	16.1769	9.90821	286.54025	278423
L1C2P5	8.819	4.33744	64.7149	6.16343	17.6032	7.14083	5.22437	7.37883	181.9334	241431
L2C2P9	2.277	1.71852	52.4445	2.94124	3.91289	2.97944	2.38285	3.04527	87.42891	71234
L2C2P7	2.278	1.23588	32.2242	2.22858	2.28123	2.30797	1.01073	2.34679	61.69948	64426
L2C2P5	2.315	0.875663	17.5301	1.65231	2.15698	1.80506	0.56491	1.78931	44.526633	59046

Table 4.1: Timing results for skull dataset with CUDA algorithm.

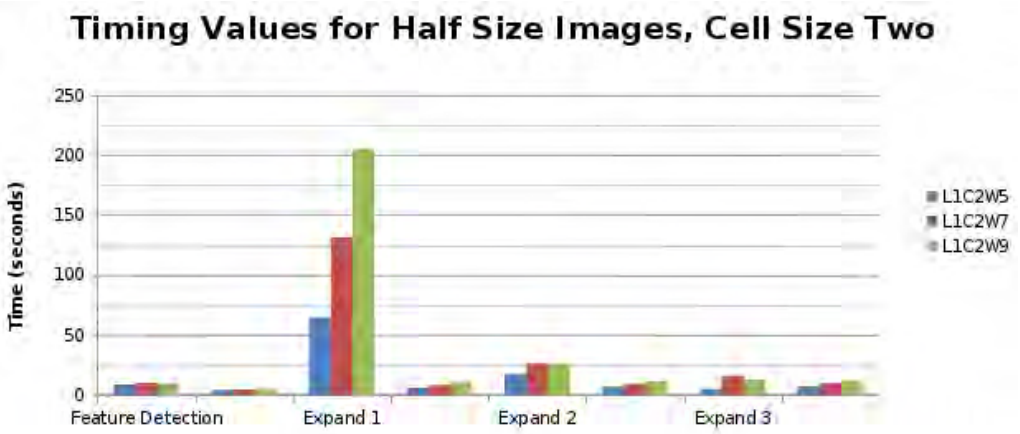

Figure 4.32: Timing graph for half size images with a cell size of two.

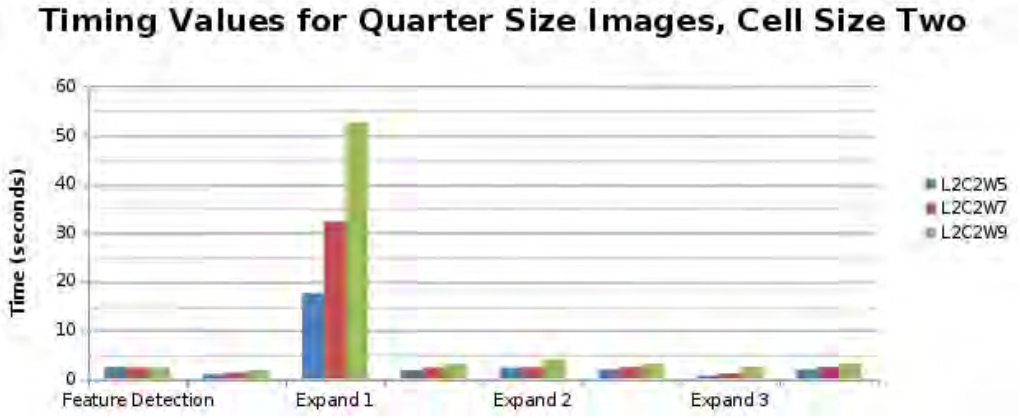

Figure 4.33: Timing graph for quarter size images with a cell size of two.

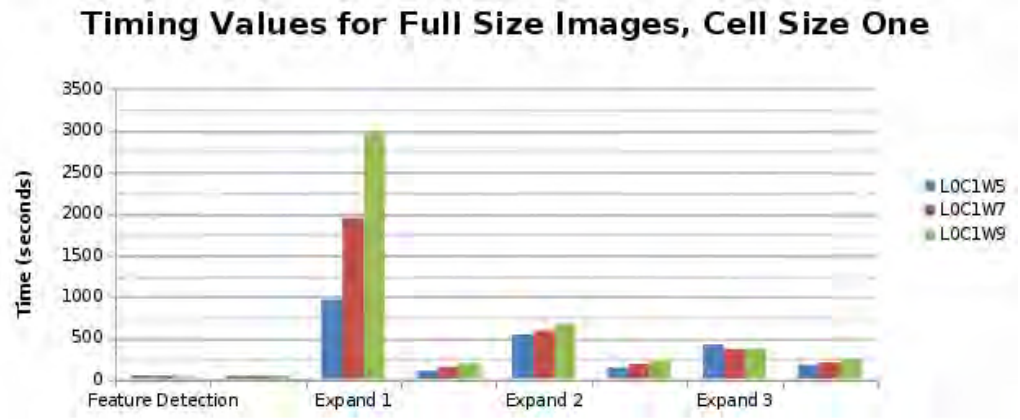

Figure 4.34: Timing graph for full size images with a cell size of one.

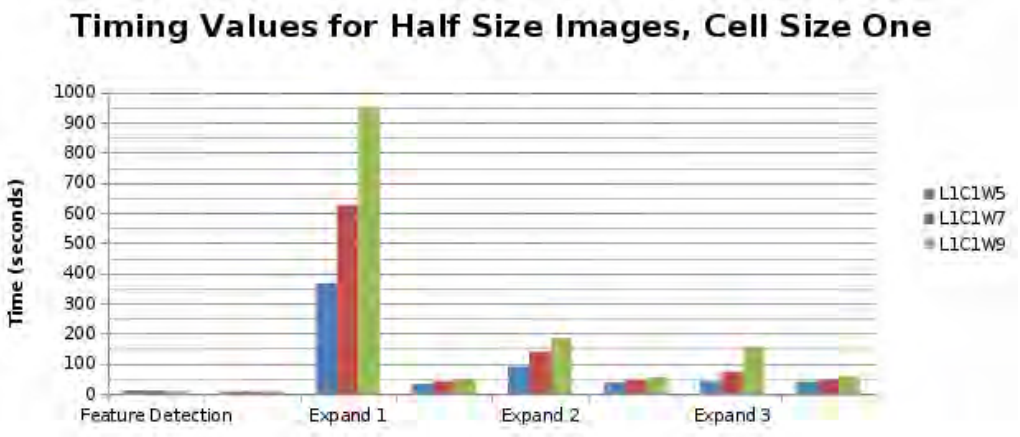

Figure 4.35: Timing graph for half size images with a cell size of two.

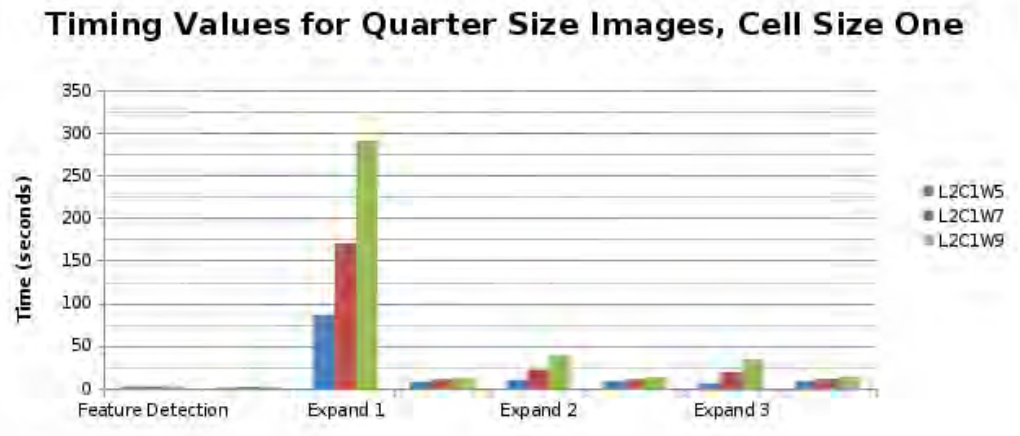

Figure 4.36: Timing graph for quarter size images with a cell size of one.

Figures 4.31 to 4.36 show bar graphs of the amount of time spent on each stage of the processing. From these figures it can be seen that the majority of the total processing time is spent on the first expansion step.

4.8.1.6 Comparison to the Original Algorithm

The original PMVS algorithm was the written by Furukawa [5] and was used to test the performance of the modified algorithm presented in this dissertation. The original program was run on the skull dataset described in Section 4.8.1 using the same configuration files that were used to generate Table 4.1. The timing results of the original algorithm are shown in Table 4.1.

The first modified expansion stage runs slower than the original algorithm when small cell sizes are used: for reconstruction L0C1P9 the modified algorithm takes 2977.61 versus 2842.72 for the original algorithm.

The modified algorithm ran the first expansion stage faster for the datasets with full size images and a cell size of 2×2.

The modified algorithm then performs consistently faster for the subsequent expansion stages.

ID	Detection (s)	Matching (s)	Expand 1 (s)	Filter 1(s)	Expand 2(s)	Filter 2(s)	Expand 3(s)	Filter 3(s)	Total(s)	Number of Patches
L0C1P9	57.407	35.082	2842.72	249.284	849.436	328.59	888.296	-	-	-
L0C1P7	57.079	33.719	2041.71	221.417	839.232	315.86	895.019	-	-	-
L0C1P5	58.141	30.961	936.36	202.74	936.586	-	-	-	-	-
L1C1P9	25.184	5.291	660.405	67.61	157.366	72.345	168.179	75.969	1261.93	1355292
L1C1P7	24.57	3.603	445.965	58.115	107.921	61.629	114.245	65.926	910.695	1326829
L1C1P5	24.8	3.337	296.137	49.894	87.852	55.487	78.526	58.4753	680.752	1294459
L2C1P9	6.254	2.354	198.85	18.647	68.864	19.925	70.854	20.454	417.615	367479
L2C1P7	6.229	1.269	120.384	15.389	41.544	17.166	42.086	17.152	272.605	353711
L2C1P5	6.134	0.713	65.286	12.481	23.858	13.713	23.5107	14.305	170.58	338857
L0C2P9	57.676	21.748	711.867	58.266	130.451	62.693	86.586	64.799	1212.63	1217299
L0C2P7	58.046	20.996	498.354	49.851	120.514	54.479	78.654	56.979	954.246	1178692
L0C2P5	59.02	16.531	329.309	42.885	140.837	48.671	105.889	52.146	810.435	1094757
L1C2P9	24.689	2.448	158.245	15.315	26.673	16.07	17.767	16.571	276.475	322161
L1C2P7	24.705	1.317	104.768	12.492	16.701	13.396	12.63	13.828	198.278	309523
L1C2P5	24.457	1.188	61.735	10.5	11.073	11.232	8.627	11.555	138.22	294770
L2C2P9	6.246	1.598	41.932	4.221	8.985	4.325	6.632	4.46	84.122	86285
L2C2P7	6.212	0.799	26.624	3.386	6.291	3.626	4.495	3.641	60.533	83384
L2C2P5	6.152	0.339	15.312	2.778	3.7	2.815	2.365	2.863	39.641	76177

Table 4.2: Timing results for skull dataset with the original PMVS algorithm.

The approximately 2× speed increase in the feature detection stage is due to the fact that the output of the Gaussian smoothing operation is data parallel and thus each GPU thread can compute the modified value of a single pixel, compared to the CPU version where each new pixel value is calculated sequentially.

The optimisation step does not benefit as much from GPU processing due to the fact that every time that Equation 3.1 is computed several image reads are required to calculate the NCC score. The slowest operation for most processing is memory reading and writing. This problem is dealt with by attempting to do other processing tasks while the memory is being loaded. The computation of the photometric discrepancy is fairly inexpensive in terms of processing and due to this the memory operations are the bottleneck.

The number of patches reconstructed by the original and the new algorithm are very similar. This is due to the fact that the algorithm tries to fill every single image cell that contains the object or scene. The feature detection step is used to seed the creation of patches, so provided that correct features are detected the following expansion and filtering steps are able to create a sufficiently dense collection of patches.

4.8.1.7 Discussion

Table 4.1 shows the output information gathered from the running of the experiment. In the Section 4.1 it was stated that the number of patches reconstructed related directly to both image and patch size. When the image size is halved the number of cells reduced to a quarter. Similarly when the cell size is doubled the total number of cells is reduced to a quarter. Table 4.3 shows the average number of patches reconstructed for each pair of image and cell sizes and Table 4.4 shows the ratio of patches created for each pair of image and cell sizes.

Pair	Average Reconstructed Patches
L0C1	5455223
L1C1	1253847
L2C1	300540
L0C2	1060420
L1C2	268507
L2C2	64902

Table 4.3: Average number of patches reconstructed.

	L0C1	L1C1	L2C1	L0C2	L1C2	L2C2
L0C1	1	4.35	18.15	5.14	20.32	84.05
L1C1		1	4.17	1.18	4.67	19.32
L2C1			1	0.28	1.12	4.63
L0C2				1	3.95	16.34
L1C2					1	4.13
L2C2						1

Table 4.4: Ratio of the number of reconstructed patches .

Table 4.4 shows that the expected relationships between cell size and image size hold in

reality. Halving the size of the input images or doubling the image size cause the number of patches that are reconstructed to be quartered. This can be seen when comparing L0C1 to L1C1 with a ratio of 4.35, meaning that there are 4.35 times more patches in L0C1 than in L1C1. L0C1 has 5.14 times as many patches as L0C2. These values are close to the expected value of four times more patches for full sized images, or for one pixel size cells.

When examining the timing results along with the number of patches reconstructed it can be seen that a patch size of nine, when compared to a patch size of seven, only creates 0.58% more patches but takes 25% longer for the L0C1 pair. For the L1C1 pair a cell size of nine creates 5.48% more patches and takes 31% longer than a patch size of seven. Similarly for the L2C1 pair, a patch size of nine creates 7.62% more patches but takes 38% longer. Similar values are found when looking at the results when a cell size of two is used. From this data and inspection of the reconstruction results there is not a clear global advantage to using a patch size of nine over a patch size of seven.

The first expansion step is the most expensive set of computations performed during the run time of the program. The execution time of the expansion steps increases with patch size, image size, and smaller cells.

4.8.2 Middlebury Dinosaur Dataset

The Middlebury dinosaur dataset was discussed in Section 2.1.6. The captured images of the dinosaur were split into three datasets with varying numbers of image. The modified algorithm discussed in this thesis was used to generate a closed mesh model for each of the three datasets.

The meshing algorithm used for the dinosaur in this dissertation differs slightly from the one that was used by Furukawa and Ponce. They used Furukawa's iterative snapping algorithm [31, 1] to create the mesh models for these datasets. This implementation uses the Poisson surface reconstruction method that cannot make use of foreground and background information. After the closed mesh has been created from the oriented patches. The mesh optimisation algorithm that was discussed in Section 3.6.2 was used to pull the mesh vertices closer to the original reconstructed patches.

4.8.2.1 Dinosaur Sparse Ring

The sparse ring dataset contains 16 images of the dinosaur. The reconstruction was performed using full size images, cell size of 2×2, and a patch size of 9×9. The point cloud model is shown in Figure 4.37 and the mesh model is shown in Figure 4.38. The reconstruction from the sparse ring is the most complete according to the Middlebury evaluation (Table 4.5) The patches are more sparse around the base of the dinosaur and on its right shoulder (Middle image of Figure 4.37). The reconstructed mesh smooths over these sparse areas but had to estimate the surface where there was no patch data.

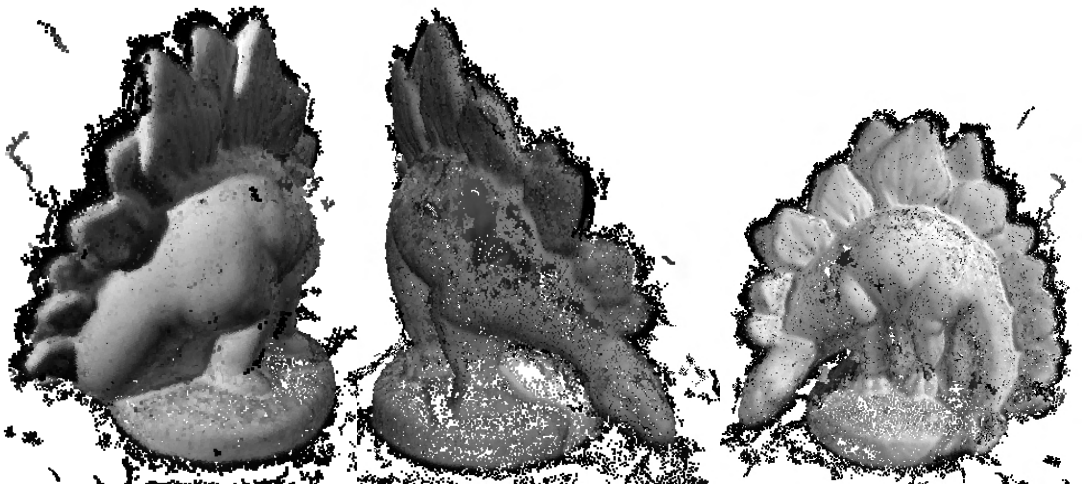

Figure 4.37: Point cloud reconstruction of the dinosaur sparse ring dataset.

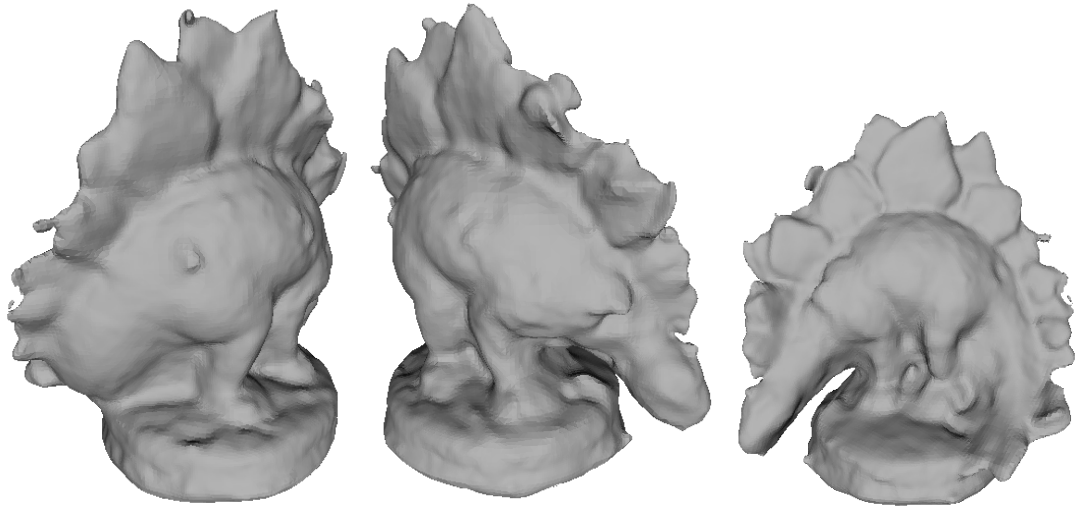

Figure 4.38: Mesh reconstruction of the dinosaur sparse ring dataset.

The reconstruction of the dinosaur from the sparse ring has a large hole on the right shoulder of the dinosaur. There are many noisy patches along the edge of its spines.

4.8.2.2 Dinosaur Ring

The ring dataset contains 48 images of the dinosaur. The reconstruction was performed using full size images, cell size of 2×2, and a patch size of 9×9. The point cloud model is shown in Figure 4.39 and the mesh model is shown in Figure 4.40. The patch reconstruction of the dinosaur ring dataset was the most accurate according to the Middlebury evaluation (Table 4.5). The reconstructed patches are very sparse over the right shoulder (Middle image of Figure 4.39). These sparse patches and a few outlying patches that were not removed caused the mesh model to balloon out at the right shoulder.

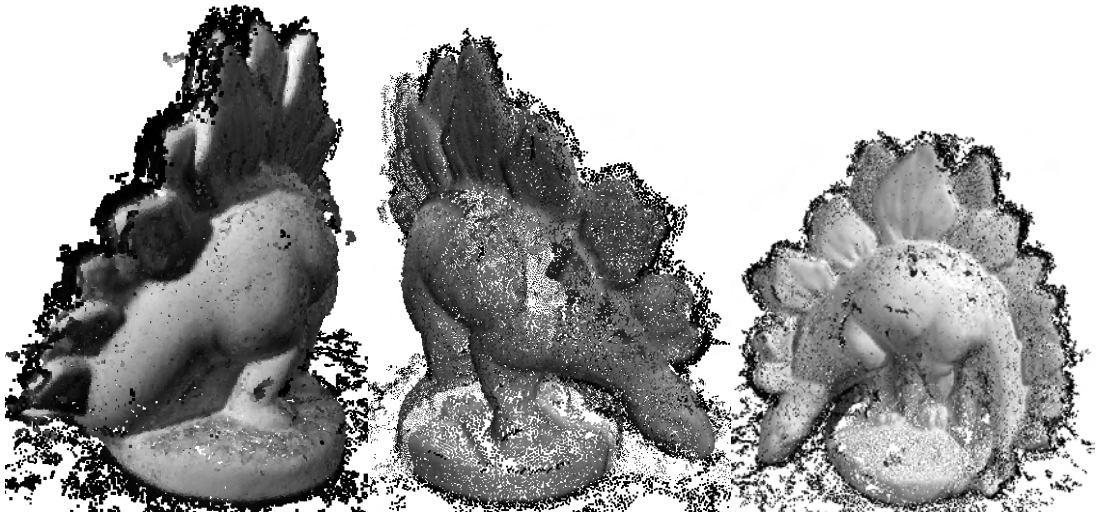

Figure 4.39: Point cloud reconstruction of the dinosaur ring dataset.

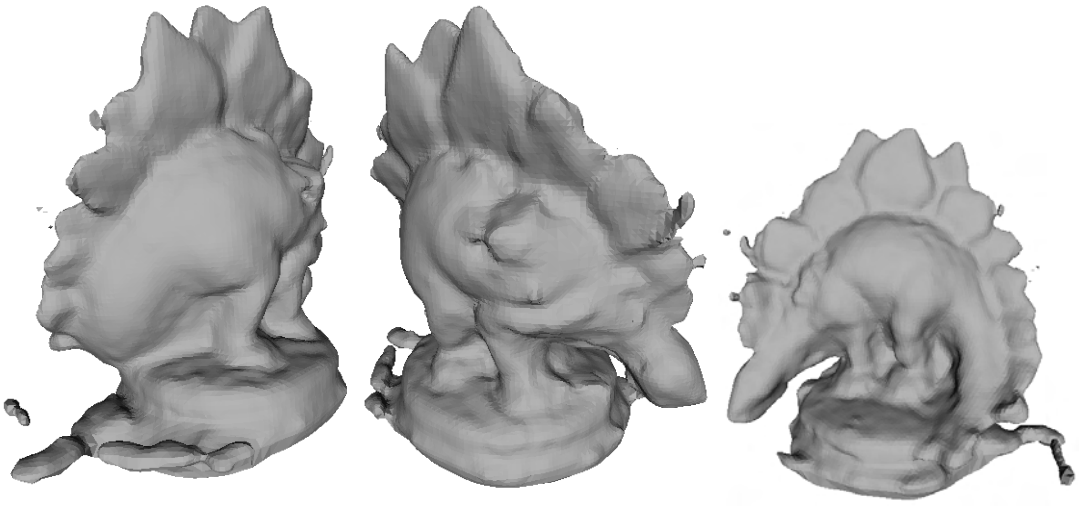

Figure 4.40: Mesh reconstruction of the dinosaur ring dataset.

4.8.2.3 Dinosaur Hemisphere

The hemisphere dataset contains 363 images in a hemisphere around the dinosaur. The reconstruction was performed using half size images, cell size of 2×2, and a patch size of 9×9. The point cloud model is shown in Figure 4.42 and the mesh model is shown in Figure 4.42. The reconstruction from the full hemisphere dataset provides the least accurate and complete reconstruction as seen in Table 4.5. The entire left side of the dinosaur is sparsely reconstructed (Right image of Figure 4.41). Because of these sparse patches the mesh reconstruction of the right hand side is inaccurate.

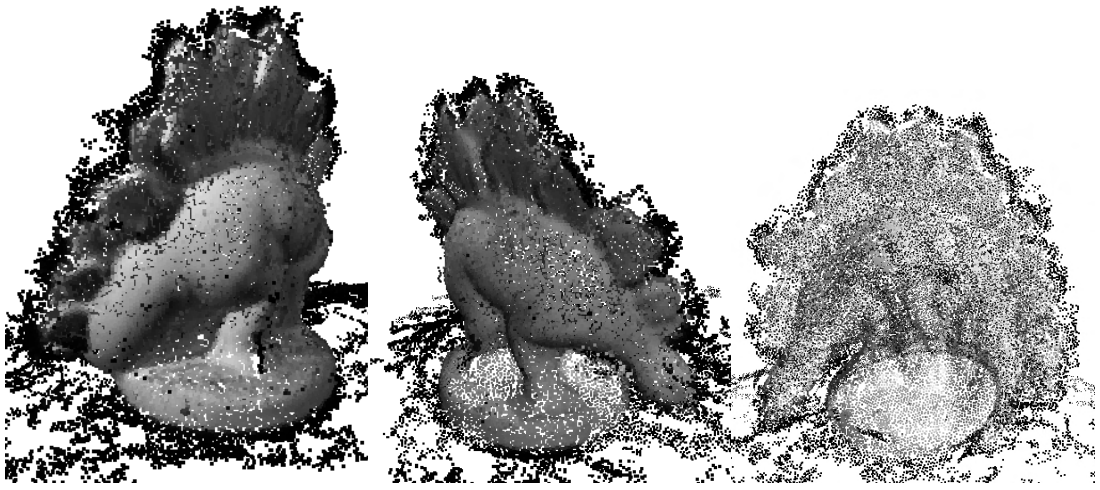

Figure 4.41: Point cloud reconstruction of the dinosaur hemisphere dataset.

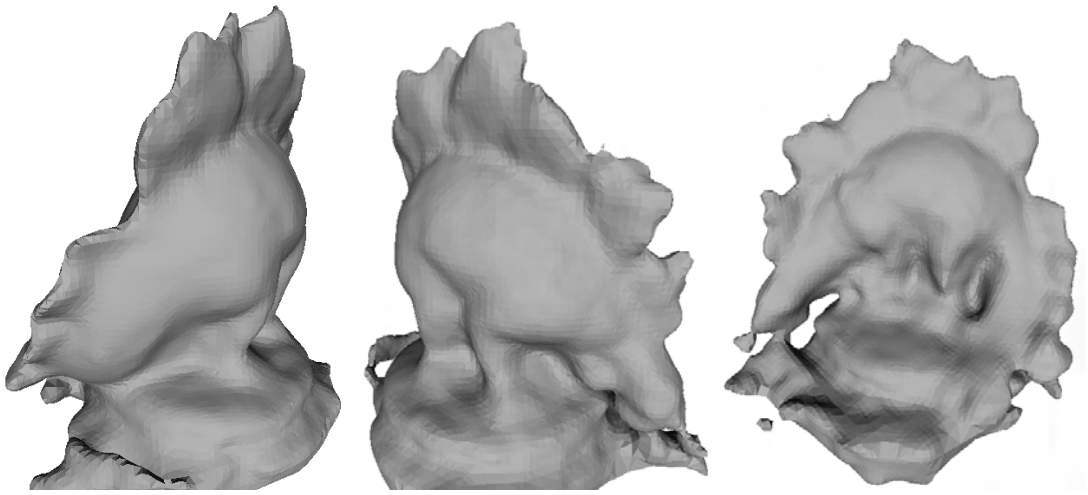

Figure 4.42: Mesh reconstruction of the dinosaur hemisphere dataset.

4.8.2.4 Results of the Middlebury Evaluation

The results of Middlebury evaluation of the dinosaur reconstruction are shown in Table 4.5. The table shows that the sparse ring and ring datasets provide the best reconstructions of the dinosaur. These reconstructions compare well to the original algorithm in both accuracy and completeness. The full hemisphere reconstruction did not provide an accurate reconstruction and this can be seen in Table 4.5. The completeness is poor at 89.4% compared to the original algorithm at 99.8%.

	Modified Algorithm		**Original Algorithm** [1]	
Metric	**Accuracy**	**Completeness**	**Accuracy**	**Completeness**
Sparse Ring	0.44 mm	98.8%	0.37 mm	99.2%
Ring	0.43 mm	98.4%	0.28 mm	99.8%
Hemisphere	0.89 mm	89.4%	0.33 mm	99.8%

Table 4.5: Results of the modified and original reconstruction algorithms for the Middlebury dinosaur datasets.

Chapter 5

Conclusion

5.1 Summary

This dissertation shows the work that was done to improve upon the existing PMVS-2 algorithm. A CUDA version of the feature detector was implemented and it was seen to run 10 times faster than the original algorithm for all image sizes. The numeric optimisation algorithm was replaced with the CUDA based L-BFGS. This change improved the execution time of the initial matching step.

The primary aim of the work was to construct a dense cloud of oriented patches that closely model the surface. A mesh creation and optimisation step were added that turned the 3D point cloud into a closed mesh model.

The final models of the skull show a dense collection of patches that model the surface. The most dense reconstruction was shown in Figure 4.13. This image shows a reconstruction containing 5576044 patches that were created in 4956.0618 seconds using the modified algorithm. The same reconstruction with the original algorithm ran out of memory and did not produced an output model. The fastest reconstruction can be seen in Figure 4.30 with 59046 patches created in 44.526633 seconds using the modified algorithm. The original algorithm performed the same reconstruction in 39.641 seconds and created 76177 patches.

The feature detector was changed such that the common part of the algorithm, Gaussian smoothing, would run on the GPU. The algorithm was a texture memory based implementation of the separable Gaussian kernel operator. This algorithm ran faster than the original CPU based sequential algorithm with the performance increase related to image size and magnitude of σ. For $\sigma = 4.0$ as used in Figure 2.5 the GPU algorithm performed approximately 10 times faster than the CPU implementation. For the reconstructions of the skull dataset the feature detection stage ran approximately two times faster due to the smaller value of $\sigma \approx 1.0$.

The initial matching and expansion steps both made use of the patch optimisation algorithm described in Section 3.2.4. The optimisation algorithm was changed to a CUDA version of the L-BFGS algorithm. The result of this change to the optimisation stage made a small difference.

The modified algorithm performed faster on the first expansion stage with full size

images and a cell size of 2×2. The modified algorithm performed the first expansion slower than the original algorithm on the other datasets. The modified algorithm performed faster than the original algorithm on the subsequent expansion steps.

The modified algorithm performed well in the Middlebury evaluation on the dinosaur dataset. The ring and sparse ring provided the best accuracy and completeness respectively. The full hemisphere reconstruction was not very accurate or complete due to the sparseness of the reconstructed patches. This reconstruction could be improved by running the algorithm again with different parameters until a better solution is found.

These results show that it is indeed feasible to use GPUs and more specifically CUDA to improve the execution time of some portions of the PMVS algorithm. Specifically parts that are highly data parallel such as feature detection. CUDA does not provide any real improvement to the expansion stage with its GPU-based optimisation step. The reason that the subsequent expansion steps ran faster on the modified algorithm is because fewer patches were generated when using the GPU-based optimisation.

5.2 Future Work

This dissertation discussed work that turned a series of calibrated images into a 3D mesh model. The primary extensions to make would be to allow uncalibrated images to be used as an input. This is possible using structure from motion to determine camera parameters from the images themselves [2, 9].

It is also possible to allow for larger sets of input images by clustering like views and generating sub-models from these using any MVS technique which are then grouped into a final model. Furukawa *et. al.* described and implemented this idea in "Towards Internet-Scale Multi-View Stereo" [2, 37].

More research could be done on using different CUDA optimisation methods to determine which method works best for patch optimisation. Research should be conducted on other types of 3D reconstruction algorithms to determine their suitability for CUDA optimisation.

Porting of the expansion and filtering stages onto a CUDA or generic GPGPU platform should be possible. The expansion step attempts to create new patches in empty cells from existing nearby patches. This can be run independently since it would not matter if multiple new patches were created in a new cell since erroneous patches are removed by the filters.

Bibliography

[1] Y. Furukawa and J. Ponce, "Accurate, dense, and robust multiview stereopsis," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 32, pp. 1362 –1376, August 2010.

[2] Y. Furukawa, B. Curless, S. Seitz, and R. Szeliski, "Towards internet-scale multi-view stereo," in *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*, pp. 1434 –1441, June 2010.

[3] S. Seitz, B. Curless, J. Diebel, D. Scharstein, and R. Szeliski, "A comparison and evaluation of multi-view stereo reconstruction algorithms," in *Computer Vision and Pattern Recognition, 2006 IEEE Computer Society Conference on*, vol. 1, pp. 519 – 528, 2006.

[4] D. Kirk and W. Hwu, *Programming Massively Parallel Processors*. Morgan Kaufman, 1st ed., 2010.

[5] Y. Furukawa, "Patch-based multi-view stereo software." http://grail.cs.washington.edu/software/pmvs/, 2010.

[6] N. D. Campbell, G. Vogiatzis, C. Hernández, and R. Cipolla, "Using multiple hypotheses to improve depth-maps for multi-view stereo," in *Proceedings of the 10th European Conference on Computer Vision: Part I*, ECCV '08, (Berlin, Heidelberg), pp. 766–779, Springer-Verlag, 2008.

[7] C. Zach, "Fast and high quality fusion of depth maps," in *3D Data Processing, Visualization and Transmission*, 2008.

[8] A. Laurentini, "The visual hull concept for silhouette-based image understanding," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 16, pp. 150 –162, Feb. 1994.

[9] S. Agarwal, N. Snavely, I. Simon, S. Seitz, and R. Szeliski, "Building Rome in a day," in *Computer Vision, 2009 IEEE 12th International Conference on*, pp. 72 –79, Oct. 29 2009.

[10] Y. Furukawa and J. Ponce, "Accurate camera calibration from multi-view stereo and bundle adjustment," in *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on*, pp. 1 –8, June 2008.

[11] G. Vogiatzis, P. Torr, and R. Cipolla, "Multi-view stereo via volumetric graph-cuts," in *Computer Vision and Pattern Recognition*, pp. 391–398, 2005.

[12] G. Vogiatzis, C. Hernandez, P. Torr, and R. Cipolla, "Multiview stereo via volumetric graph-cuts and occlusion robust photo-consistency," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 29, pp. 2241 –2246, dec. 2007.

[13] C. Hernandez and F. Schmitt, "Silhouette and stereo fusion for object modeling," *Computer Vision and Image Understanding*, vol. 96(3), pp. 367–392, 2004.

[14] S. Seitz, B. Curless, J. Diebel, D. Scharstein, and R. Szeliski, "Multi-view stereo," *http://vision.middlebury.edu/mview/*, 2009.

[15] D. Bradley, T. Boubekeur, and W. Heidrich, "Accurate multi-view reconstruction using robust binocular stereo and surface meshing," in *Computer Vision and Pattern Recognition*, 2008.

[16] Y. Furukawa and J. Ponce, "High-fidelity image-based modeling," tech. rep., University of Illinois at Urbana-Champaign, 2006.

[17] M. Goesele, B. Curless, and S. Seitz, "Multi-view stereo revisited," in *Computer Vision and Pattern Recognition*, 2006.

[18] V. Kolmogorov and R. Zabih, "Multi-camera scene reconstruction via graph cuts," in *Proceedings of the 10th European Conference on Computer Vision*, vol. III, pp. 82–96, 2002.

[19] J. Pons, R. Keriven, and O. Faugeras, "Modelling dynamic scenes by registering multi-view image sequences," in *Computer Vision and Pattern Recognition*, vol. II, pp. 822–827, 2005.

[20] R. Hartley and A. Zisserman, *Multiple View Geometry in Computer Vision*. Cambridge University Press, 2nd ed., 2003.

[21] B. Cyganek and J. P. Siebert, *An Introduction to 3D Computer Vision Techniques and Algorithms*. Wiley, 2009.

[22] R. Hartley and P. Sturm, "Triangulation," tech. rep., 1997.

[23] D. C. Liu and J. Nocedal, "On the limited memory BFGS method for large scale optimization," *Mathematical programming*, vol. 45, no. 1-3, pp. 503–528, 1989.

[24] Y. Fei, G. Rong, B. Wang, and W. Wang, "Parallel L-BFGS-B algorithm on GPU," *Computers and Graphics*, vol. 40, no. 0, pp. 1–9, 2014.

[25] O. Bretscher, *Linear Algebra with Applications*. Upper Saddle River: Prentice Hall, 3rd ed., 1995.

[26] B. Barney and L. L. N. Laboratory, "POSIX threads programming." https://computing.llnl.gov/tutorials/pthreads/, December 2013.

[27] G. Stockman and L. G. Shapiro, *Computer Vision*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1st ed., 2001.

[28] C. Harris and M. Stephens, "A combined corner and edge detector," in *In Proc. of Fourth Alvey Vision Conference*, pp. 147–151, 1988.

[29] D. Marr and E. Hildreth, "Theory of Edge Detection," *Proceedings of the Royal Society of London. Series B, Biological Sciences*, vol. 207, no. 1167, pp. 187–217, 1980.

[30] M. Kazhdan, M. Bolitho, and H. Hoppe, "Poisson surface reconstruction," in *Proceedings of the fourth Eurographics symposium on Geometry processing*, SGP '06, (Aire-la-Ville, Switzerland, Switzerland), pp. 61–70, Eurographics Association, 2006.

[31] Y. Furukawa and J. Ponce, "Carved visual hulls for image-based modelling," *International Journal of Computer Vision*, vol. 81, pp. 53–67, March 2009.

[32] N. Okazaki, "libLBFGS: a library of limited-memory Broyden-Fletcher-Goldfarb-Shanno (L-BFGS)." Internet: http://www.chokkan.org/software/liblbfgs/, December 2010.

[33] B. Karlsson, "Smart pointers in boost." C/C++ Users Journal, April 2002.

[34] G. Bradski, "OpenCV," *Dr. Dobb's Journal of Software Tools*, 2000.

[35] K. G. Derpanis, "Outline of the relationship between the difference-of-Gaussian and Laplacian-of-Gaussian," tech. rep., York University, September 2006.

[36] Y. Furukawa and J. Ponce, "3d photography dataset." http://homes.cs.washington.edu/ furukawa/research/mview/index.html, May 2006.

[37] Y. Furukawa, "Clustering views for multi-view stereo (CMVS) software." http://www.di.ens.fr/cmvs/, July 2010.

Appendix A

Derivations

A.1 Separability of Gaussian Kernels

Given a two-dimensional Gaussian kernel $g(x,y)$ and an image $f(x,y)$ the convolution of the kernel with the image is:

$$\begin{aligned} g(x,y) * f(x,y) &= \sum_{k=1}^{m}\sum_{l=1}^{n} g(k,l)\, f(x-k, y-l) \\ &\sum_{k=1}^{m}\sum_{l=1}^{n} e^{-\frac{k^2+l^2}{2\sigma^2}} f(x-k, y-l) \\ &\sum_{k=1}^{m} e^{-\frac{k^2}{2\sigma^2}} \left\{ \sum_{l=1}^{n} e^{-\frac{l^2}{2\sigma^2}} f(x-k, y-l) \right\} \end{aligned} \tag{A.1}$$

From this it can be seen that convolving the image with a two-dimensional Gaussian kernel is the same as convolving the image first with one of the kernels, then convolving the result with the other kernel, the order of convolution is unimportant because the the linear convolution operator is commutative and associative.

A.2 Full Response of the Harris Filter

$$H = \det(M) - \lambda \mathrm{trace}^2(M) \tag{A.2}$$

$$M = G_{\sigma_1} * \left(\nabla I \nabla I^T\right) \tag{A.3}$$

$$\nabla I = \begin{bmatrix} \frac{\partial I}{\partial x} & \frac{\partial I}{\partial y} \end{bmatrix}^T \tag{A.4}$$

where:

$$I_x = \frac{\partial I}{\partial x},$$

$$I_y = \frac{\partial I}{\partial y},$$

$$
\begin{aligned}
M &= G_{\sigma_1} * \left(\begin{bmatrix} I_x \\ I_y \end{bmatrix} \begin{bmatrix} I_x & I_y \end{bmatrix} \right) \\
&= G_{\sigma_1} * \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix}
\end{aligned}
$$

$$
H = \det\left(G_{\sigma_1} * \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix} \right) - \lambda \mathrm{trace}^2 \left(G_{\sigma_1} * \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix} \right) \tag{A.5}
$$

A.3 Finding the Fundamental Matrix

Given two camera matrices P and P'. The fundamental matrix F is found according to

$$
F = \begin{bmatrix}
\det \begin{bmatrix} P^1 \\ P^2 \\ P'^1 \\ P'^2 \end{bmatrix} & \det \begin{bmatrix} P^1 \\ P^2 \\ P'^2 \\ P'^0 \end{bmatrix} & \det \begin{bmatrix} P^1 \\ P^2 \\ P'^0 \\ P'^1 \end{bmatrix} \\
\det \begin{bmatrix} P^2 \\ P^0 \\ P'^1 \\ P'^2 \end{bmatrix} & \det \begin{bmatrix} P^2 \\ P^0 \\ P'^2 \\ P'^0 \end{bmatrix} & \det \begin{bmatrix} P^2 \\ P^0 \\ P'^0 \\ P'^1 \end{bmatrix} \\
\det \begin{bmatrix} P^0 \\ P^1 \\ P'^1 \\ P'^2 \end{bmatrix} & \det \begin{bmatrix} P^0 \\ P^1 \\ P'^2 \\ P'^0 \end{bmatrix} & \det \begin{bmatrix} P^0 \\ P^1 \\ P'^0 \\ P'^1 \end{bmatrix}
\end{bmatrix} \tag{A.6}
$$

where:

p^j = the j^{th} row of P, and

$\begin{bmatrix} p^{j_1} \\ p^{j_2} \\ p'^{j_1} \\ p'^{j_2} \end{bmatrix}$ = a 4×4 matrix.

Appendix B

Reconstruction Results

B.1 Roman Action Figure

The roman dataset contained 48 high resolution images of an action figure of a roman soldier as well as the corresponding extrinsic and intrinsic parameters. The parameters that were used on this dataset are the same as the skull dataset (Section 4.8.1). However due to memory requirements on the experimental PC full size images with 1×1 cells could not be tested. This dataset is available from Furukawa's website [36].

B.1.1 Reconstructions from Full Size Images

Figures B.1, B.2, and B.3 show the results of the new reconstruction algorithm on the roman dataset. The reconstructions from full size images using a patch size of 9×9, shown in Figure B.1, provides the best reconstruction of the soldier however the reconstruction took 74 minutes. From the images and Table B.1 it can be seen that the number of reconstructed patches decrease as patch size decreases.

Figure B.1: Full size images, cell size of two, patch size of nine.

Figure B.2: Full size images, cell size of two, patch size of seven.

Figure B.3: Full size images, cell size of two, patch size of five.

B.1.2 Reconstructions from Half Size Images

Figures B.4, B.5, and B.6 show reconstructions of the roman action figure using a cell size of 1×1 pixels. These reconstructions have dense coverage of the object and take between 28 and 72 minutes to be computed. Once again the most dense reconstruction is created with a patch size of 9×9 but the difference between the three reconstructions is small.

Figures B.7, B.8, and B.9 show reconstructions of the roman action figure with a cell size of 2×2 pixels. These reconstructions took between nine and 19 minutes to compute. Patch sizes of 9×9 and 7×7 create dense reconstructions with similar numbers of patches. The reconstruction using 5×5 patches was much less dense.

Figure B.4: Half size images, cell size of one, patch size of nine.

Figure B.5: Half size images, cell size of one, patch size of seven.

Figure B.6: Half size images, cell size of one, patch size of five.

Figure B.7: Half size images, cell size of two, patch size of nine.

Figure B.8: Half size images, cell size of two, patch size of seven.

Figure B.9: Half size images, cell size of two, patch size of five.

B.1.3 Reconstructions from Quarter Size Images

Figures B.10, B.11, and B.12 show reconstruction using quarter size images with a cell size of 1×1 pixels. The most patches were reconstructed when the patch size was 9×9 and took 16 minutes to compute.

Figures B.13, B.14, and B.15 show reconstruction using quarter size images with a cell size of 2×2 pixels. These reconstructions took between two and four minutes to compute.

Figure B.10: Quarter size images, cell size of one, patch size of nine.

Figure B.11: Quarter size images, cell size of one, patch size of seven.

Figure B.12: Quarter size images, cell size of one, patch size of five.

Figure B.13: Quarter size images, cell size of two, patch size of nine.

Figure B.14: Quarter size images, cell size of two, patch size of seven.

Figure B.15: Quarter size images, cell size of two, patch size of five.

B.1.4 Timing Results

Table B.1 shows the timing results for the reconstructions of the roman dataset. Results could not be captured for full sized images, with a cell size of 1×1 pixels due to memory limitations on the PC running the experiments.

ID	Detection (s)	Matching (s)	Expand 1 (s)	Filter 1(s)	Expand 2(s)	Filter 2(s)	Expand 3(s)	Filter 3(s)	Total(s)	Number of Patches
L0C1P9	162.459	-	-	-	-	-	-	-	-	-
L0C1P7	160.849	-	-	-	-	-	-	-	-	-
L0C1P5	161.412	-	-	-	-	-	-	-	-	-
L1C1P9	36.161	35.2215	2498.39	121.482	507.223	135.055	581.78	151.164	4316.2555	2360536
L1C1P7	36.639	37.6334	1470.54	97.0055	344.346	111.299	327.769	122.853	2801.3889	2133122
L1C1P5	38.544	33.5088	673.506	69.9898	215.184	84.0836	176.027	93.656	1651.8532	1771500
L2C1P9	9.646	11.3342	544.461	28.8227	134.136	31.8214	114.086	34.2942	974.8972	639801
L2C1P7	9.756	8.25902	304.1	22.0622	85.4569	25.0604	60.1859	26.771	608.38072	537049
L2C1P5	8.823	5.98642	120.581	15.3829	49.3863	18.2487	43.2755	20.6776	343.16612	436063
L0C2P9	160.684	173.018	1954.93	90.1433	476.038	103.124	288.434	111.834	4472.7913	1786239
L0C2P7	162.345	139.154	885.71	57.4701	265.888	67.2907	144.039	73.6022	2922.124	1386768
L0C2P5	163.197	111.466	344.889	37.4765	103.595	41.9679	74.9108	46.7083	2056.9235	1036441
L1C2P9	38.617	37.3248	548.586	25.0379	117.636	28.5818	41.5441	28.9226	1133.5662	494905
L1C2P7	39.188	26.9211	292.839	18.1346	83.6878	21.434	41.731	23.3546	817.9881	438066
L1C2P5	39.950	23.6269	122.618	12.2861	31.3192	13.8644	15.8701	14.4131	550.4118	318453
L2C2P9	8.951	9.42004	110.401	5.74502	21.1102	6.10041	7.79461	6.26891	237.67199	121783
L2C2P7	9.0	6.25954	49.2206	3.96826	12.1482	4.41996	5.18813	4.55338	156.21267	99820
L2C2P5	8.635	4.84123	22.7433	2.75011	6.6714	3.02412	4.60414	3.39137	116.61447	83954

Table B.1: Timing results for roman dataset.

Appendix C

DVD

The DVD contains all of the source code for the modified PMVS-2 algorithm and the meshlab models that were generated by the algorithm. It also contains some sample reconstructions of the intermediate stages of the algorithm.