

**An Investigation into the
Recovery of Three-Dimensional
Structure from Two-Dimensional
Images**

Bruce J. Lamond

Master of Science
School of Computer Science
Division of Informatics
University of Edinburgh
2002

Abstract

This thesis details the implementation and investigation of an algorithm to recover models in 3D from sets of 2D images. Recent attention in this field has switched from investigations into static scenes to dynamic reconstruction of scenes, for applications in creative media and the biomedical sectors, with particular attention on real time performance. The technique by which model recovery is achieved here is the shape-from-silhouette method. Past work has focussed largely on the voxel based approach to specifying the visual hull, where this work builds an alternative representation by recovering a polyhedral version of the hull. The implementation creates model scenes in a 3D environment, the images from which are contoured to extract a silhouette. The main pipeline then follows on from work in (17) by constructing optimised data structures which allow the intersections required to recover the visual hull to be done in 2D. Contrary to the voxel based methods which tend to progressively carve away regions of space where the object does not lie, the polyhedral method rather retains information about where the object is in terms of calculated intersections. Despite some problems with creating a robust implementation, some precision is obtained in the recovery of model structures.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(Bruce J. Lamond)

Table of Contents

| | | |
|----------|---|-----------|
| 1 | Introduction. | 1 |
| 2 | Background | 3 |
| 2.1 | Coordinate systems. | 3 |
| 2.2 | Epipolar geometry. | 6 |
| 2.3 | Previous and Related Work. | 9 |
| 3 | Method | 15 |
| 3.0.1 | Model Scene Specification. | 15 |
| 3.0.2 | Camera Calibration | 18 |
| 3.0.3 | The Fundamental Matrix. | 22 |
| 4 | Implementation | 24 |
| 4.1 | Image Processing. | 24 |
| 4.2 | Polyhedral Visual Hull algorithm. | 30 |
| 4.2.1 | Optimised Data Structures. | 32 |
| 4.2.2 | Intersection of Cones. | 36 |
| 4.3 | Calculating Visual Hull Faces. | 40 |
| 4.3.1 | Recovery to 3D. | 40 |
| 4.3.2 | Intersection in 3D. | 42 |
| 4.4 | Program Structure. | 42 |
| 4.4.1 | Description of Classes. | 43 |
| 5 | Results. | 53 |
| 5.1 | Model Scene and Calibration. | 53 |

| | | |
|----------|-------------------------------------|-----------|
| 5.2 | Image Processing. | 54 |
| 5.3 | Fundamental Matrix. | 54 |
| 5.4 | PVH Algorithm. | 54 |
| 6 | Discussion and Further Work. | 61 |
| 6.1 | Discussion. | 61 |
| 6.2 | Further Work. | 63 |
| | Bibliography | 64 |

Chapter 1

Introduction.

This thesis details work undertaken in order to recover and evaluate a three-dimensional surface model representation of an arbitrary object, imaged by a number of static and calibrated cameras. The study of full-scale real time systems with this goal is relatively new and has been gaining momentum in the last ten years. Implementation and evaluation of the system described herein sits as a potential research tool within a large scale project which aims to create infrastructure for research into modelling of the human form being undertaken at EdVec in Edinburgh for use in the creative media and biomedical sectors. The technique implemented establishes an approximate maximal containing volumetric representation of the object, the visual hull. The hull is constructed by the mathematical intersection of a number of volumes which are defined by the silhouette of the object seen from various viewpoints. These volumes take as input a set of coordinates of silhouette contour vertices for a particular camera view image plane, and the coordinates of the camera centre of projection. Rays extrapolated from the camera centre through the silhouette contour image plane points and beyond define a generalised polyhedral cone for that view which is guaranteed to contain the object. Intersection of cones from several viewpoints define an approximation to the object as a polyhedral visual hull. Optimisations made to the input silhouette contours are used to reduce the dimensionality of the intersection calculations from three to two dimensions and thus increase the performance of the system. These optimisations use properties of epipolar geometry which describes the relationship between corresponding points of interest imaged from a multiple-view imaging setup. Ordering of the

input images is employed as a preprocessing step so that the representation of the images and epipolar constraint can be combined to maximise efficiency. In the following sections the background geometry which underpins the implementation is introduced, together with a survey of other relevant and precedent work in this area. The report continues with detailed instructions of how the implementation is achieved, followed by the results obtained in its operation. These investigations are then evaluated and discussed.

Chapter 2

Background

The techniques employed in the implementation of the system described in this work are drawn from the discipline of 3D geometry. Some specific results from this field are introduced here in order to highlight their utilisation throughout the report. In particular the different coordinate systems which are used and the matrices which relate these systems are introduced, along with an explanation of epipolar geometry which describes the relationship between multiple views in a scene.

2.1 Coordinate systems.

The transformation between different coordinate systems allows a multiple view 3D scene to be described in terms of its components. When dealing with multiple components it is natural to consider local descriptions in terms of the local coordinate frame, which in this case is a camera's coordinate system. Representation of a 3D object as a 2D image is achieved by application of a perspective projection, which maps a 3D point in space to a plane applying perspective foreshortening in the process. The line connecting the camera centre and the point in space intersects a camera image plane at a point with coordinates (x, y, f) shown in figure 2.1. This can be written as

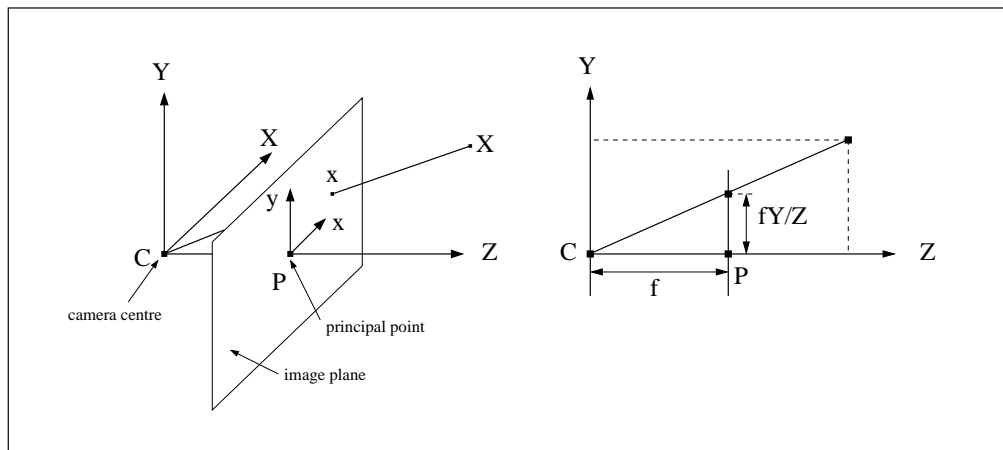


Figure 2.1: The Perspective Projection

$$\begin{bmatrix} \frac{fX}{Z} \\ \frac{fY}{Z} \\ f \end{bmatrix} = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X_W \\ Y_W \\ Z_W \\ 1 \end{bmatrix}$$

where x and y are coordinates in the camera coordinate frame, f is the focal length of the camera and $(X, Y, Z, 1)$ represents the coordinates of a point in 3-space in homogeneous coordinates.

This situation applies for orthogonal coordinate axes where one axis is perpendicular to the image plane. The axis intersect the image plane at the principal point with coordinates (p_x, p_y, f) . The main frame of reference for the whole system is the world coordinate system. Individual components relate to one another via this system and may be described by a rotation and a translation relative to the world origin (figure 2.2).

This may be written as

$$\begin{bmatrix} X_A \\ Y_A \\ Z_A \\ 1 \end{bmatrix} = \begin{bmatrix} R & T \\ 0^T & 1 \end{bmatrix} \begin{bmatrix} X_B \\ Y_B \\ Z_B \\ 1 \end{bmatrix}$$

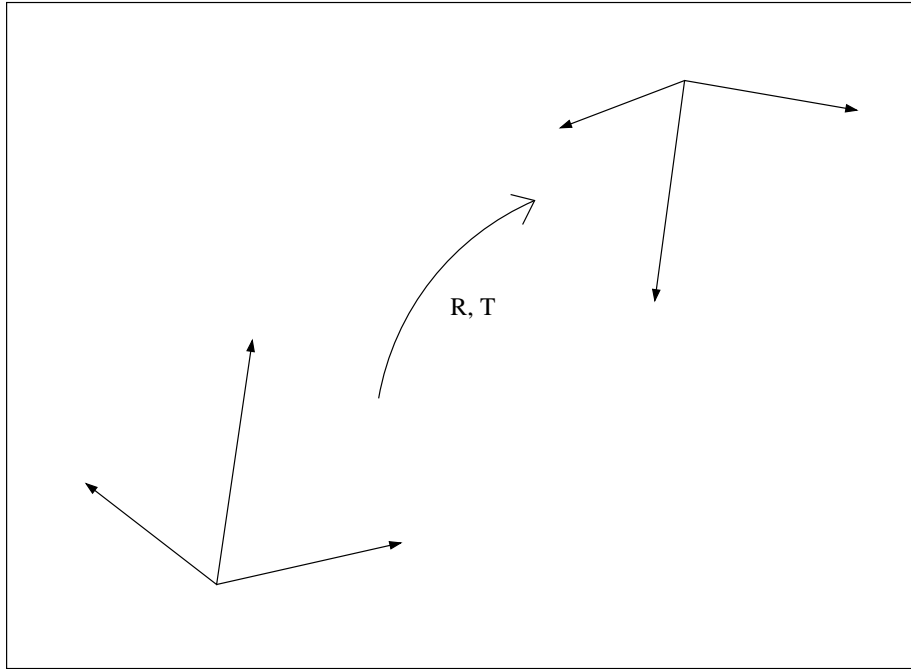


Figure 2.2: Relation between coordinate systems in world frame.

for any 2 locations in space A and B. R is a 3×3 compound rotation matrix and T represents the translation. This may be abbreviated to

$$X_c = [R|T]X_w$$

Due to the fact that pixel coordinates are defined with the top left of the screen as the origin, a third system relates camera coordinates to the image plane. Image coordinates (u, v) can be defined in terms of world and camera points

$$(X, Y, Z) \leftrightarrow \left(\frac{fX}{Z} + p_x, \frac{fY}{Z} + p_y\right)^T$$

where $u = \frac{fX}{Z} + p_x$, $v = \frac{fY}{Z} + p_y$. This can be expressed as

$$\begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} \leftrightarrow \begin{pmatrix} fX + Zp_x \\ fY + Zp_y \\ Z \end{pmatrix} = \begin{bmatrix} f & 0 & p_x & 0 \\ 0 & f & p_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}$$

where

$$K = \begin{bmatrix} f & 0 & p_x \\ 0 & f & p_y \\ 0 & 0 & 1 \end{bmatrix}$$

is called the camera calibration matrix and contains parameters which map between image points and rays in world space. Concatenating the relationships defining the three different systems it is possible to arrive at a single expression which allows a world point to be mapped directly to its image point representation. This expression is the projection matrix P and the mapping can be written as

$$(u, v, 1)^T = K[R|T] \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}$$

which shortens to

$$x = PX$$

where $P = K[R|T]$. At this point it should be noted that although it is possible to obtain an image point from its world point, an inverse injective mapping is not possible, except to obtain the line on which the 3D point lies (7).

2.2 Epipolar geometry.

Epipolar geometry describes the fundamental relationship between two views, and allows operations between views to be specified from a knowledge of the camera internal parameters (in K) and the relative position of the cameras. Figure 2.3a shows a typical two-view setup with 2 cameras A and B with centres C and C' , their respective image planes and a world point, X . The projection of X on the image planes is x and x' . The first important constraint is that the plane formed by C , C' and X has x and x' coplanar. This holds for any X . The baseline $C - C'$ joining the camera centres meets the image planes at e and e' , called the epipoles, and these points represent the projection of one

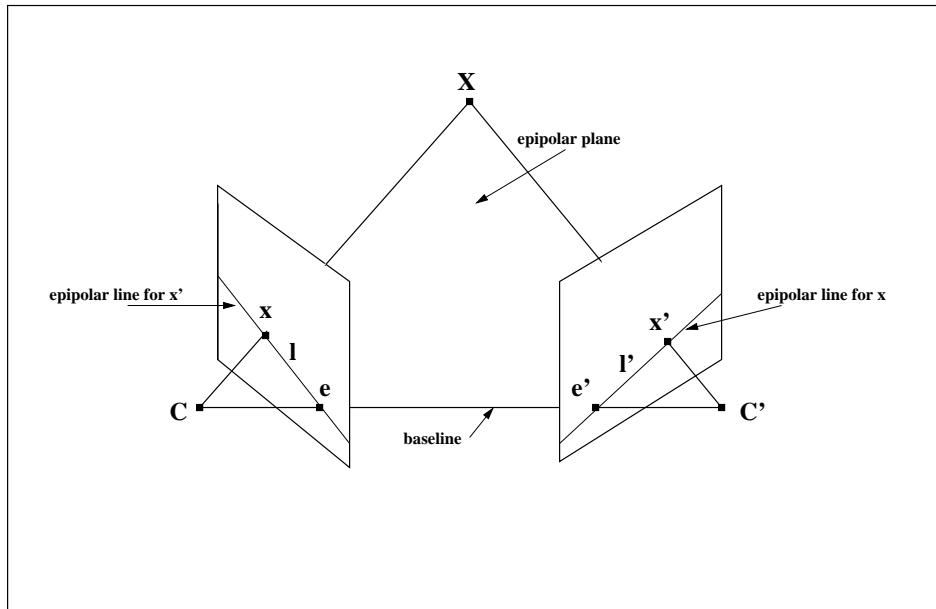


Figure 2.3: Epipolar geometry in a two-view setup

camera's centre on the image plane on the others image plane. Clearly, if the only information known is the camera centres and the image point x , a line projected back from C through x will contain X . This line projects to a line in image plane B. Thus a back-projected image point viewed by either camera will project to a line in the others image plane.

The establishment of two camera centres in space defines a baseline between them. The baseline is a fixed entity, which together with a number of points in world space will describe a set of planes hinged about the baseline. Such planes are called epipolar planes, and this relationship is illustrated in figure 2.3b. The figure also shows how two distinct vertically displaced points define two planes which intersect the image planes at two lines in each view. These lines intersect at the epipoles in the corresponding images planes and are called epipolar lines. Epipolar lines all have the useful property that they intersect at the epipole. Thus a set of image points in one plane projects to a set of epipolar lines in the other, tracing out what is called an epipolar pencil of lines. Clearly these constraints are useful where attempting to recover 3D information from

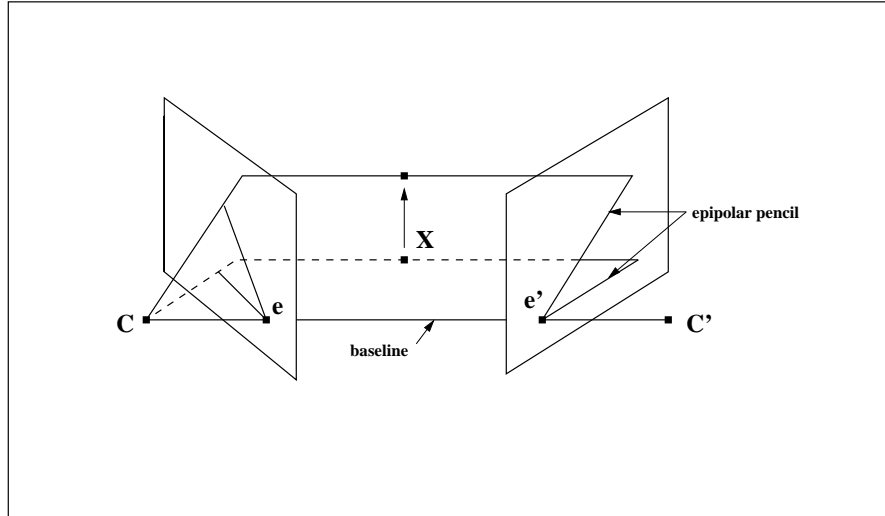


Figure 2.4: Epipolar planes and epipolar pencil

more than one camera.

Two useful algebraic representations of the relationship between two cameras in such a setup can be obtained in the form of matrices, the essential and fundamental matrices. Two camera coordinate systems, as in figure 2.3, imaging a point X at x and x' are related by a rotation R and a translation T , then

$$x' = Rx + T$$

Taking the vector product with T followed by the scalar product with x' :

$$x' \cdot (T \times Rx) = 0$$

This can be written as

$$x'^T E x = 0$$

in camera coordinates, where $E = [T]_{\times}R$ is the 3×3 essential matrix and $[T]_{\times}$ is the skew-symmetric representation of T . The essential matrix is the algebraic representation of epipolar geometry for known calibration. Using the calibration matrices, the essential matrix can be generalised to relate image points to camera coordinates:

$$x'^T K^{-T} E K^{-1} x = 0$$

or

$$x'^T F x = 0$$

where $F = K'^{-T} E K^{-1}$ is the 3×3 fundamental matrix. The fundamental matrix has very useful properties, particularly $l' = Fx$ and $l = F^T x'$ with x and x' in image coordinates. This means that the projection of an epipolar line onto image plane B is given by the operation of the fundamental matrix on the corresponding image point in A. Conveniently, the reverse mapping between B and A is achieved by the application of the transpose of F which mapped A to B. Inclusion of the calibration parameters in the fundamental matrix means that both the intrinsic and extrinsic parameters of a scene are encapsulated in a single relationship, an advantage compared to the essential matrix (11). Knowing K , R and T allows the whole epipolar geometry of a scene to be specified.

2.3 Previous and Related Work.

In recent times, much work has focussed on the problem of construction of realistic 3D models from multiple viewpoints. Traditionally, the approach has been intensity-based or feature-based image correlation. Success in these areas has been tempered by several limiting factors:

- views are required to be sufficiently proximal to achieve effective correspondences.
- large changes in viewpoint require accurate series of correspondences across images.

- 3D surface models have to be fitted to sparse data before detail can be added.
- lack of detail regarding occlusion results in models.

These factors have led to an interest in constructing scenes in terms of the volumes or surfaces derived from the images, replacing the image-based search with a scene-based search. These techniques can generally be described as volumetric scene modelling. Within this scope there are several approaches to reconstruction including dense stereo matching and shape from silhouette which build an explicit representation, image-based methods where only novel views are required, and hybrid approaches which draw on both model and image-based techniques (3). The approach adopted in this work is the shape from silhouette technique, widely studied in the past 10 years. All systems and algorithms which use this method have at their root a construction of the visual hull concept introduced by Laurentini (9). 3D information about shape can be gained from visual cues about an objects silhouette in 2D. Computationally this can be realised by first projecting rays from a viewpoint (camera centre) tangential to the silhouette outline of the object for all points on the outline. This defines a cone-like region in space called a silhouette cone which is guaranteed to contain the object (figure 2.4a).

Intersection of cones obtained from different viewpoints describes an approximating volume to the original object. The visual hull is defined as the intersection of the cones from the infinite number of viewpoints surrounding the object outside of its convex hull. This defines a maximal volume which is guaranteed to contain the object and which is at least as good a fit as the convex hull. The converse to this result is that two objects can be discriminated by silhouette from a viewpoint from a distinct visual hull. There are some limitations in this technique: Any global concavity within the object cannot be resolved by examining any of the silhouettes described by the visual hull. This reflects what is called a silhouette inactive surface (figure 2.4b). At the practical level, it is infeasible to compute an infinite number of cones from which to recover the visual hull perfectly. Some trade-off must be employed between constraining a complete hull and the expense of obtaining many cones. For this reason an approximating visual hull is generally calculated from which an approximate volume is described (figure 2.4b). The most common means of deriving a model via its visual hull is to

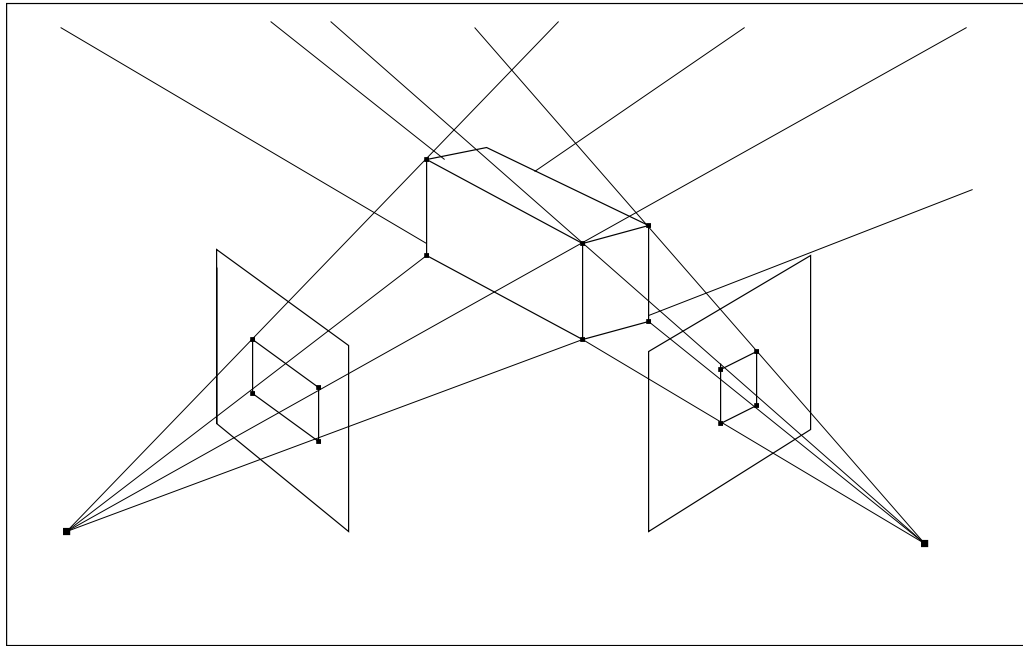


Figure 2.5: Object, image planes, silhouettes and silhouette cones.

discretise the model space into structured volume elements (voxels) at a progressively increasing resolution, calculating occupancy of the elements. Efficient resolution is normally achieved by linking the spatial subdivision via branch and leaf nodes of an octree structure. This eliminates the need to exhaustively calculate occupancy of large numbers of fine scale elements which contain empty space. The technique generally proceeds by projecting each silhouette cone of voxels onto every other image plane and then removing those voxels which lie outside of the boundary of the silhouette in each image plane. In this manner regions of space can be progressively removed where the object is *not* for each intersection. For this reason the technique is variously referred to as space or voxel carving (8).

Work related to this approach may be found in (6) which calculates the intersection of back projected silhouette cones in 3D obtaining volume occupancy from the result. An alternative approach is taken in (Snow et al.) which optimises the problem by computation of a global minimum of an energy function consisting of a pixel intensity data term and a local smoothness term. Applications in the field are numerous. A virtual

h

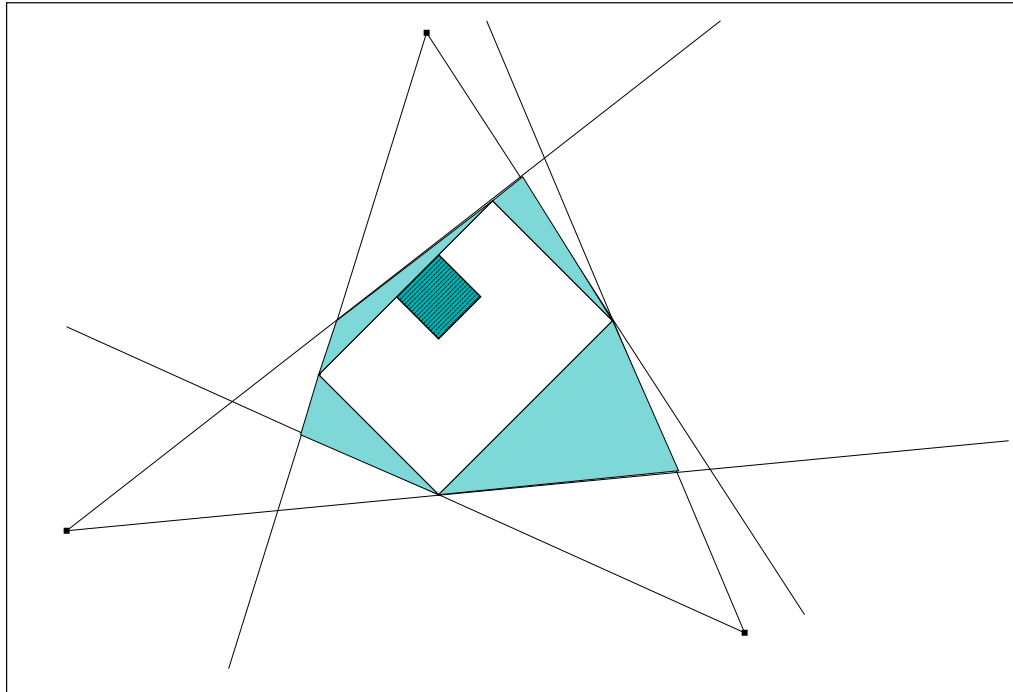


Figure 2.6: Object with global concavity and visual hull of object.

view generation system (12) utilises multiple view video streams of a scene to determine volume occupancy and specialised hardware to determine colour in the model producing photorealistic interactively rendered scenes for arbitrary viewpoints. The approach taken in (5) combines model reconstruction and non-explicit reconstruction in ellipsoid fitting for moving humans in real time, while the system in (13) constructs a simple model of a scene for each camera at each instant from range and intensity information for multiple video inputs. The collection of instants is recombined into a global 3D surface model of the scene which allows reconstruction of novel photorealistic views

An alternative approach, and the one presented in this work, is to obtain the visual hull by calculating an explicit and exact polyhedral model. This technique also uses the intersection between a silhouette cone and silhouette to build the visual hull, but instead of discarding areas outside of the hull, intersections are retained and lifted back into 3D to define polygonal boundaries where the hull *is*. The choice to implement a polyhedral representation of the visual hull in place of one realised by voxel carving is influenced by several factors:

- the method is newer and much less well documented so is ripe for investigation.
- the representation is view-independent so once calculated for a single set of inputs, the recovered hull lends itself well to being rendered in novel views.
- a polyhedral output is well suited to being rendered by standard graphics hardware.
- voxel based methods suffer from antialiasing effects which are a result of quantisation in the choice of element dimensions and topology.
- a polyhedral representation can be said to be exact for the number of views in the sense that it is composed of sets of lines which represent exact locations of the object.

A system which adopts this approach and which provides the algorithm which forms the basis of the work presented here is the Polyhedral Visual Hull (PVH) system (17). This system produces a real time reconstruction of the visual hull as a polyhedral

model. Silhouettes of an object from several cameras are extracted. Rays are extrapolated from a camera centre through the silhouette image points to define a silhouette cone for each image. The silhouettes are preprocessed into special data structures called Edge-Bins which are used to accelerate the calculation of cone-cone intersections in 2D. The intersections are then lifted back into 3D and combined to form a set of polygonal faces which describe the visual hull. In the system, images captured at the time of input are used to texture map the hull giving a photorealistic output.

Chapter 3

Method

Successful recovery of a polyhedral representation of a virtual object's surface from its silhouettes should negotiate the following stages:

- specification of a constrained model scene.
- recovery of the relative geometry of the scene.
- silhouette extraction of input images.
- implementation of an algorithm to extract a polyhedral model from the silhouettes.

These main difficulties can be subdivided and related to form a linear pipeline of sub-problems (figure 3.1). This chapter examines the methods used to achieve the first two main stages.

3.0.1 Model Scene Specification.

To produce a realistic simulation of the behaviour of a multiple camera imaging system, the initial stage of the problem concerned how to create and relate virtual cameras and objects. Suitability of an appropriate modelling environment and choice of test models had to be considered. The number of applications for modelling a multiple camera setup is large and any of Matlab, VRML or Java3D would have made a reasonable choice. In the end it was decided to use the Visualisation Toolkit (VTK). VTK is a

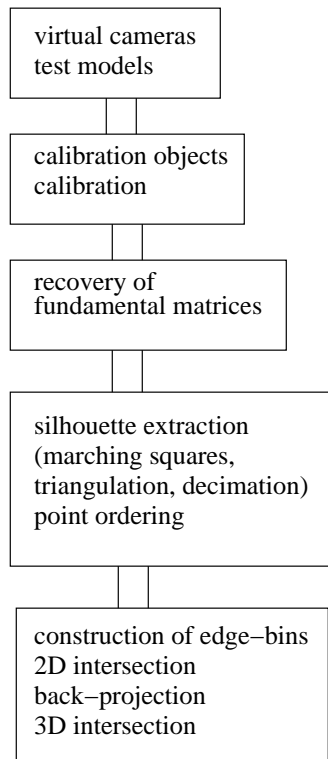


Figure 3.1: Stages of operation in the pipeline

powerful and extensive set of C++ class libraries built on top of a basic graphics library (OpenGL in this case). It allows easy specification and control of virtual cameras and models and is particularly suited to visualising 3D data. The fact that VTK affords implementation in the Tcl/Tk interpreting language (with which prior experience had been gained) allowed progress to be made while the C++ language was being digested. To create a model scene in VTK requires some objects to be specified in a pipeline:

- one or more source objects. In the absence of any external data local parameters are used to generate data. The source has one or more outputs and no inputs.
- zero or more filter objects to transform the data. A filter has one or more inputs and outputs.
- a mapper object to transform the data to graphics primitives. A mapper has one or more inputs and no outputs.

Implementation of a scene requires that a rendering window object is created along with a renderer object. This object is created with a default light and camera which are coordinated by the renderer along with an actor object representing the properties and geometry of an object in the scene. All concrete classes have user-accessible functions through which object behaviour may be controlled (REF VTK). Initially a test scene was created with 4 cameras rotated by 30° , 120° , 210° , and 300° about the world y-axis and -20° about the world x-axis with optic axes intersecting at a particular point so that the cameras should be well dispersed. Further test scenes had cameras in orientations as shown in figure 3.2. The initial test model was a cube with side length 600mm. Other models describe a 3D L-shape composed of two rectangular solids (figure 3.3) and a cube with a sphere adjoined at one corner. The rendered scene was found to be suitably proportioned using the default VTK camera parameters so these were left unchanged. The background for the scene was defined as a simple monochromatic flat shading. Although this is unrealistic, it was hoped to add backgrounds and one of the well known techniques for background removal, time permitting. In order to evaluate the accuracy of extrinsic parameters to be recovered later in the pipeline, each camera's centre of projection and transform matrix relative to the world origin was stored. This was done to provide a ground truth for subsequent parts of the system. The

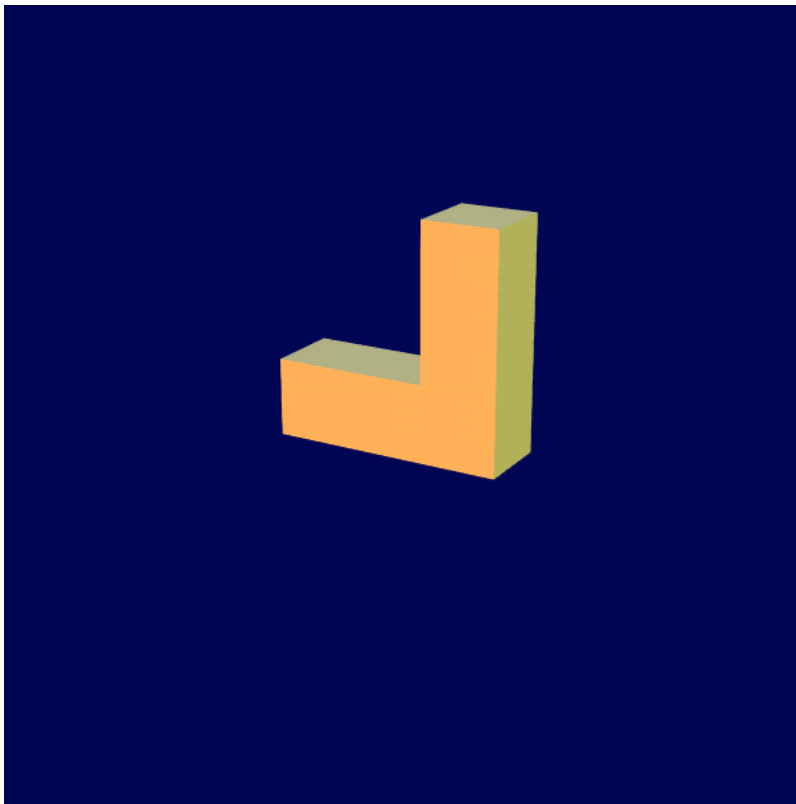


Figure 3.2: Sample input image

images obtained from the sets of camera views of the models were stored in portable pixmap (PPM) format, a standard colour image file format supported by VTK and which affords platform independence and is readily compressed.

3.0.2 Camera Calibration

This stage in the process is concerned with the extraction of the relative positioning of the cameras in the scene. As discussed in chapter two, knowledge of the calibration matrix K and the projection matrix P can lead to the complete description of the epipolar geometry of a scene. This in turn allows information to be gained regarding the third dimension. Also noted earlier, K contains the camera focal length and principal point, the intrinsic parameters. Camera calibration is a technique to recover K together

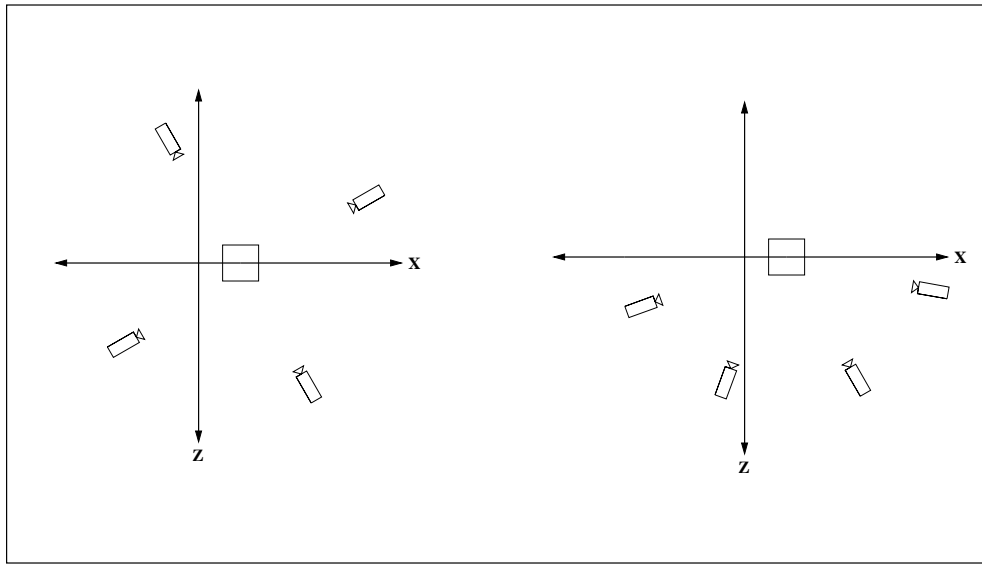


Figure 3.3: Initial and subsequent camera setups

with the rotation and translation components of the viewpoint frame. An image point is given by the action of a projection matrix on a world point. This can be written

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} p_{11} & p_{12} & p_{13} & p_{14} \\ p_{21} & p_{22} & p_{23} & p_{24} \\ p_{31} & p_{32} & p_{33} & p_{34} \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

or

$$x = P \begin{bmatrix} X \\ 1 \end{bmatrix}$$

where P has 11 degrees of freedom (p_{34} is a scale factor). Calibration proceeds by first ascertaining P from a set of known world point positions and their projective position on the image plane. P is then decomposed into the constituent K , R , and T components. For each pair of corresponding world and image points X and x , two linear equations can be written in terms of the elements of P :

$$u = \frac{xp_{11} + yp_{12} + zp_{13} + p_{14}}{xp_{31} + yp_{32} + zp_{33} + p_{34}} \text{ and } v = \frac{xp_{21} + yp_{22} + zp_{23} + p_{24}}{xp_{31} + yp_{32} + zp_{33} + p_{34}}$$

and this gives

$$xp_{11} + yp_{12} + zp_{13} + p_{14} = u(xp_{31} + yp_{32} + zp_{33} + p_{34})$$

$$xp_{21} + yp_{22} + zp_{23} + p_{24} = v(xp_{31} + yp_{32} + zp_{33} + p_{34})$$

The following matrix relationship can then be constructed, given n 3D points:

$$\begin{bmatrix} X_1 & Y_1 & Z_1 & 1 & 0 & 0 & 0 & 0 & -u_1X_1 & -u_1Y_1 & -u_1Z_1 \\ 0 & 0 & 0 & 0 & X_1 & Y_1 & Z_1 & 1 & -v_1X_1 & -v_1Y_1 & -v_1Z_1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ X_n & Y_n & Z_n & 1 & 0 & 0 & 0 & 0 & -u_nX_n & -u_nY_n & -u_nZ_n \\ 0 & 0 & 0 & 0 & X_n & Y_n & Z_n & 1 & -v_nX_n & -v_nY_n & -v_nZ_n \end{bmatrix} \cdot \begin{bmatrix} p_{11} \\ p_{12} \\ \vdots \\ p_{33} \\ p_{34} \end{bmatrix} = \begin{bmatrix} u_1 \\ v_1 \\ \vdots \\ u_n \\ v_n \end{bmatrix}$$

so with 11 unknowns and each correspondence defining two equations, at least six points are required to solve. Following this a linear least squares based or a non-linear minimisation approach can be applied to minimise the distance between the measured and projected point. The non-linear method is more robust and is used in the implementation of this work. With P computed thus, the matrix decomposes into K , R , and T . The product KR is the first 3×3 submatrix of P and can be decomposed using a QR decomposition resulting in the upper triangular K and orthogonal R . T is then given by(7).

$$T = K^{-1}(p_{31} \ p_{32} \ p_{33})^T$$

As noted, a minimum of six input points is required to calibrate. Points are introduced typically through imaging a physical object with an arrangement of well-constrained points, such as a checkerboard pattern. Corner points on the pattern can then be specified and their pixel coordinates calculated. To calibrate the virtual setup, a calibration object consisting of a planar black-and-white checkerboard pattern was constructed in VTK. Initially 20 images of the object in various orientations was produced. A library running on Matlab which automates the mathematics of the calibration process was employed. This library, the Camera Calibration Toolbox (1) takes as input a number of images of a planar checkerboard calibration object. The first stage

involves the user specifying the grid's extremities with four mouse clicks at the corners. The number of squares in the x and y directions is specified and their dimensions so that the rest of the grid corners locations can be computed affording a large number of known points. Corner extraction is repeated for all images before the calibration matrix corresponding to that camera view is computed according to the methods described above. With real cameras and especially those with poor quality optics, distortion of the images often occurs particularly towards the image boundaries. Facilities to compensate for this radial distortion are included with the library, although in the virtual setup this is not a concern. Based on the computed parameters, the grid points are projected onto the original images and a plot of the reprojection error is produced. From this it is deduced whether any of the process needs to be repeated for increased accuracy in any of the images. Initially when this procedure was performed, the calculated reprojection errors were a significant fraction of a pixel (around 0.4). Antialiasing was performed on the input images via a simple function call in VTK and the errors were reduced substantially (to 0.1 of a pixel). Extraction of the extrinsic parameters R and T is an optional final stage in the process. Test scenes were constructed showing the calibration object in positions according to the model scene virtual cameras. The grid of corners was specified and using the intrinsic parameters for each camera in K , the 3D location of the grid in the world reference frame was computed. At this stage a slight glitch in the toolbox was discovered. As figure 3.2a shows, relative to the world coordinate frame, two of the virtual cameras lie in the negative z sector. When extracting corners with the toolbox, the user is constrained to specify the first clicked point as the origin of the world frame. If the user clicks on the lower left corner of the pattern, the toolbox assumes that the user is referring to a right-handed coordinate system, while a click in the lower right corner infers a left-handed system. This is generally a user friendly automation, but in the case of multiple views such as the test scene here, slightly confusing results can be obtained. Using a right-handed coordinate system, the world coordinate origin is constrained with a click on the lower left corner of the calibration object. However, for the cameras in the negative z sector, the same world origin now lies at the lower right corner and although the coordinate frame is still right-handed, the toolbox produces R and T corresponding to a left-handed sys-

tem. This problem was rectified by applying a simple rotation of 90 about the z axis and 180 about the x axis to the resulting matrices. Perhaps a slight improvement to the toolbox would be to make the axis system specification explicit from the user or default to one system.

A final consideration is that the makeup of K is slightly different for CCD devices compared to standard cameras in that the image sensors in a CCD chip are arrays of pixels which may not be square. This results in a skew between the x and y components which is reflected in the value of the element in the first row and second column of K as the cosine of the angle of skew. In the virtual setup as with most cameras, the angle of skew is 90 so the value of this element is zero.

3.0.3 The Fundamental Matrix.

The discussion in chapter two emphasised how knowledge of the fundamental matrix may be used in relating image points and epipolar lines between different camera views. For this reason it was deemed necessary to recover the matrices describing the model setup. Despite the fact that in general the fundamental matrix expresses epipolar geometry in the case where uncalibrated cameras are being used, knowledge of the matrix can be useful in the calibrated case also. The essential matrix discussed in chapter two was developed before the fundamental matrix and can be thought of as the specialisation of the fundamental matrix, where the calibration parameters must be known in order to recover mappings between image points and epipolar lines. The more generalised fundamental matrix simply removes the requirement to know calibration parameters as they are incorporated into the matrix. Thus, knowing the fundamental matrix requires that less calculation need be done between image domains compared to using the essential matrix, and image points can be projected to epipolar lines directly. Algebraic derivation of the fundamental matrix is reflected in the equation

$$x'^T F x = 0$$

where $F = K^{-T} E K^{-1}$ and $E = [T]_{\times} R$, $[T]_{\times}$ is the skew-symmetric representation of translation between camera coordinate frames and R the rotation component. This reflects the fact that corresponding image points and the camera centres are coplanar.

In practice, derivation of F in this manner proves to be difficult and safer and easier methods exist. It was proposed to employ a linear method to compute F known as the 8-point algorithm. This algorithm uses the match up of corresponding image points between views in the equation $x'^T F x = 0$. With a sufficient number of matching $x : x'$ pairs (the minimum number being seven) a set of linear equations may be defined and used to solve for F . With corresponding points $x = (u, v, 1)^T$ and $x' = (u', v', 1)^T$ each point match gives rise to one linear equation in F . In terms of knowns in x and x' and unknowns in F :

$$u'uf_{11} + u'vf_{12} + u'f_{13} + v'uf_{21} + v'vf_{22} + v'f_{23} + uf_{31} + vf_{32} + f_{33} = 0$$

From a set of n point matches, the following matrix can be constructed:

$$Af = \begin{bmatrix} u'_1u_1 & u'_1v_1 & u'_1 & v'_1u_1 & v'_1v_1 & v'_1 & u_1 & v_1 & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ u'_nu_n & u'_nv_n & u'_n & v'_nu_n & v'_nv_n & v'_n & u_n & v_n & 1 \end{bmatrix} f = 0$$

As a homogeneous set of equations, f can only be determined up to scale and for a solution to exist, A must have minimum rank of eight. A rank of exactly eight specifies a unique solution which can be obtained by linear methods. This specifies a solution for perfect data unperturbed by noise. In real applications noise will nearly always be present and this has the effect of increasing the rank of A to nine. In this case a least squares solution must be found which is equal to the singular vector corresponding to the smallest singular value of A . This is found from the last column V in the singular value decomposition $A = U^*DV^T$ (7). To calculate F , sets of image point correspondences across camera views in the model scene were obtained and the method above, implemented in the Structure and Motion toolkit in Matlab (14) was used to compute F .

With model test scenes specified, calibrated cameras constrained, and values for fundamental matrices recovered, the next stage in the system design involves processing the input images in order to extract silhouettes, and following this, the main PVH algorithm must be implemented. Image processing and the PVH algorithm are dealt with in the following chapter.

Chapter 4

Implementation

The second two main tasks in the subdivision of the specification introduced in the previous chapter are examined now. These deal with the implementation of image processing and the main PVH algorithm. Discussion of the techniques employed to achieve image processing and a detailed description of the main PVH algorithm precede a full discussion of the implementation structure and operation. The main implementation was written in C++ for the power, robustness and neatness that building a C-based object-oriented environment provides. Advantages in terms of portability and efficiency also influenced the choice.

4.1 Image Processing.

As discussed in chapter two, the method chosen to achieve reconstruction of a model involves construction of a visual hull using the shape from silhouette technique. This requires image processing to be performed initially, in order to extract a representation of the silhouette of the object from a particular image. The true definition of a silhouette is an outline portrait of an object infilled with black. For computer vision techniques, the infilling of the interior with black is a superfluous concern so we are left with what is actually sought in this process, a silhouette contour, just the outline of the object in question. Due to the model scene being implemented over a simple monochromatic background, the images are presented in the same manner as images

from a real system which has had the background removed. The algorithms used to achieve contour extraction are introduced before being examined in more detail. A technique for extracting the contours in a 2D image is the marching squares algorithm (W. Lorensen), a cell based divide and conquer strategy which preprocesses the input into an indexed case table recording the inclusion or exclusion of a cell vertex with respect to the contour value. The operation of this algorithm alone is sufficient to reduce an image to a single contour, however the output is an exhaustive point-by-point isoline list of every pixel pair which constitutes the contour. To define a silhouette based on a large contour point set and then calculate intersections of a number of pairs of cones will obviously be computationally expensive. With this in mind, and considering that the implementations of most multi-view model recovery systems compute a temporal sequence of dynamically changing objects, it was proposed that this system like others should reduce the quantity of points being input to the main algorithm. A suitable technique to achieve this is the decimation algorithm (W. Schroeder), a polygon-data compression algorithm which removes or preserves points in a triangular mesh based on point boundary criteria and relative distance of mesh vertices from reference lines. Importantly, the algorithm preserves the original topology of the input and makes a good approximation the the original geometry (19). As decimation requires a triangular mesh framework as input and marching squares produces a list of isolines as output, a further process is required to carry out the intermediate transformation of lines to polygons. For this stage, Delaunay triangulation was proposed. This technique takes a set of individual points and creates a triangular tessellation in 2D which has been shown to be the optimal triangulation based on the interior angles of the triangles in the resulting mesh (4). With this design specified, it was hoped to extract the silhouette contour for each image and reduce the number of output points for efficiency later in the pipeline, while preserving the original geometry of the input.

The marching squares algorithm takes as input a regular structured dataset, in this case the PPM images of the test scenes. The image is divided into square cells which have pixel dimension. The next stage is to assign each cell a bitwise code based on whether each vertex in the image lies inside or outside the specified contour value. Initially each vertex is classified thus, then the cells are classified based on the com-

bination of the four vertices in the cell. As each vertex may be in one of two states there exists 2^4 possible ways to describe each cell. With the binary state of each vertex setting a bit value, this defines an index into each vertex. Examination of the state of the four vertices in a cell defines a topological state for each cell, constraining the cell path of the contour. Interpolation between cell vertices is then used to ascertain the exact location of the entry and exit points of the contour. In this manner the contour is constructed for each cell. In order to maintain connectivity across cells, the algorithm interpolates in the same direction relative to vertices to avoid numerical round-off error (19).

With a set of contour points defined, compression can be employed via decimation after the point set is triangulated with Delaunay triangulation. Delaunay triangulation is a mesh construction technique which is defined by two useful properties:

- the centre of the points which make up a triangle define a circle which passes through those points, the circumcircle of the triangle. This circumcircle will not contain any other points in the dataset.
- Delaunay triangulation creates an optimal triangulation by maximising the minimum angle for all triangular elements.

The most commonly implemented Delaunay algorithm, and that used in this work is the Bowyer-Watson algorithm (2)(18). The algorithm proceeds by initially constructing one large Delaunay triangulation which bounds the whole dataset. Points which populate the dataset are injected one at a time and all circumcircles in the existing dataset which contain that new point are identified (figure 4.2). This involves first locating the triangle which contains the point, then the neighbouring triangles are selected to see if their circumcircles contain the new point and so on until all circumcircles are found. All the triangles which belong to these circumcircles are deleted creating a cavity. Finally the new point is joined to all the vertices on the boundary of the cavity. This process continues with all points being treated thus before the final step is to remove the triangles connecting the points forming the initial bounding triangulation (4). With the point set specified as a triangular mesh, this can now be passed as input to the decimation algorithm. This algorithm attempts to reduce the number of

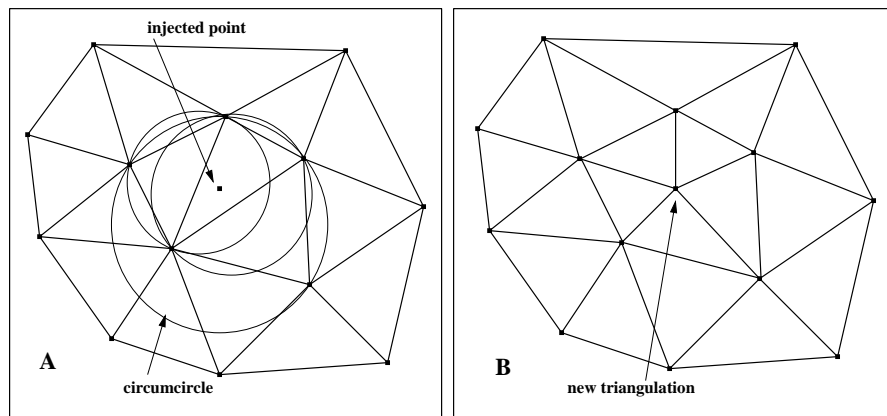


Figure 4.1: Delaunay triangulation.

points in a triangular mesh by a three-stage process of vertex classification followed by error evaluation followed by triangulation.

The first stage of point classification visits each vertex in the mesh iteratively and classifies it in an attempt to establish the local topology and geometry of the mesh. Classification determines whether a point is selected for deletion or explicitly retained according to which one of five possible categories it falls in from simple, complex, boundary, interior edge and corner. A simple vertex is one that is completely surrounded by triangles so that each edge from the vertex is used by exactly two triangles. If a vertex is not completely surrounded or a triangle not in the cycle of triangle uses the vertex, it is classified as complex. A boundary edge point lies within a semicircle of triangles. Simple points can be classified as interior edge or corner points, but this is based on 3D character and need not be considered here. Complex (and corner) points are never deleted from a mesh but all other vertices are eligible to be deleted. When a point is marked as a candidate for deletion, the next stage is to estimate the error that would be incurred were the point and its conjoined triangles deleted. The error is taken as a function of the distance a point makes with a reference line (plane in 3D). Simple points in 2D are always removed. Boundary and interior edge points are taken to lie on an edge and here the distance to that edge from the point defines the error such that a point which is close enough to the edge will be deleted while a point which is not will

be retained. When a point is deleted, all associated triangles are also removed leaving a gap to be triangulated in a more simple manner. Although the algorithm is being used in a 2D context, it is designed to operate in 3D as well. For this reason it employs a 3D triangulation technique. The cycle of points formed after deletion of a vertex is divided by two with the insertion of plane. To be a valid split, all points in the subdivisions must lie on opposite sides of the plane and the split must create two cycles which both conform to an aspect ratio test. This ratio is a measure of the minimum distance of a point in a subdivision to the split plane relative to the length of the plane. Once a suitable split plane is found, each subdivision is recursively split until each contains only three points. At this stage the process halts and a triangle is created.

After the operation of the three algorithms, the output is a reduced set of points in x and y representing an approximation to the silhouette contour of the object.

A further processing step which had not been foreseen surfaced at this point. The main algorithm like other silhouette techniques operates via the interaction between consecutive pairs of points in cones. Using the fact that values obtained pertaining to the second point are used to derive values for the first point in the following pair is a common optimisation which reduces searching times compared to when an arbitrary sequence of points is used. Unfortunately the output from the marching squares algorithm does not represent such a consecutive sequence of points which meant that the some way would have to be found to order the point list into the consecutive manner that they appear in the contour. An initial approach to this problem involved somehow specifying a central point to the interior of the contour and creating a list of points by sweeping a radial line through the whole point set, adding to the list each time the sweep arm encountered a point. This simple approach would work fine for a convex polygon, but would incorrectly insert points at concavities. A subsequent approach was taken where a list of cyclically ordered points could be created by measuring the distance between points in pixels. This approach begins with any point in the dataset, flags it to prevent it from being considered in subsequent iterations, and searches through the whole list of points storing the location of the point which is closest to the current point. This nearest point is then added to the list and flagged. The process continues until all points have been added. Obviously with this technique, it is not specified

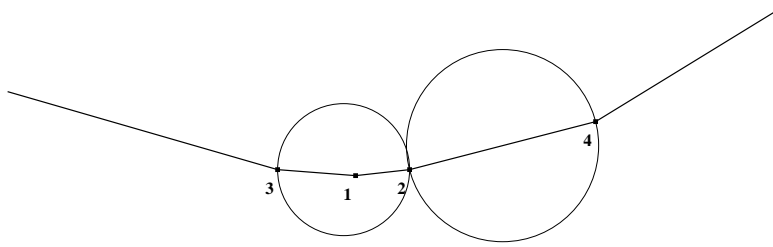


Figure 4.2: Illustration of degenerate case.

whether the ordering of points results in a clockwise or anticlockwise ordering, this factor will be determined by the relationship between the initial and second points. One further requirement for this algorithm is illustrated by the set of points in figure 4.4a.

The operation of this algorithm on a set of points with this arrangement will add point 3 to the list incorrectly before point 4 due to a gap in points between 2 and 4. This situation is not unlikely in the first few iterations of the algorithm until points which lie to the rear of the initial point are sufficiently distant. A partial solution to this problem was found by specifying a vector between the current and previous points, and a vector between the current and candidate points. The angle made by these two vectors can then be calculated where a particularly obtuse angle will suggest that a candidate point lies in the same direction as the other two points. A low angle will suggest that the point lies in the opposite direction, in which case the point can be flagged and emplaced at the start of the list instead of the rear (figure 4.4b).

With the solution to this problem being non-trivial, limitations still exist as it stands: spatially adjacent members of the point set must be close enough together to prevent the erroneous inclusion of points which are made candidate by local concavity (figure 4.4c). This may well prohibit the correct ordering of any set of sparse data and will very likely have an incorrect result with self-intersecting data. Conversely, the rel-

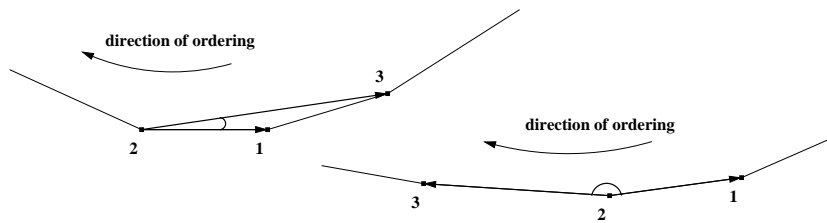


Figure 4.3: Illustration of use of vector angle to order points correctly.

atively high number of contour points generated in the image processing stage of this project supports operation of this algorithm and allows this technique to be extended to work on models which do not consist of simple geometric shapes. this approach will not produce a correct result when faced with multiple contours such as separate objects or those with holes.

4.2 Polyhedral Visual Hull algorithm.

The Polyhedral Visual Hull (PVH) algorithm developed by Matusik et al (17) takes as input a set of contour-representing polygons from a number of calibrated cameras and uses these to build a polyhedral representation of the visual hull. A term used throughout the text here is 'cone face'. This represents the triangle formed by a camera centre and lines extrapolated from the centre through two image points in the silhouette contour and beyond. A silhouette cone is composed of adjacent cone faces. The initial phase of construction of the perspective version of the visual hull is to define a silhouette cone of points with apex at the camera centre and edges projected back through each point in the polygonal representation of the silhouette, thus defining a polyhedral cone-like object. As noted before, each cone is guaranteed to contain the object being imaged as a silhouette. Extraction of the visual hull is then achieved by calculation of

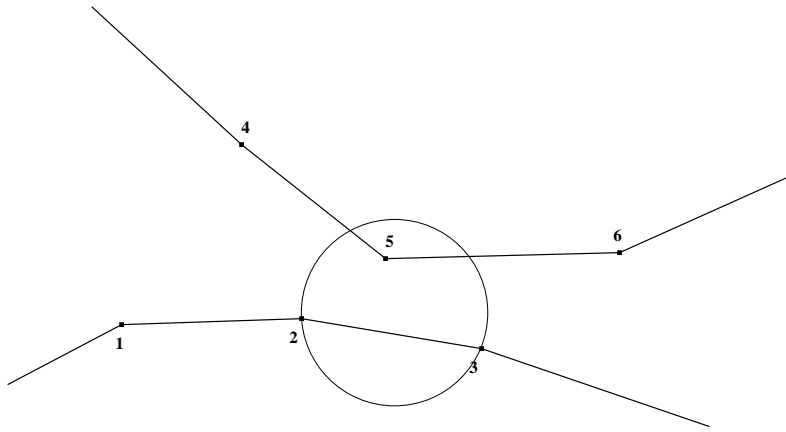


Figure 4.4: Local concavity causes the erroneous selection of point 5

the polyhedron formed by the intersection of all the polyhedral silhouette cones. Each adjacent pair of points in the silhouette contour will contain the faces of the visual hull. That is, faces of the visual hull are constrained to lie on the cone faces of the original silhouettes. This fact suggests a method to compute the cone-cone intersections: for each cone face in turn in a silhouette, calculate its intersection with each other cone, then for each of these results calculate the intersection within each cone face. This will define each face of the visual hull. As the calculation of the intersection of polyhedral cones is in 3D thus computationally expensive, the algorithm proposes to reduce the calculation into 2D by employing features of epipolar geometry. This reduction uses the fact that each cone face is defined by a 2D silhouette in cross-section. Thus intersection in 2D can be achieved by projecting a cone face through silhouette A onto the image plane of silhouette B and so the cone face will lie across the silhouette in B. This polygon-polygon intersection can be calculated much more readily and efficiently before the result is projected back onto the cone face in 3D. The algorithm can be divided into three sections:

1. creation of data structures optimised for 2D intersection calculation.
2. intersection of cones in 2D.
3. calculation of the visual hull faces.

4.2.1 Optimised Data Structures.

The calculation of 3D intersection of intricate multiple instances of distinct polyhedra with each other is computationally intensive. When the system employing these calculations is to have the capacity to extend its utility into having temporal constraints put on its operation, expensive computation should be minimised. To this end, the PVH algorithm preprocesses each input image into edge intersection reservoirs to take maximum advantage of epipolar geometry in describing a 2D scene from its 3D origin. The epipolar constraint is used in two ways:

- all points on image plane A, when projected onto image plane B will project to an epipolar pencil in image plane B. This allows each epipolar line in B (and thus each point in A) to be parameterised in terms of the gradient that each line in the pencil makes with the tip of the pencil, the epipole. The line gradient is henceforth called α .
- projection of a 3D polyhedral cone in space to a flat epipolar pencil in an image plane immediately reduces the domain from three to two dimensions. This will simplify construction of the data structures and improve performance incurring only the cost of recovering the third dimension at a later stage.

Data structures called edge-bins are constructed which make use of these properties. The process is described using a projected cone from the image in A and a silhouette contour in image plane B (figure 4.5a). To fully understand the algorithm, one should appreciate how the two structures in A and B interlink.

Firstly the epipolar projection point of camera A's centre is computed on image plane B. Then the contour points in silhouette B are ordered against the α value that each point makes with A's epipole. This creates a list of silhouette points ordered by α . Note that adjacent points in the list do not have to be physically adjacent in the contour. Although not made explicit in the algorithm as reported in (17), the list of points ordered by α must be able to index into the list of points ordered cyclically to ascertain which points in the cyclic list precede and follow the current point in the α list. An edge bin is defined by the two lines from epipole A associated with two

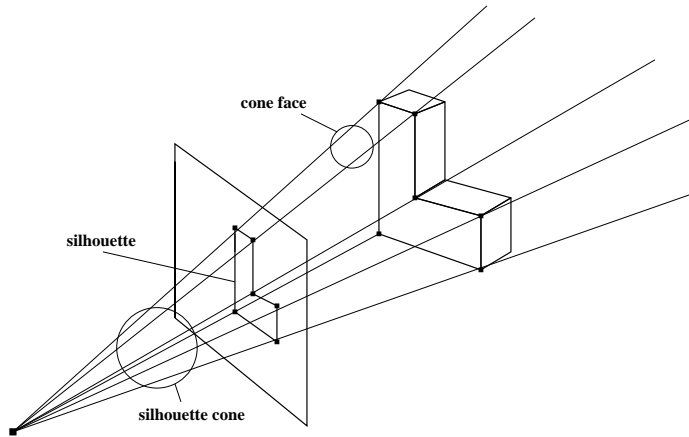


Figure 4.5: Silhouette contour and silhouette cone

adjacent points in the alpha list. Each bin has gradient α_{st} and α_{end} at its boundaries (figure 4.5b).

Along with an α_{st} and α_{end} value, each bin has a set of polygon edges associated with it. The set of edges is defined by the list of all silhouette edges which lie between α_{st} and α_{end} . Note that each silhouette vertex defines a bound on a bin and the genesis point for two edges. The list of edges in each bin is created by traversing the list of α ordered vertices from either the vertex with lowest α to that with highest α , or vice-versa. For illustration consider construction of the first and second bins in figure 4.6a traversing vertices in the direction of increasing α (from $v1$ to $v5$). For bin 1, its lower boundary α_{st} is the α corresponding to the first vertex in the α -ordered list ($v1$). At this vertex, two edges begin ($e1$ and $e5$) so both are added to the bin's list of edges. Bin 1 ends at the next vertex in the list ($v2$), which defines α_{end} for bin 1. Note that α_{end} for bin 1 is α_{st} for bin 2. This is the first manifestation of optimisation in the process: during bin construction, α values corresponding to bin boundaries do not need to be searched for. After construction of the initial bin, to create the list of edges to be added

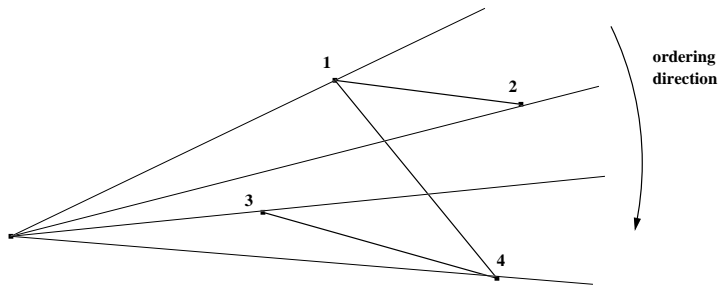


Figure 4.6: Points ordered by α are not necessarily adjacent in the contour.

to subsequent bins the rules of a subsidiary algorithm are followed:

for each new Bin_i

 get α_{st}

if other endpoint $\alpha > \alpha_{st}$

 add edge

else continue

 get edge list for Bin_{i-1}

for each edge in Bin_{i-1} not associated with α_{st}

 add edge to Bin_i

The list of edges in the preceding bin are carried over during selection. Define the vertex found at the start of the new bin as the 'current point'. Each edge from the previous bin has the α value of its endpoints examined and those that do not have an endpoint with an α value the same as the current point are added to the list for the new bin. Next the index into the cyclically ordered list is used to ascertain the α values of the endpoints of those edges which emanate from the current point in the contour list ($v1$ and $v3$). If the endpoint α value is higher than or equal to that of the current point, that edge is added to the list for the new bin. Otherwise it is not added. Following this set of rules, the edge list for bin 2 is constructed (as $e1$ and $e4$) from the edges in bin

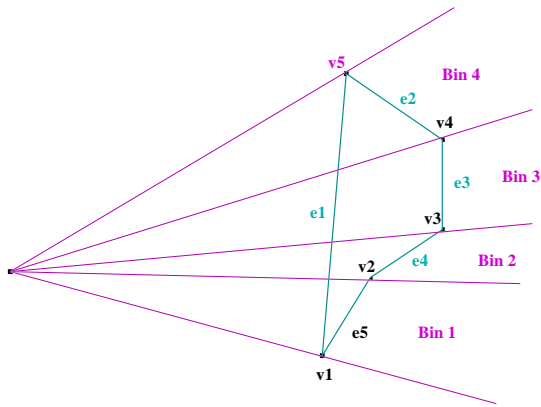


Figure 4.7: Simple construction of edge-bins

Table 4.1: Edge-Bins and contents for Figure 4-6a

| Bin 1 | Bin 2 | Bin 3 | Bin4 |
|--------|--------|--------|--------|
| e1, e5 | e1, e4 | e1, e3 | e1, e2 |

1 and the characteristics of the edge endpoints associated with bin 2's α_{st} point. The same edge list selection algorithm governs construction of subsequent bins. Table 4.6c shows the final edge bins for the figure in 4.6a.

Note that for construction of bins in order of progressively decreasing α , the sense of inequality tests should be reversed. The choice of traversal direction is arbitrary at the time of implementation. Figure 4.6d-f show some more complex situations which influenced the particular form of the algorithm as implemented in this project. Figure 4.6d shows a case where a local concavity occurs at $v2$. If construction of bin A begins at $v1$ then the selection algorithm carries over and adds edge $e1$. Edges $e2$ and $e3$ are associated with $v1$ and as both their endpoints have higher α than $v1$, both edges are added. Construction of bin B begins at $v2$. Edges $e1$, $e2$ and $e3$ are carried over from bin A and since edges $e1$ and $e2$ are associated with $v2$ they are not considered by this part of the algorithm. $e3$ is added to the list. The second loop in the algorithm examines $e1$ and $e2$ (as associates of $v2$). The endpoints of both edges have lower α

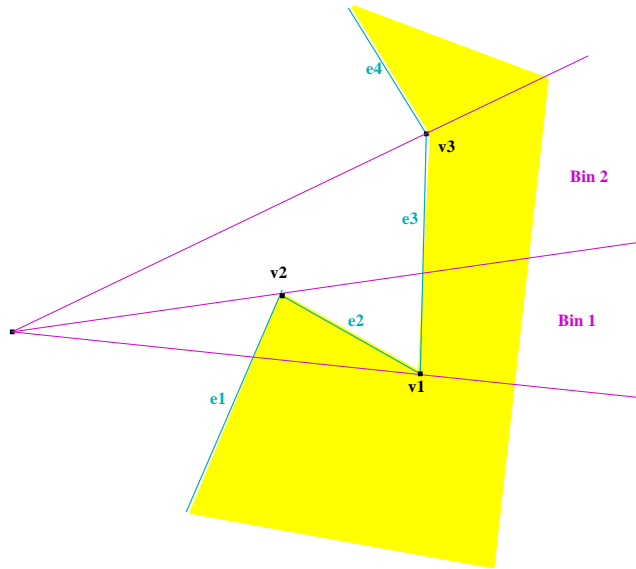


Figure 4.8: Edge-Bin construction around local concavity

than $v2$ so neither is added.

Figure 4.6e shows a case where two vertices have exactly the same α . The lower bin is defined as ending at the next α point in the list. This point may be $v1$ or $v2$, determined by the order the points were encountered in the cyclically ordered list during sorting by α . In either case, the same selection rules apply and edges are added, deleted or carried over in the same way. At the bin boundary an infinitely narrow bin is created with $\alpha_{st} = \alpha_{end}$.

Figure 4.6f shows the edge-bin construction process where a cone face straddles several bins. This example is discussed further in the following section as it leads to the requirement of a further processing step. A final small optimisation is made in the construction of the bins. This sees the edges in each bin ordered by the distance that each edge is from the projected epipole. The reason for this ordering is it allows the intersection calculation to proceed more easily, as explained in the following section. With edge-bins constrained by the projected apex of cone A and silhouette vertices in B it can be appreciated that any line forming part of the cone A in 3D will project onto

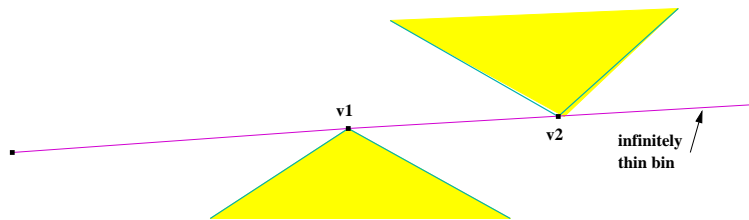


Figure 4.9: Edge-Bin construction where two vertices have exactly the same α

image plane B as an epipolar line which will fall within the bounds of one of the bins just constructed. Thus intersection can proceed by calculating the intersection of the projected cone lines with the edges in the bin within which the line falls.

4.2.2 Intersection of Cones.

Just as construction of edge-bins uses a pairwise boundary definition in inception, so the intersection calculation considers a pair of points at a time. This stage of the algorithm operates by calculating the intersection of each projected cone face from cone A with the edges in the edge-bins in B (figure 4.6f). A cone face projects onto B defining two epipolar lines with associated values that they make with the epipole, α_1 and α_2 . To calculate the intersection of the cone face with the silhouette, the bin within whose bounds α_1 lies, is found. The intersection between the line with α_1 and all the edges in the bin is computed and the points obtained used to initialise a set of intersection polygons. At this point, the reason that bin edges were ordered by distance from the epipole becomes clear in that the intersection points will be calculated in an order which allows the points to be specified in a cyclical order, producing convex polygons. Examination of figure 4.6f shows that only triangular or quadrilateral poly-

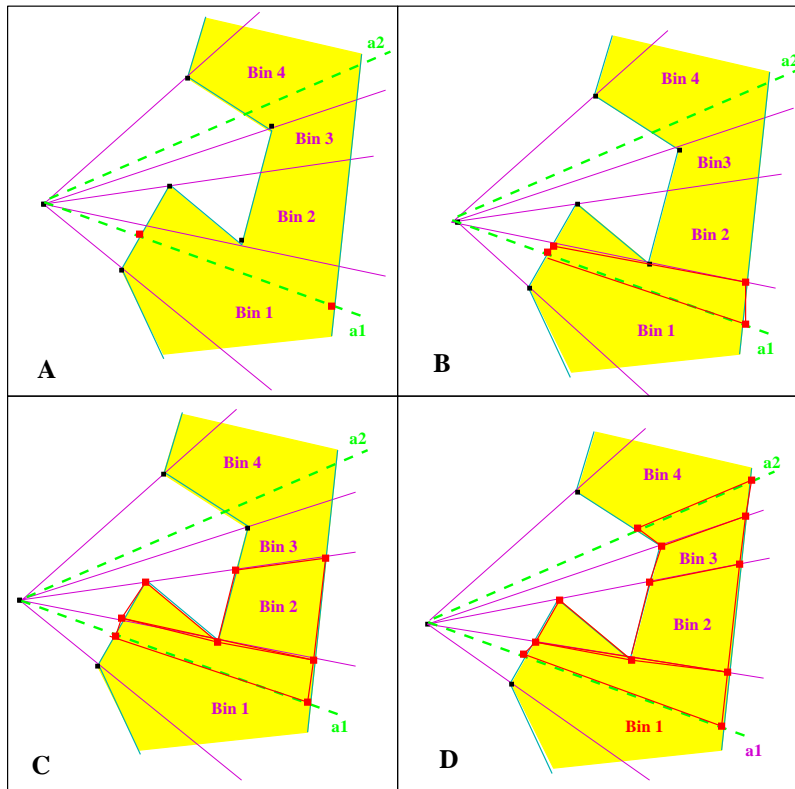


Figure 4.10: Edge-Bin construction where a cone face straddles several bins

gons are produced at this stage. This means that each polygon is constructed in two stages: the points (or point) which initialise each polygon, and the points (or point) which close it. The points which initialise a polygon are computed in order of distance from the epipole, and to ensure that closure of the polygon defines a convex shape, the closing points are added one to the front of the list of points and one to the back. With the intersection polygon construction underway, bins are traversed in the direction of the second cone face line, α_2 . If α_2 lies outwith the current bin, then the intersection points created from the intersection of the upper bin bound with its edges are added to the intersection polygons. In this manner construction continues across bins until the bin whose further extent lies beyond α_2 is reached. Then the cone face upper bound, α_2 , is intersected with the edges in the current bin to form the points which complete the intersection polygons for that cone face. With the intersection of the cone face calculated, the process is repeated for the adjoining cone face. This explains the necessity of pre-ordering the points cyclically immediately following image processing. This means that the next cone face chosen lies immediately adjacent to the last one, which means that α_2 for the previous cone is α_1 for the new cone so a search for the new line location in the bins has been avoided. The initial intersection points for this new bin's polygons are also known as they are the same as those which closed the last bin. This process is continued for each face in the cone resulting in a set of intersection polygons for the intersection between one cone and the silhouette of another. The example in Figure 4.6f shows the creation of four intersection polygons for one cone face. Since each cone gets intersected with each other cone in the overall operation of the algorithm, an individual cone face takes part in one intersection for each of the other views in the setup and therefore that number of sets of intersection polygons is produced for each cone face from this stage of the process. In order to prime the sets of intersection polygons for intersection between cone face sets, it is necessary to amalgamate the members of each set. It was proposed to employ a clipping algorithm suitable for calculation of both the union and intersection of sets of arbitrary polygons. A library in C called the Generic Polygon Clipping library (10) was used for this purpose. The library uses the generic solution to polygon clipping algorithm developed by Vatti (15). This is a robust algorithm which allows convex, concave or self-intersecting

polygons to be clipped with good efficiency. In the algorithm, one polygon (the subject) is clipped against another (the clip). A polygon is defined by a set of left and a set of right bounds. Bounds begin at a local minimum and end at a local maximum. Subject and clip polygons are traversed once to specify the bounds. Each edge of a bound is assigned a flag depending on whether it is a clip or subject edge. Next a set of horizontal lines is defined with one passing through each vertex in subject and clip so that adjacent pairs of lines define a horizontal strip. All the edges in a strip are added to a lookup table after being ordered by increasing x coordinate at the base of the strip. Left and right bound vertices are classified as those which start or end a bound while intermediate vertices do not. Vertices are further classified according to which clipping operation has been specified. With this setup, horizontal strips are traversed vertically and intersections between edges in each strip are evaluated and assigned to the output relative to the clipping operation (15). Thus far, for a single cone on cone intersection, the algorithm produces one set of amalgamated intersection polygons for each cone face. To calculate the final visual hull requires the pairwise intersection of each cone with each other cone, so for example, for a four-camera setup the number of combinations of cone-cone intersections is twelve. For n cameras this figure is $n \times (n - 1)$ intersections. Thus each cone gets intersected with $(n-1)$ others and by the end of the overall calculations, each cone face will contain $(n-1)$ sets of intersection polygons. Each of the $(n-1)$ sets must be resolved to create the final faces of the visual hull. This is what the remainder of the algorithm is concerned with. The operation of the main algorithm can be specified as follows:

```
for each cone
    extract contour
    for each other cone
        make edge-bin set (see figure 4.6b)
        for each cone face
            2D intersect with edge bins
            union of intersection polygons
            back-project
    intersect 3D
```

4.3 Calculating Visual Hull Faces.

The final stage of the process can be split into two parts: the first is concerned with recovering 3D information from the projected cone face intersections in 2D; the second with resolving the contribution that each set of (n-1) polygons makes towards each final visual hull face.

4.3.1 Recovery to 3D.

In order to compute the final visual hull faces, intersection polygons must first be projected back onto the representation of the cone face in 3D to which they belong. The process of back projection involves specifying a 3D planar representation of a cone face for the projected cone, then casting rays from the silhouette view's camera centre through each point on the intersection polygon formed within the projection of the cone face. Knowledge of the projection matrix gained earlier in the pipeline is used in this process. The rays cast thus in 3D will project back onto the 3D cone face, where the intersection of ray and plane can be obtained. To calculate the planar representation of a cone face, three points in 3D must be specified. The first is obtained from the value for the camera centre, which is obtained from the extrinsic translation component recovered during calibration. The other two points of the plane come from the two image plane silhouette contour points and the reverse of the projection matrix which defined those image points. As discussed in chapter two, world points can not be obtained directly from the operation of the inverse projection matrix on the image points it describes, but the direction in which a world point lies may be gained. Calculation of the intersection of each directional indicator (line) with the plane at infinity defines a world point. The camera centre, the two points at infinity and the cone face are all coplanar and so it is a case of obtaining the coefficients of the plane equation through application of linear algebra to the coordinates of the three points. To obtain the location of a ray projected through an image point to where it meets the plane at infinity, a portion of the projection matrix is used. Writing

$$P = [M|p_4]$$

where M is the first 3×3 submatrix of P which represents the product KR (internal calibration and rotation), the camera centre is given by

$$C = -M^{-1}p_4$$

and an image point back projected to a point at infinity, D , is given by

$$D = ((M^{-1}x)^T, 0)^T$$

The definition of each back-projected ray is achieved in the same manner, this time using the projection matrix for the camera imaging the silhouette which defined the intersection polygons and the vertices of the polygons on the image plane. Camera centre and point at infinity describe the parametric representation of a line in 3-space. Algebraic combination of cone face plane coefficients and parametric equations allows the parameter 't' to be solved which then defines the intersection of ray and face in 3D.

4.3.2 Intersection in 3D.

With $(n-1)$ sets of intersection polygons recovered in 3D, all that remains is to compute the intersection of the sets. Again the intersection calculation can be reduced from 3D to 2D as all sets of polygons for a cone face lie in the plane of that cone face. With this observation, the method employed to obtain the intersections is to first calculate the normal vector to the cone face. This is done so as to discover in which Euclidean plane to carry out the intersection. To achieve this, the absolute values of the normal vector components are examined and the largest value dropped leaving the plane in which to calculate intersection. This works because the largest normal vector component will lie furthest from the plane most closely oriented to the cone face plane. This method avoids cases such as trying to intersect in the x - z plane, for polygons lying in a vertical cone face. Intersection in xy or xz or yz is then carried out using the Generic Polygon Clipping library and then the dropped parameter is recovered by interpolation between the other two parameters using the coefficients of the cone face plane equation. This process is repeated for each cone face in each cone and the resulting intersections constitute the final visual hull faces. This marks the end of the pipeline whereby it has been explained how polyhedral models may be reconstructed from sets of images. The following section discusses details specific to the implementation structure.

4.4 Program Structure.

As stated in chapter three, the language used in the main implementation is C++. Basic structure for the program components has all extraneous header components and class declarations in a single reference header file. The functionality of classes with related behaviour is defined in three main utility files, while a relatively compact main function coordinates the parts into a whole. A basic design consideration in object oriented programming is to represent all the nouns in the description of the problem as classes, while the verbs will likely correspond to major functions. A description of the pipeline presented here might take the following form: "extract silhouette contours from the input images and represent each as a silhouette cone of cone faces. Construct a set of edge-bins composed of bins of edges for intersection in 2D. Intersect all pairs of cones to produce intersection polygons. Back project and intersect in 3D." Perusal of the class descriptions which follow will show that most nouns and verbs in the above description are represented in abbreviated form by classes and member functions. The behaviour of the classes is designed to reflect these meanings.

4.4.1 Description of Classes.

The exception to the above rule is the specification of the two classes, `vec4` and `mat4x4`, designed to represent the behaviour and functionality required of 4×1 column vectors and 4 matrices respectively. In the absence of abstract base classes in the implementation, these two classes represent the most basic objects which all other classes utilise exhaustively throughout the operation of the program. Functions in these classes form the basic tools for the system.

4.4.1.1 `mat4x4` class.

The `mat4x4` class is used to describe all matrices presented in this report, calibration K , projection P , essential E , fundamental F , rotation R , and translation T . The rank of matrix the class represents reflects the maximum rank of matrix required for transformations in 3D homogeneous coordinates. Where matrices of lower rank are created, the spare elements can simply be set to zero. Floating point precision in the elements

represents a compromise between accuracy and memory efficiency. Several basic matrix operations are defined. Matrix multiplication and addition are defined as overloaded operator functions along with the vector result of the product of a matrix and a vector. Orthogonal rotation is defined in `rot_x()`, `rot_y()` and `rot_z()` and translation in `trans()`. The transpose of a matrix is taken with `transpose()` and representation as a skew-symmetric matrix with `skew()`. More specialised matrix functions are:

- `read()` locates an ascii file representation of sixteen floating point numbers. The function reads the values into elements of a matrix.
- `projection()` calculates the projection matrix for a particular camera. The function obtains the calibration matrix and the rotation/translation transform matrix via calls to `read()` and then computes the product of the two.
- `essential()` specifies the attempt to compute the essential matrix between two cameras from the product $[T]_{\times}R$. To calculate $[T]_{\times}$, the translation between the two camera coordinate system origins is evaluated. The skew-symmetric representation of the resulting three-component vector is obtained with a call to `skew()`. Next the relative rotation between camera systems is calculated from the camera's positions in world space. The matrices representing the world transform for both cameras (RT product matrices constructed from the extrinsic parameters) are read. Only the upper 3×3 submatrices of these transforms represent the rotation component so the translational elements are set to zero. To obtain the relative rotation, the `inverse()` of one of the matrices is obtained (see `inverse` function) and the product of this inverse and the other matrix is evaluated. The result describes relative rotation between the two systems.
- `fundamental()` uses the relation $F = K'^{-T}EK^{-1}$ described in chapter two to calculate F between two cameras from the calibration matrices for the cameras and the essential matrix. This involves reading in calibration matrices which are inverted and one inverse is also transposed. the product of the two and the essential matrix describes F for the two views.
- `inverse()` incorporates two numerical techniques taken from (16). Calculating

the inverse involves decomposing the matrix into lower and upper triangular matrices (LU decomposition done by the function `ludcmp()`) which define a system of equations solvable by forward and back substitution (in the function `lubksb()`). The `inverse()` function processes the matrix column by column using its LU decomposition passed to the `lubksb()` function.

4.4.1.2 **vec4 class.**

The `vec4` class is the abstraction of a 4×1 column vector and is used primarily to represent homogeneous points in 3-space. Thus the members are floating point fields `x`, `y`, `z`, `w`. As for matrices of lower rank in the `mat4x4` class, vectors with less components are still created as objects of the `vec4` class with redundant components being zeroed. In other parts of the program image point (homogeneous or otherwise) use the class with the spare components being used to carry flags and identifiers. Basic vector operations are defined after constructor and initialiser functions. `length()` calculates the length of the vector by Pythagoras and `norm()` normalises the vector. `cross()` calculates the cross-product and `vecIntAngle()` calculates the angle between two vectors. `getDist()` calculates the distance between two vector endpoints. `vPrint()` prints a vector to `stdout` along with a string. Various operators are overloaded to allow addition, subtraction and the dot-product of vectors and between vectors and matrices. The more specialised vector functions are:

- `intersectEdgeLine()` takes a 2D polygon edge represented by two endpoints, and the coefficients of a 2D line in slope intersect form as input. The slope intersect form of the line between the edge endpoints is computed then the cross-product between the two lines is calculated with a call to `getLineIntersect()`. The result is a `vec4` representing the point of intersection.
- `quicksort()` and `partition()` are the implementation of the efficient quicksort algorithm for sorting an unordered list. The function is implemented to sort a list of vectors by one of the vector components. While the list is unordered, the `quicksort()` function recursively calls itself to work on two parts of the list, with the split point based on the value returned by the `partition()` function. The `partition()`

function also does the physical swapping of elements in the list.

- `camCoords()` calculates the coordinates of a world point in the camera coordinate frame by reading an RT product matrix from a file and returning the vector result of the product of this matrix and the vector representation of the world point.
- `imgCoords()` does the same as `camCoords()` with the additional operation of the calibration matrix for that camera to yield the homogeneous $(u, v, 1)$ image point representation of a world point.
- `getCentre()` computes the location of the centre of projection of a camera in world coordinates from its extrinsic parameters. The RT matrix is read in and the upper 3×3 submatrix representing rotation is extracted and its inverse computed. The product of this inverse and the original RT matrix results in a 4×4 matrix equal to the identity matrix except for the upper three elements in the final column which contain the values for the camera centre position. These elements are written to a vector and this result returned.
- `getTcamA_B()` calculates the translation between two camera centres in world coordinates as a vector from the difference between the camera centres.
- `exactImgPts()` is a function which uses the actual model data to calculate where known world points project to a camera's image plane for its projection matrix. This is used to obtain point correspondences in the absence of accurate values for F . These points are introduced into the Structure and Motion library (14) to obtain accurate version of F .

Six classes are defined to represent the behaviour and functionality required to model edge bin construction, intersection in 2D and calculation of visual hull faces that form the stages of the PVH algorithm. The classes are `Edge`, `EdgePair`, `Polygon`, `Bin`, `EdgeBinSet` and `ConeFace`.

4.4.1.3 Edge class.

This class is a simple class designed to represent an edge in 2D or 3D. It consists of the projected vector representation of two endpoints, a constructor and accessor functions

to initialise/ update and retrieve the member fields.

4.4.1.4 EdgePair class.

EdgePair class inherits all functionality from class Edge. An EdgePair object holds a pair of Edge objects which are those two edges which meet at the vertices in the silhouette contour. Specification of this class of object allows access to adjacent endpoints in the list of cyclically ordered points during construction of the edge-bins. The class consists of two protected Edge fields, a constructor and accessor functions.

4.4.1.5 Polygon class.

The Polygon class mimics an arbitrary sized polygon as a list of points (as vec4 objects). Functionality is built into the class so that a polygon may be initialised in a completed form or constructed progressively. In order to achieve this, objects of the class are represented by an extensible template class deque object from the Standard Template Library. Also specified are a constructor and various accessor functions to initialise, enlarge and probe the internal state of the object.

4.4.1.6 Bin class.

The Bin class represents an edge bin defined by two α (2D line gradient) values, α_1 and α_2 for the start and end bounds of the bin, and an extensible list of Edge objects held as a template class vector object. Each Bin object has an internal mechanism triggered externally on completion of bin construction which orders the Edge objects in the bin by increasing distance from the projected cone epipole. The result is a pointer to a list of Edge objects.

4.4.1.7 EdgeBinSet class.

Class EdgeBinSet defines the behaviour required to construct a set of edge-bins for each silhouette and also to perform the construction of the 2D intersection polygons that result from each cone-silhouette intersection. The class inherits from the Bin class and consists of several protected member fields which are assigned values via accessor

functions and three main class functions. Those functions order a silhouette's points by the alpha values (`orderPointByAlpha()`), construct a set of edge-bins (`makeEdgeBinSet()`) and compute a cone-silhouette intersection (`ConeSilhtIntersect()`). The function `OrderPointsByAlpha()` is called at an early stage in the `makeEdgeBinSet()` function. Each point in the cyclic list for the silhouette and the projected epipole form a line. The gradient (alpha) of this line is calculated and assigned to the 'w' field in the `vec4` representation of the point. When all points are treated thus, the whole list of points is sorted by their alpha values via a call to `quicksort()`. The `makeEdgeBinSet()` function begins by initialising a protected class member field called `vWS` which holds the value for the camera's centre which is responsible for the silhouette being used to construct a set of edge bins. Next the number of Bin objects is specified based on the number of points in the silhouette. The silhouette points are ordered by alpha as explained above. A short routine then creates an `EdgePair` object for each point so as to associate the two edges which emanate from that point in the cyclic list with each point in the alpha sorted list. This allows the immediate examination of the alpha values for the opposite endpoints of each alpha point's associated edges in the contour. As explained earlier in the chapter, edge-bins are constructed in order of alpha values for points in the alpha list using the algorithm in figure 4.6 to govern edge selection criteria for each bin. The set of bins created and the alpha ordered list initialise protected fields of the `EdgeBinSet` object along with integer fields to specify the extents of those fields. The `ConeSilhtIntersect()` function calculates intersection polygons in 2D, cone face by cone face for each cone. An orderly set of events occurs for each call of this function: a set of intersection polygons is defined for each cone face; the amalgamation of these polygons is obtained; the result is projected back onto the 3D cone face. The latter two events are specific to each individual cone face and so the functions which cause these activities are held in the `ConeFace` class. To perform the intersection, the particular silhouette cone object and set of edge-bins are retrieved. The first cone face is located within the set of bins by testing bin bounds alpha values to see if the cone face lower bound falls between the bin bounds. When the correct bin is found intersection proceeds by traversing bins in the direction of the upper cone face bound, constructing polygons on the way as described in the discussion earlier in the chapter on intersection

polygon construction. As discussed, to build each polygon the edges which initiate it are added in order of occurrence and points which close a polygon are added before and after the first two points. Lines used to form the intersection points are selected based on whether a bin bound or a cone face bound will be encountered next in the progressing construction. When a set of polygons has been built for each face, the set is unioned and the union set is back-projected (both ConeFace functions).

4.4.1.8 ConeFace class.

The ConeFace class contains the parameters and describes the behaviour of each silhouette cone face. Each ConeFace object is described by the line equation, slope of each bound, and image points responsible for the face. Further protected fields consist of a temporary store for each set of intersection polygons specified as a vector template, a value for the world coordinate plane lying closest to the normal to the cone face, and the coefficients of the plane equation that represents the face. The latter two pieces of information are used in back-projecting points calculating intersections in a plane in 3D. Besides accessor functions affording an interface to the protected members, the two main class functions are the function to amalgamate intersection polygons, `getUnionOfBinPolys()`, and the function to project polygons back to the 3D cone face, `backProject()`. The `getUnionOfBinPolys()` function is called from within the `ConeSilhouetteIntersect()` function on completion of the creation of intersection polygons for a cone face. The function utilises the Generic Polygon Clipping library (10) by looping over each polygon. The first and second polygons are clipped with the UNION operation and the result is then clipped with the next polygon in the loop. In this manner an overall union of all polygons in that cone face is produced. The `backProject()` function calculates the equation of the plane coefficients representing a cone face by obtaining three world points on the cone face, as described earlier in the chapter. One point is the cone (camera) centre held in the `vWC` field for the object representing the silhouette cone (see below). The other two points are where back-projected rays through the image points defining the face meet the plane at infinity. The parametric representation of a ray through each intersection polygon point is obtained in the same manner. The parameter 't' is then calculated and substituted into the line equations to define a

point in 3-space. These points are written out to a temporary file created at the call to `ConeSilhtIntersect()`.

4.4.1.9 SilhtCone class.

A single class is used to carry out the functions involved in image processing the input images and defining the behaviour of a silhouette cone produced thus. The whole image processing section of the pipeline is achieved using one function in the class, `getSilht()`. A call to this function causes a single sequence of execution which results in the extraction of a silhouette contour. The function draws heavily on the functionality provided by various VTK classes. The first operation is to allocate the value of the particular camera centre to a protected field via a call to the `getCentre()` function, with a parameter identifying the camera in the function call. VTK graphical initialisation objects are then created to allow the image processing objects to be rendered. The image processing pipeline begins by reading an image into an object of an all-purpose reader class `vtkPNMReader`. The output from this object is passed to the marching squares algorithm via an object of class `vtkMarchingSquares` where the image is contoured with the specification that a single contour be produced. The calculation is forced by attaching the marching squares filter to a mapper object which is then rendered via an actor object. The contour result produced is an exhaustive list of point pairs representing isoline segments. To prepare the data for input to the triangulation procedure, every second point is assigned to a list of objects of class `vtkPoints`, a basic point representation class. This requires that a number of sequential calls backward through the class hierarchy is made in order to access raw data (VTK is designed to hide such details from the user). Triangulation and decimation are performed by the `vtkDelaunay2D` and `vtkDecimation` classes. A set of parameters governing the behaviour of operation of these algorithms is specified by a set of values chosen on the basis of a command line parameter. The output from decimation is a set of tuples of four points, representing a count of the points in each output (which is three in the case of triangles) and the list of points itself. Shared vertices in the output mesh represent redundant data in the final contour. The next part of the function is to make the contour data usable to the rest of the program. A loop is set to transfer the decimated output in VTK format to

a format more amenable to the rest of the program. The transfer proceeds by adding non-duplicate points to a vector object. Duplication testing involves constructing a frequency array which is the size of the whole number of points output by decimation. Each point that is accepted has its identifying location in the array increased from zero to one. The test for acceptance of a point id that its identifying location value in the array is zero. In this way duplicate points are discarded. The final task of the `getSilht()` function is to see that the contour points are ordered cyclically. This is achieved by passing the output points to the subsidiary function, `orderPointsCyclically()`. As discussed earlier in the chapter, cyclical ordering of points is achieved by testing a point against all active points in the list by relative distance from the current point. The nearest point is added to the list to front or rear depending on the vector angle between the current/next and current/previous point pairs. A further constraint on addition of points is that the distance calculated is above a certain threshold value. This excludes points which are too closely spaced on the contour. The final list of points is in the form of a vector. The list forms a protected field in the `SilhtCone` class which in turn specifies the size of a protected integer field. At the end of construction of the list, the size of this variable is used to allocate a chunk of memory which holds a sequence of `ConeFace` objects. In this manner the class is used to represent the silhouette contour and silhouette cone gleaned from the input image. Accessor functions allow other parts of the program to obtain information about the class members. The first capacity in which cone behaviour is required is in the function `projectCone()`. This function is called early on in 2D intersection in `ConeSilhtIntersection()` and has the effect of projecting a 3D silhouette cone onto a 2D image plane. This is done by the function taking each of the ordered silhouette points in the class's protected field 'cycPts' and calculating the line produced from the operation of the relevant fundamental matrix on the point. Values produced in this process are used to initialise member fields in each `ConeFace` object in the cone: two `vec4` objects representing the coefficients of each cone boundary line in line-intersect form, an alpha value associated with each line, and the image points used to form the cone. After sets of intersection polygons are produced at each cone face, the final expression of cone behaviour is exhibited in the `poly3DIntersect()` function. This function calculates the intersection of the sets of

intersection polygons for all the cone faces in a cone. The $n \times (n - 1)$ files which were created to hold all the amalgamated intersection polygons for each face are read into a four-dimensional pointer list of `vec4` objects which represents the number of cones in one dimension and the number of cone faces for each cone, the number of polygons in each face, and the number of vertices in each polygon in the other three. With this data structure initialised, the plane to do intersection in is chosen from the field set during back-projection and the set of polygons for each face is sequentially intersected in 2D using the Generic Polygon Clipping library. This causes each visual hull face to be clipped back into its final form. As with the polygon amalgamation stage, intersection proceeds by the first set of intersection polygons being clipped against the second with the result being intersected with the next set revealing the final face. Each final face is written out to a file in `vtk` format to enable a constructed polygon renderer to display the results.

4.4.1.10 Main Function.

The `main()` function coordinates the subsidiary parts of the program after initialising a set of global variables based on command line parameters introduced. A first loop initialises a number of silhouette cone objects as specified from the command line. On completion of this loop a number of silhouette contours reside in memory and the dimensions of lists and memory allocations for associated objects are specified. A second loop creates $n \times (n - 1)$ edge bin sets and intersects each cone with each silhouette. As described previously, the functions associated with these actions causes sets of amalgamated back-projected intersection polygons to be produced for each cone face. A final loop calls the 3D intersection function for each cone so that the final hull faces are produced

Chapter 5

Results.

This chapter gives details of output obtained at each stage in the operation of the implemented system. The model scene setups illustrated in figure 3.2 form the basis for all the results presented in this section.

5.1 Model Scene and Calibration.

As discussed in chapter 3, the test scene setups designed in VTK gave rise to spatial data recording the transformations relating to each of the specified cameras. These data were used to constrain the accuracy of the same parameters recovered during the calibration procedure. The values below show the results obtained from the test scenes in the left hand matrices, and those obtained from the Camera Calibration Toolbox in the right hand matrices.

$$\begin{array}{l} \text{camera 1} \begin{bmatrix} 2359 \\ 1766 \\ 3012 \end{bmatrix} : \begin{bmatrix} 2279 \\ 1684 \\ 3009 \end{bmatrix} \quad \text{camera 2} \begin{bmatrix} 4046 \\ 1766 \\ 604 \end{bmatrix} : \begin{bmatrix} 3982 \\ 1701 \\ 623 \end{bmatrix} \\ \text{camera 3} \begin{bmatrix} -569 \\ 1766 \\ 3269 \end{bmatrix} : \begin{bmatrix} -648 \\ 1686 \\ 3266 \end{bmatrix} \quad \text{camera 4} \begin{bmatrix} -2649 \\ 1766 \\ 1189 \end{bmatrix} : \begin{bmatrix} -2713 \\ 1700 \\ 1205 \end{bmatrix} \end{array}$$

5.2 Image Processing.

Results achieved from the image processing pipeline are in the form of successfully contoured and decimated output images. Figure 5.1a and b shows an example set of input images, the contours extracted by the marching squares algorithm and the results of the decimation procedure. Figure 5.1 c, d and e shows the silhouette contour which results from the point ordering algorithm after decimation. Figure 5.1e shows the result of poorly chosen decimation parameters.

5.3 PVH Algorithm.

Results obtained from the operation of the construction of edge-bins were fed through the pipeline and cone-silhouette intersections were computed. Output produced after amalgamation of intersection polygons is shown in figure 5.3.

Back-projection and intersection in 3D of the 2D intersection polygons produces a list of polygons in 3D, These are the faces of the final visual hull models. Figure 5.4 shows various views of the two models reconstructed from three and four camera views.

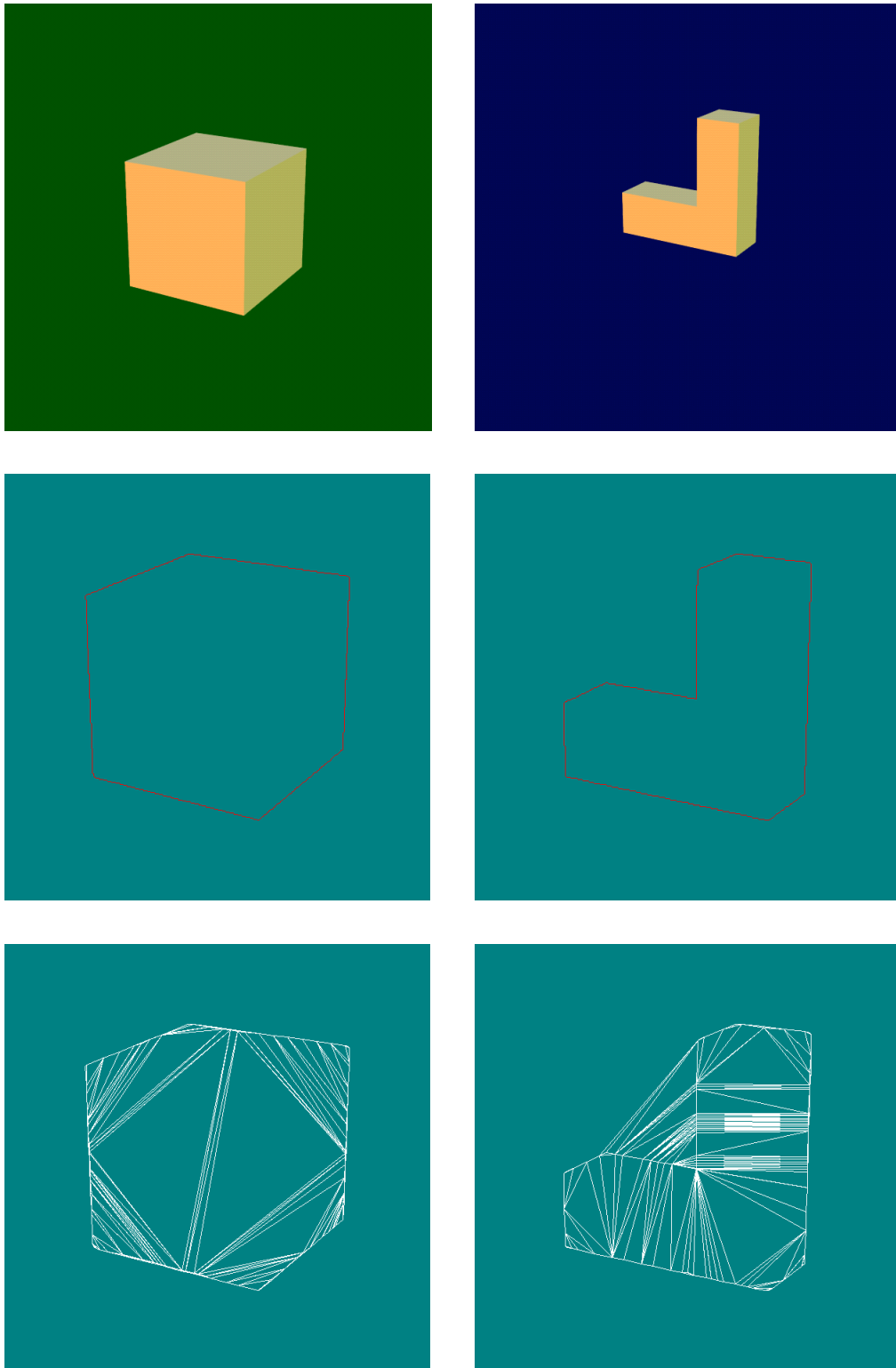


Figure 5.1: Sample images, contours and decimations

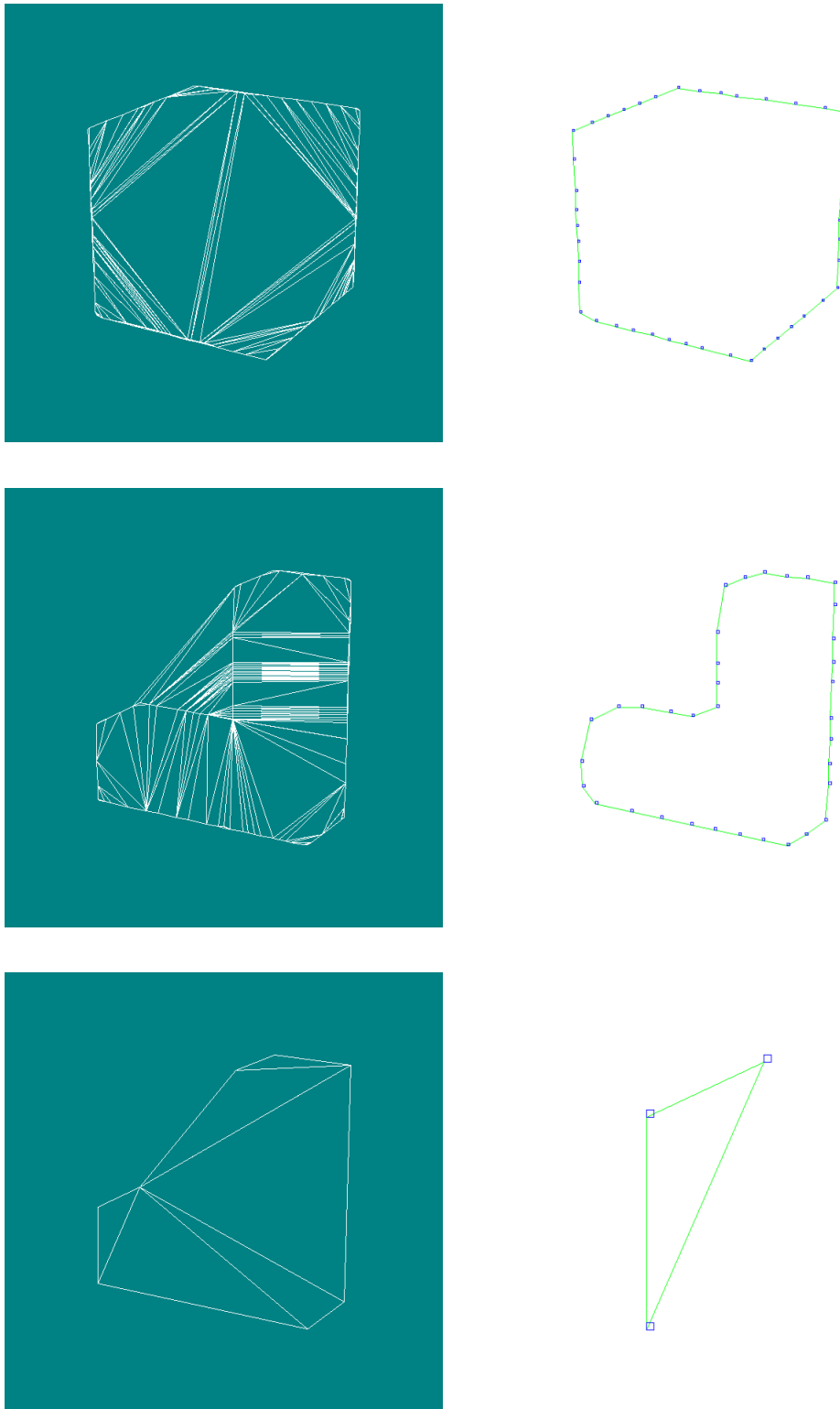


Figure 5.2: Sample decimations and silhouette contours

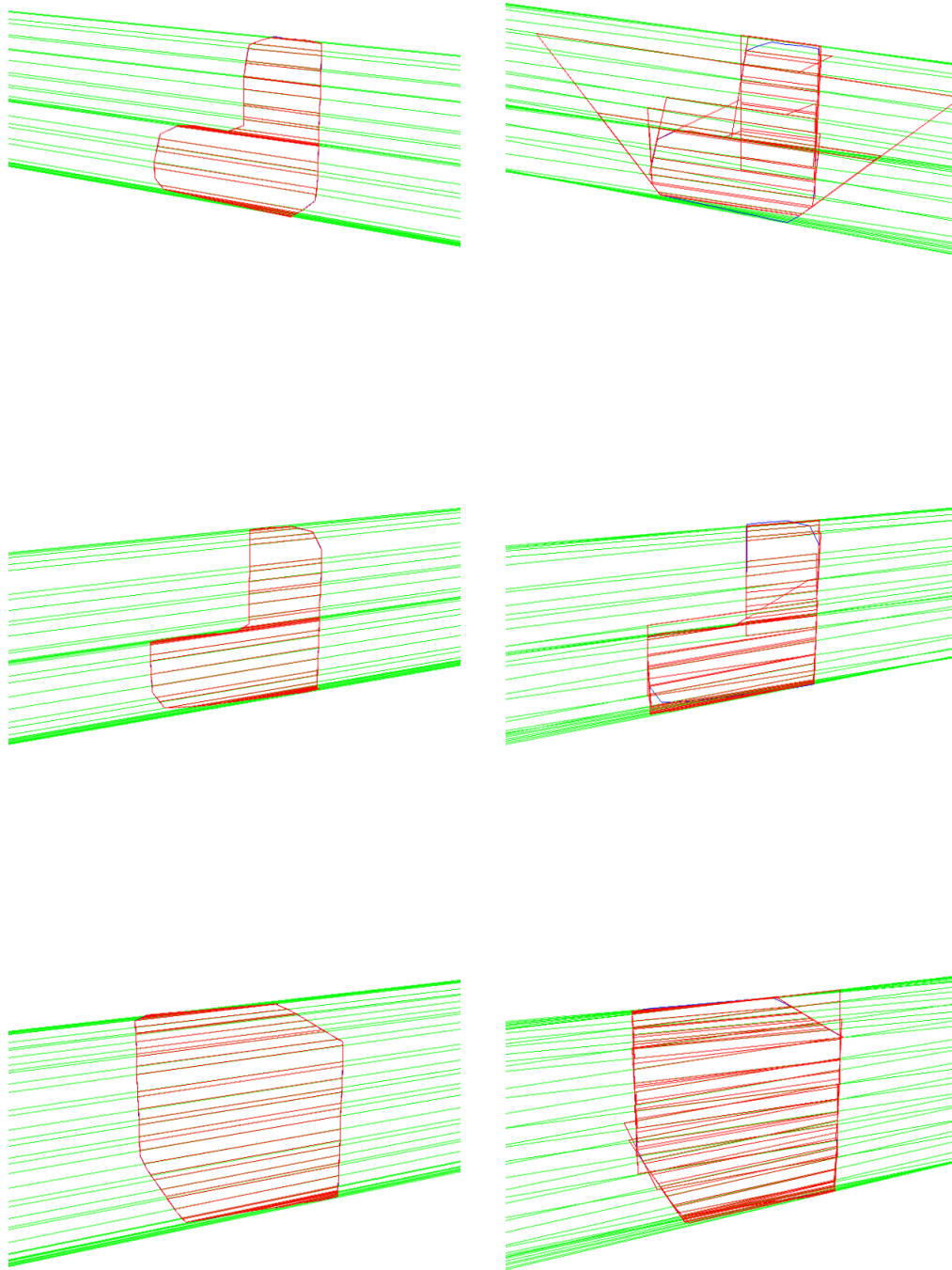


Figure 5.3: Silhouette cones and intersection polygons produced by accurate (left) and inaccurate (right) fundamental matrices

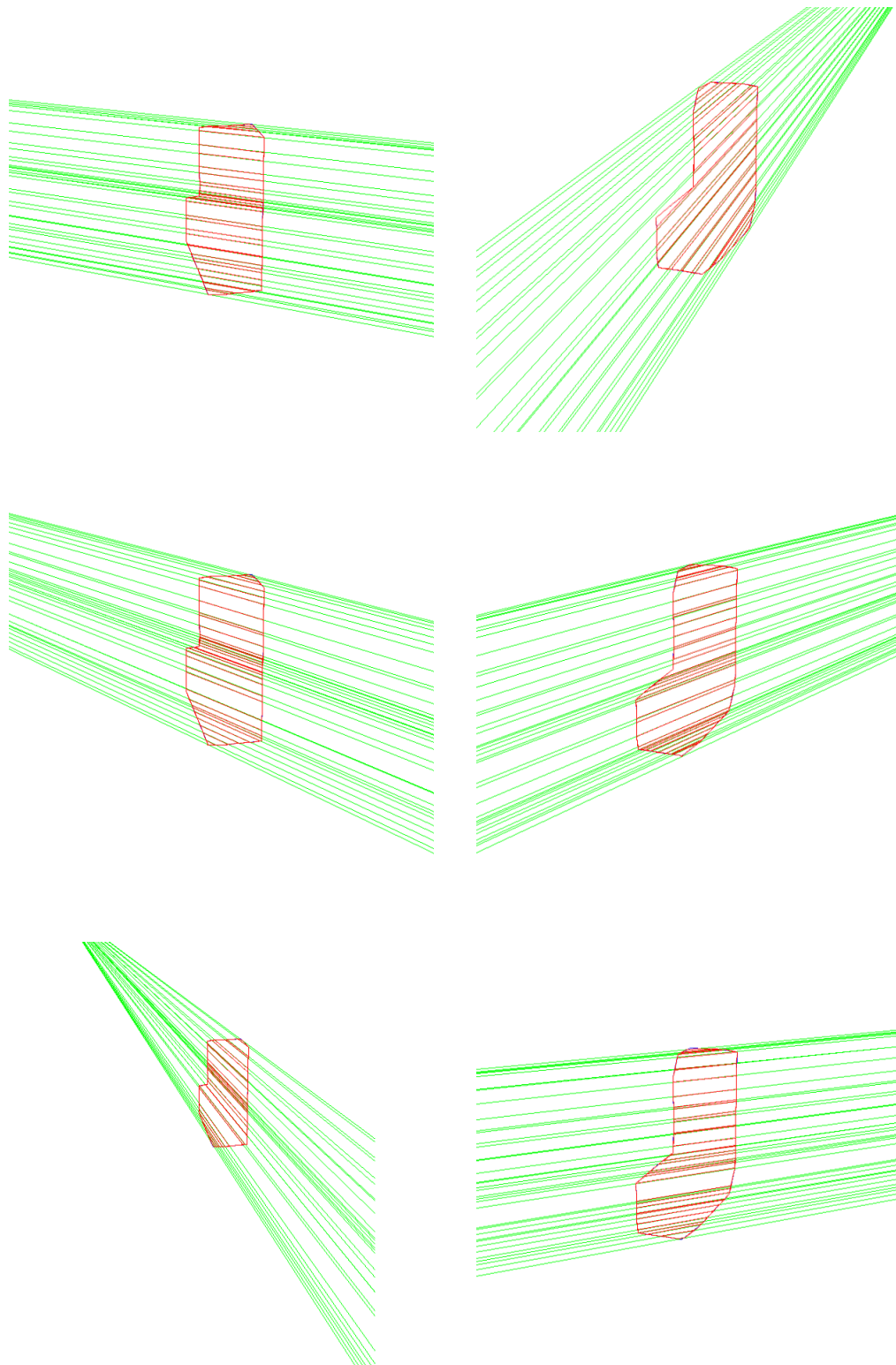


Figure 5.4: intersection polygons and silhouette cones from 4-camera view.



Figure 5.5: Different views taken of visual hull models for four cameras (left) and three cameras (right).

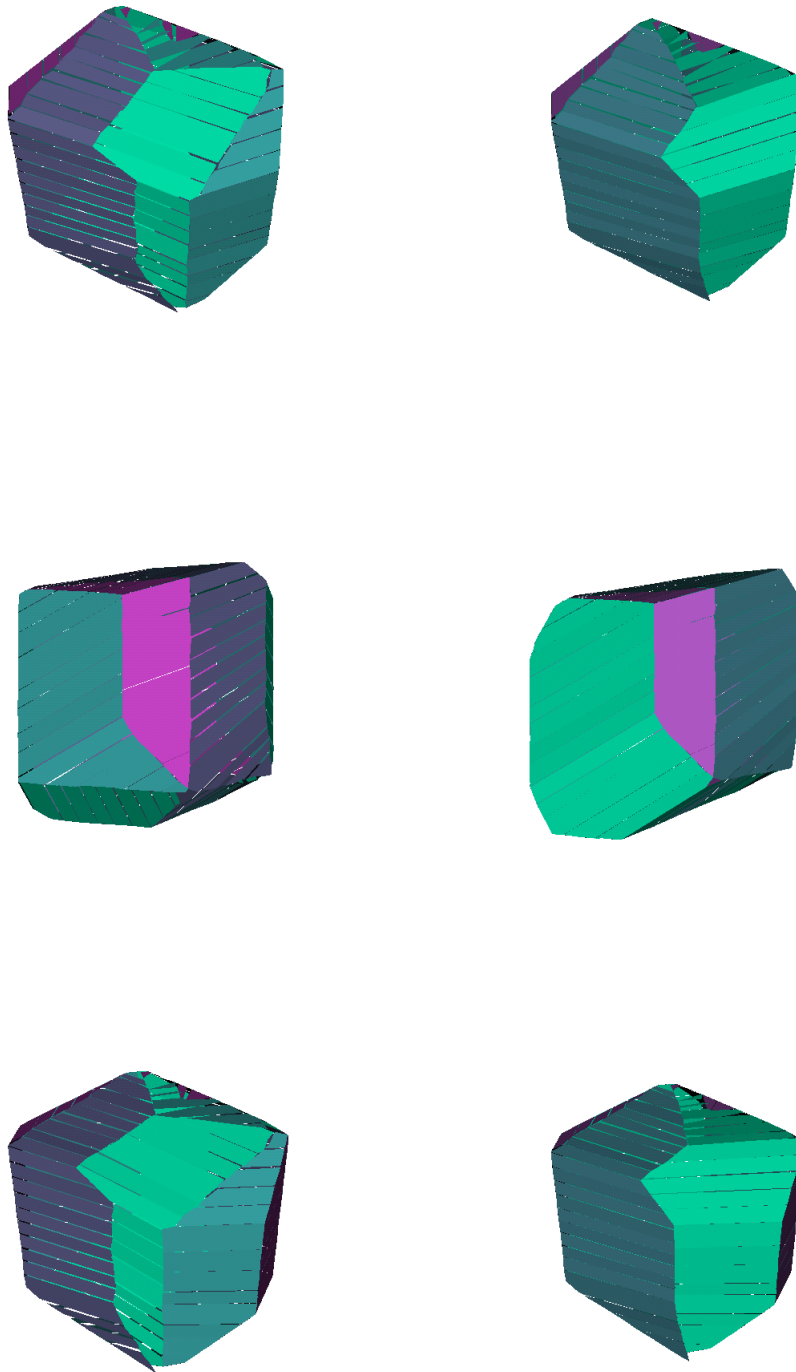


Figure 5.6: Different views taken of visual hull models for four cameras (left) and three cameras (right).

Chapter 6

Discussion and Further Work.

6.1 Discussion.

Results gained from the model scene and camera calibration stages of the pipeline at first glance appear to suggest that particularly accurate values may be obtained through the application of the calibration toolbox. The values corresponding to the positions of the camera centres computed in calibration are accurate within a radius of approximately 80mm. An 80mm positional error is small compared to the 4m distance at which the camera lies from the model. However, the images in figure 5.2b show the level of error introduced when even small errors such as these are introduced into the calculations. In the absence of another method to test the accuracy of the elements in the calibration matrix, little in the way of conclusion about the results may be drawn. The fact that the elements relating to focal length (K_{11} and K_{22}) are almost identical for a virtual camera would suggest accuracy. The values obtained for the principal point are also well constrained (the image size used for the inputs is 600×600 , therefore the principal point is at (300, 300) corresponding well to the (299, 299) calculated by the toolbox). The stage of the pipeline which deals with image processing shows that the contour extractions computed by the marching squares algorithm give an accurate representation of the input silhouette. Also apparent between the left and right views in figure 5.1c and d is the case that a slight loss in fine detail of the geometry occurs while the cyclic ordering algorithm is selecting points. To improve this, a small optimisation

could be introduced to retain feature vertices based on the vector angle between points, currently used in the algorithm to maintain cyclic ordering in selection of initial points in the point set. The results in figure 5.1e show the loss of coherence in the contour which can result in attempting to decimate to too large an extent. The set of results corresponding to values computed for the fundamental matrix in table 5.4 again show minor differences between results for the sets of data from the source model and those recovered using the results of calibration. However the artefacts apparent in figure 5.3 suggest that significant error is introduced from even minor inaccuracies in the values of F used. Results shown in figure 5.4 represent the final output of the pipeline. Subjective analysis of the figure reveals a fairly good overall approximation to the initial models, particularly in the models produced from sets of four input images. The fit is slightly poorer for the three-camera views as expected, but it is still possible to ascertain the original form of the models which produced the images. The number of artefacts in all images is fairly pronounced and failure to resolve these issues has resulted in the sparsity of representations produced and lack of quantitative analysis. Time taken in resolving many of the errors discussed here and in pinpointing memory access violations from inexperience with the implementation language contributed hugely to this situation. Despite these factors, it is possible to conclude that this approach to model reconstruction is capable of producing good overall representations of the generative models, especially using sets of images taken from four cameras. Significant errors compounded from the combination of minor errors in the pipeline would suggest that accurate rendering of models via this technique proves difficult because the process attempts to recover information about exactly where the object hull *is*, when it is always simpler to extract information about where the object is *not*.

6.2 Further Work.

In order to extend the project, it would be desirable to fully remedy the implementation so as to obtain good quantitative data. Beyond this, as discussed in chapter 3, the image extraction process can be extended to operate on more complicated images. Adding a robust background subtraction algorithm could see the system extended to

being implemented on a real camera setup and by spreading the assimilation process to multiple hosts, it would be possible to assess the capabilities of the system for the capture of dynamic scenes with temporal constraints.

Bibliography

- [1] Bouguet, J. (2002). Camera calibration toolbox for matlab.
- [2] Bowyer, A. (1981). Computing dirichlet tessellations. *The Computer Journal*.
- [3] Dyer, C. R. (2001). *Volumetric Scene Reconstruction from Multiple Views*. Kluwer, Boston.
- [4] EPCC (1996). Delaunay triangulation.
- [5] G. K. M. Cheung, T. Kanade, J.-Y. B. and Holler, M. (2000). a real time system for robust 3d voxel reconstruction of human motions. In *Proc. Computer Vision and Pattern Recognition Conf.*
- [6] H. Noborio, S. Fukada, S. A. (1988). Construction of the octree approximating three-dimensional objects by using multiple views. *IEEE Transactions on Pattern Analysis and Machine Intelligence*.
- [7] Hartley, R. and Zisserman, A. (2000). *Multiple View Geometry in Computer Vision*. Cambridge University Press.
- [8] Kutulakos, K. N. and Seitz, S. (1998). A theory of shape by space carving. In *Technical Report TR692, Computer Science Dept., U. Rochester*.
- [9] Laurentini, A. (1994). The visual hull concept for silhouette-based image understanding. *IEEE Transactions on Pattern Analysis and Machine Intelligence*.
- [10] Murta, A. (1999). A general polygon clipping library.
- [11] Owens, R. (1997). Camera calibration.

- [12] S. Moezzi, L-C. Tai, P. G. (1997). Virtual view generation for 3d digital video. *IEEE Multimedia*.
- [Snow et al.] Snow, D., Viola, P., and Zabih, R. Exact voxel occupancy with graph cuts. pages 345–353.
- [13] T. Kanade, P. W. Rander, P. J. N. (1997). Virtualized reality: Constructing virtual worlds from real scenes. *IEEE Multimedia*.
- [14] Torr, P. (2002). A structure and motion toolkit in matlab.
- [15] Vatti, R. (1992). A generic solution to polygon clipping. *Comm. ACM*.
- [16] W. H. Press, S. A. Teukolsky, W. T. V. and Flannery, B. P. (1992). *Numerical Recipes in C*. Cambridge University Press.
- [W. Lorensen] W. Lorensen, H. E. C. Marching cubes: A high resolution 3d surface reconstruction algorithm. In *Computer Graphics*.
- [17] W. Matusik, C. Buehler, L. M. (2001). Polyhedral visual hulls for real-time rendering.
- [W. Schroeder] W. Schroeder, J. Zarge, W. L. Decimation of triangle meshes. In *Computer Graphics*.
- [18] Watson, D. F. (1981). Computing the n-dimensional delaunay tessellation with application to voronoi polytopes. *The Computer Journal*.
- [19] Will Schroeder, K. M. and Lorensen, B. (1998). *The Visualization Toolkit*. Prentice Hall.