3D-MODEL RECONSTRUCTION OF COMPLEX OBJECTS FROM MULTIPLE IMAGES

A DISSERTATION

Submitted in partial fulfillment of the requirements for the award of the degree

of

MASTER OF TECHNOLOGY

in

INFORMATION TECHNOLOGY

By

RAVIKANT AMBADAS GEDAM

DEPARTMENT OF ELECTRONICS AND COMPUTER ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY ROORKEE
ROORKEE - 247 667 (INDIA)

JUNE, 2005

CANDIDATE'S DECLARATION

I here by declare that the work, which is being presented in this dissertation, entitled "**3D-MODEL RECONSTRUCTION OF COMPLEX OBJECTS FROM MULTIPLE IMAGES**", in partial fulfillment of the requirements for the award of the degree of **Master of Technology** with specialization in **Information Technology**, submitted in the Department of Electronics and Computer Engineering, Indian Institute of Technology Roorkee, Roorkee, is an authentic record of my own original work carried out from July 2004 to June 2005, under the guidance and supervision of **Dr. Kuldip Singh**, Professor, Department of Electronics and Computer Engineering, Indian Institute of Technology Roorkee, Roorkee, and **Dr. Sumit Gupta**, Lecturer, Department of Electronics and Computer Engineering, Indian Institute of Technology Roorkee, Roorkee.

I have not submitted the matter embodied in this dissertation for the award of any other degree.

Date: 30-06-2005

Place: Roorkee

[illegible]

(RAVIKANT AMBADAS GEDAM)

CERTIFICATE

This is to certify that the above statement made by the candidate is correct to the best of my knowledge and belief.

[illegible]

Dr. KULDIP SINGH
Professor,
E & CE Deptt.,
IIT Roorkee, Roorkee
India – 247 667

[illegible]

Dr. SUMIT GUPTA
Lecturer,
E & CE Deptt.,
IIT Roorkee, Roorkee
India – 247 667

ACKNOWLEDGEMENT

It is my privilege and pleasure to express my profound sense of respect, gratitude and indebtedness to my guide, **Dr. Kuldip Singh**, Professor, Department of Electronics and Computer Engineering, Indian Institute of Technology Roorkee, Roorkee, for his inspiration, guidance, constructive criticisms and encouragement throughout this dissertation work.

I would like to express deepest gratitude to my co-guide, **Dr. Sumit Gupta**, Lecturer, Department of Electronics and Computer Engineering, Indian Institute of Technology Roorkee, Roorkee, for his inspiration, guidance and support throughout the course of the work. His insights on computer vision have given many fresh perspectives needed to solve problems.

I express my sincere thanks to **Dr. A. K. Sarje**, Professor and H.O.D., Department of Electronics and Computer Engineering, Indian Institute of Technology Roorkee, Roorkee, for providing necessary facilities.

I am greatly indebted to all my friends, who have graciously applied themselves to the task of helping me with ample morale support and valuable suggestions. Most of all I would like to thank to my family. My parents provided me a perfect environment for my studies and supported me throughout. Finally, I would like to extend my gratitude to all those persons who directly or indirectly helped me in the process and contributed towards this work.

[illegible]

(Ravikant Ambadas Gedam)

ABSTRACT

This thesis is primarily concerned with the design and construction of image-based object reconstruction system from multiple silhouette images taken by a natural camera. We briefly explore two techniques of object reconstruction, voxelization and reconstruction based on SOR properties.

We propose a view dependent image-based method to recover a 3D-model of generic complex object. We discuss the problem of reconstruction to compute geometric information by finding corresponding feature points among multiple reference images. We exploit the geometry of single axis motion along with the surface of revolution to acquire the surface properties of object from the sequence of multiple homologous images. We further extend our technique to comply with non-homologous images.

The constrained and controlled environment will enable us to avoid the need for actual camera calibration.

CONTENTS

CHAPTER 1. INTRODUCTION

1.1 Computer Vision

Computer vision is the science of making computer to see, i.e. giving human like visibility to computer. It has received multiple accolades as a key phenomenon that will change significantly, the way humans interact with computers/devices of tomorrow. As a field it is an intellectual frontier. Like any other frontier, it is exciting and interesting.

Computer vision's great trick is in extracting meaningful information of the real world from pictures or sequences of pictures. For example, extracting information like, what object is shown in the picture or what the shape of object is? The information that users seek can differ widely between applications. For example, a technique known as structure from motion makes it possible to extract a representation of what is depicted and how the camera is moved from a series of pictures. People in the entertainment industry make use of these techniques to build three-dimensional (3D) computer models of buildings, typically keeping the structure and throwing away the motion. These models are used where real objects cannot be used. For instance, virtually they can be set fire to or blown up. Good, simple, accurate and convincing models can be built using various computer vision techniques [11].

Computer vision is the subject of research for various researchers. The subject itself has been around since the 1960s, but it is only recently that it has been possible to build useful computer systems using ideas from computer vision. Computer vision tries to provide artificial systems with a broader range of autonomy and action capabilities, but this can not be achieved without the key tasks: object detection, object reconstruction, object recognition and scene interpretation. These are different areas in computer vision by which it is heading towards making computer to see.

The human visual system performs these tasks very efficiently, processing several sources of implicit information (like shape, color, motion, etc.) from previous experience and learning. Due to increase in the processing capability of computer and advancements in information storing and retrieval, it is possible to make computer to perform such complex tasks.

1.2 Overview

Object reconstruction has been the subject of research in recent year. The problem is of considerable importance in the various industrial areas; such as Mobile Robotics, Virtual Reality, Tele-Shopping, Entertainment and View-Invariant Recognition. Many application environments enable the user interact with the models. In which the user can rotate, scale, even deform the models; observe the models under different lighting conditions; change the appearance (color, material, etc.) of the models; observe the interaction of a model with the other models in the environment. Therefore, the geometric properties of object must be clearly and robustly defined.

Recent advances in computer vision make it possible to acquire high resolution 3D-models of scenes and objects [1] [11]. However, reconstruction of a complex rigid objects from its two dimensional (2D) images is still a challenging computer vision problem under general imaging conditions. Without *a priori* information about the imaging environment (camera geometry, lighting conditions, object and background surface properties, etc.), it becomes very difficult to infer the 3D structure of the object from its images. For practical purposes, the problem can be simplified by using controlled imaging environments.

In general, given a real world object, the reconstruction system of object involves the problem of finding geometric shape and surface properties of an object. Figure 1.1 shows the typical computational steps of object reconstruction.

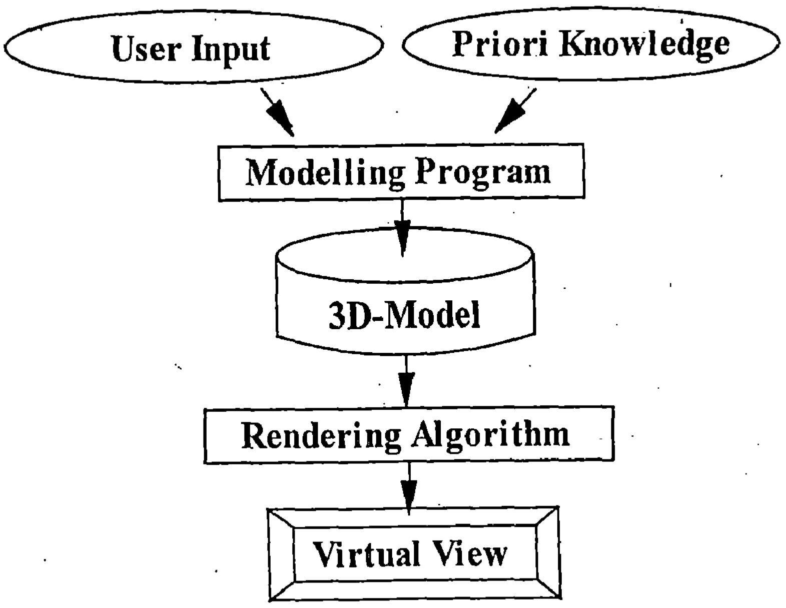


Figure 1.1: Typical computational steps of object reconstruction.

The most common way of creating 3D-models is manual design. This approach is suitable for the creation of the models of non-existing objects and mostly used in Multimedia and Virtual Reality applications. However, it is cost expensive and time consuming. Furthermore, the accuracy of the designed model for a real object may not be satisfying. Therefore, the efforts are being made towards finding techniques to automate the process of reconstruction under a simple but controlled environment.

Usually, these techniques recover the geometry of the real world scene and then render it into the desired virtual view as depicted in Figure 1.1. Such techniques are broadly categorized in two approaches, namely, object-space rendering and image-space rendering.

1.2.1 Object-space rendering

In object-space rendering, the geometry of real objects is acquired using active scanning systems that capture directly 3D data [10]. Therefore, these techniques are also referred as *active* techniques. Such systems are constructed using expensive equipments such as laser range scanners, structured light; touch based 3D scanners, or 3D digitizers. In most of these active scanning systems, the texture of the model is not captured while the geometry of the object is acquired precisely as a set of points in the 3D space. This set can then be converted to polygonal model representations for rendering.

1.2.2 Image-space rendering

In image-space rendering, the model of a real object is reconstructed from its real 2D images [1] [4] [6] [8]. These methods are also known as *passive* methods or *image-based* methods. Even using an off-the-shelf camera, considerably realistic looking models with both geometry and texture is reconstructed.

Unlike *active* methods, the *passive* methods are not expensive as they do not require very expensive equipments. Another advantage is that the shape information, such as surface normal and curvature, can be easily obtained. In addition to this, the texture properties of model can be easily captured which is very important for making realistic model. The only limitation to these techniques is that, the very complex objects can not be obtained with greater accuracy. This is due to the fact that, the world contains wide variety of objects with distinctive shape characteristics.

1.3 Problem Statement

The objective is to create curvilinear, texture mapped, 3D-models from multiple homologous images with no prior internal knowledge about the shape or topology.

A good amount of work in the reconstruction of 3D-model using image-based modeling has been reported in [1] [4] [6] [8] [11]. Various techniques have been implemented, analyzed and explored extensively. The technique used in this thesis requires multiple images of the object to be modeled from different views. As such, many requirements are set as given below:

1) As a low cost solution is desired by any business and the layman, the technique should be simple, flexible, and inexpensive.
2) It should not require any specialized hardware. Most of the image-based technique uses a simple natural camera to acquire the images of the object. We too, use the natural camera which can be made easily available.
3) The reconstructed 3D-model should be accurate from all perspectives and in a convenient format.
4) The computer algorithm must be robust against noise and in the speed of execution. If possible, the number of views to be processed by the algorithm must be minimal.

Motivation behind this thesis is that most of the image-based techniques available need to find parameters for the camera position, motion etc (called as camera calibration), we propose a method in which the need for actual camera calibration is avoided.

1.4 Organization of the Report

The work presented in this thesis combines insights, methods and algorithms developed in order to resolve the problem of 3D reconstruction of generic real world object.

Chapter 2 highlights the preliminary information about the thesis as well as the basic concepts and fundamental theories of computer vision and general assumptions used. It also outlines the several principles considered in each stage of 3D reconstruction.

A brief description to the existing techniques involved in image-based modeling and rendering is given in Chapter 3. This chapter briefly examines three such approaches, namely, Volume Intersection, Marching Cubes, and Metric 3D Reconstruction from Surface of Revolution.

The complete theoretical information for 3D reconstruction is presented in Chapter 4. The invariant properties of single axis motion and surface of revolution are discussed. The surface formation process of 3D-model is described in details and certain criteria are outlined that have to be met in order to avoid actual camera calibration.

The actual designing and implementation part is outlined with detailed information in Chapter 5. Extensive assessment of the method is made and the reconstruction results are presented in Chapter 6 along with limitations pertaining to proposed method.

Conclusions are drawn in Chapter 7.

Appendix - B includes source code listing of implementation.

To understand how human vision might be modeled computationally and replicated on a computer, we need to understand the image acquisition process. The role of the camera in machine vision is analogous to that of the eye in biological systems.

This chapter introduces the camera model and defines the *epipolar* or *two view* geometry. A perspective camera model is described in section 2.1, which corresponds to the *pinhole* camera. It is assumed throughout this thesis that effects such as radial distortion are negligible and are thus ignored.

Section 2.3 defines the epipolar geometry that exists between two cameras. A special matrix will be defined that incorporates the epipolar geometry and forms the building block of the reconstruction problem.

2.1 Pinhole camera model

The *pinhole camera* is the simplest, and the ideal, model of camera function [14]. It has an infinitesimally small hole through which light enters before forming an inverted image on the camera surface facing the hole. To simplify things, we usually model a pinhole camera by placing the image plane between the focal point of the camera and the object, so that the image is not inverted. This mapping of three dimensions onto two dimensions, is called a *perspective projection* (see Figure 2.1), and perspective geometry is fundamental to any understanding of image analysis.

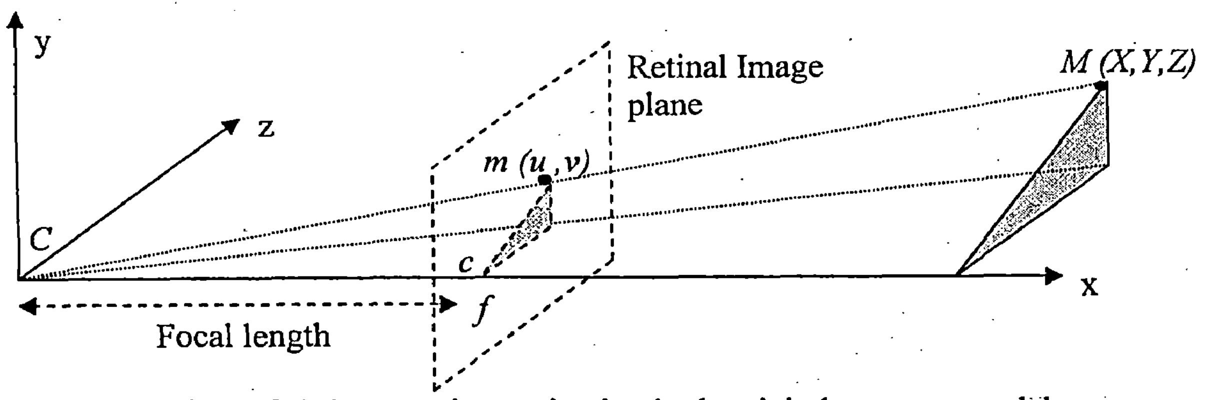


Figure 2.1: Perspective projection in the pinhole camera model.

The coordinates of a 3D point $M = [X, Y, Z]^T$ in a Euclidean world coordinate system and the retinal image coordinates $\boldsymbol{m} = [u, v]^T$ are related by the following equation:

$$s\tilde{m} = P\tilde{M} \tag{2.1}$$

where s is a scale factor, $\tilde{m} = [u, v, 1]^T$ and $\tilde{M} = [X, Y, Z, 1]^T$ are the homogeneous coordinates of vector $\boldsymbol{m}$ and $\boldsymbol{M}$ respectively. And $\boldsymbol{P}$ is a 3x4 matrix which is called as the perspective projection matrix.

Figure 2.1 illustrates this process. The figure shows the case where the projection centre is placed at the origin of the world coordinate frame and the retinal plane is at $Z = f = 1$. Then $u = \frac{fX}{Z}, v = \frac{fY}{Z}$ and

$$\boldsymbol{P} = [\boldsymbol{I}_{3x3}\ \boldsymbol{0}_3] \tag{2.2}$$

The optical axis passes through the centre of projection (camera) C and is orthogonal to the retinal plane. The point c is called the *principal point,* which is the intersection of the optical axis with the retinal plane. The focal length f of the camera is also shown, which is the distance between the centre of projection and the retinal plane.

If only the perspective projection matrix $\boldsymbol{P}$ is available, it is possible to recover the coordinates of the optical centre or camera.

The world coordinate system is usually defined as follows: the positive Y-direction is pointing upwards, the positive X-direction is pointing to the right and the positive Z-direction is pointing into the page.

2.2 Camera Calibration

Recovering 3D structure from images becomes a simpler problem when the images are taken with *calibrated* cameras. For our purposes, a camera is said to be *calibrated* if the mapping between image coordinates and directions relative to the camera center are known. However, the position of the camera in space (i.e. its translation and rotation with respect to world coordinates) is not necessarily known.

For an ideal pinhole camera delivering a true perspective image, this mapping can be characterized completely by just five numbers, called the *intrinsic parameters* of the camera. In contrast, a camera's *extrinsic parameters* represent its location and rotation in space. The five intrinsic camera parameters are:

1. The x-coordinate of the center of projection, in pixels (u_0)
2. The y-coordinate of the center of projection, in pixels (v_0)
3. The focal length, in pixels (f)

4. The aspect ratio (a)
5. The angle between the optical axes (α)

The camera calibration matrix, denoted by ***K***, contains the intrinsic parameters of the camera used in the imaging process. This matrix is used to convert between the retinal plane and the actual image plane: tan α

$$K = \begin{bmatrix} \frac{f}{p_u} & (\tan\alpha)\frac{f}{p_v} & u_0 \\ 0 & \frac{f}{p_v} & v_0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.3}$$

Here, the focal length f acts as a scale factor. In a normal camera, the focal length mentioned above does not usually correspond to 1. It is also possible that the focal length changes during an entire imaging process, so that for each image the camera calibration matrix needs to be reestablished.

The values p_u and p_v represent the width and height of the pixels in the image, $c = [u_0, v_0]^T$ is the principal point and α is the skew angle. This is illustrated in Figure 2.2.

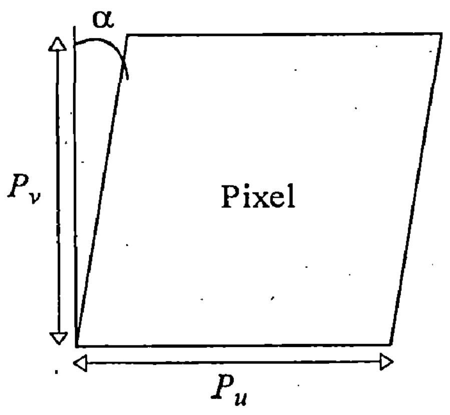


Figure 2.2: Illustration of pixel skew.

It is possible to simplify the above matrix:

$$K = \begin{bmatrix} f_u & s & u_0 \\ 0 & f_v & v_0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.4}$$

where f_u and f_v are the focal lengths measured in width and height of the pixels, s represents the pixel skew and the ratio f_u: f_v characterizes the aspect ratio of the camera.

We will approximate the skew factor s to zero and $f_u = f_v = c$, then equation (2.4) will reduces to

$$K = \begin{bmatrix} c & 0 & u_0 \\ 0 & c & v_0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.5}$$

It is possible to use the camera calibration matrix to transform points from the retinal plane to points on the image plane:

$$m^{\sim} = K\, m^{\sim}_R \tag{2.6}$$

Therefore, equation (2.6) is very important in 3D reconstruction process.

Now let us consider extrinsic parameters. Camera Motion in a 3D scene is represented by a *rotation* matrix $\boldsymbol{R}$ and a *translation* vector $\boldsymbol{t}$. The motion of the camera from coordinate C_1 to C_2 is then described as follows:

$$C_2^{\sim} = \begin{bmatrix} \boldsymbol{R} & \boldsymbol{t} \\ \boldsymbol{0}_3^T & \boldsymbol{1} \end{bmatrix} C_1^{\sim} \tag{2.7}$$

where $\boldsymbol{R}$ is the 3x3 rotation matrix and $\boldsymbol{t}$ is the translation in the X-, Y - and Z-directions. The motion of scene points is equivalent to the inverse motion of the camera (Pollefeys [16] defines this as the other way around):

$$\boldsymbol{M}_2^{\sim} = \begin{bmatrix} \boldsymbol{R}^T & -\boldsymbol{R}^T\boldsymbol{t} \\ \boldsymbol{0}_3^T & \boldsymbol{1} \end{bmatrix} \boldsymbol{M}_1^{\sim} \tag{2.8}$$

Equation (2.1) with equations (2.2), (2.5) and (2.6) then redefine the perspective projection matrix:

$$s\boldsymbol{m}^{\sim} = \boldsymbol{K}\begin{bmatrix} \boldsymbol{R} & \boldsymbol{t} \end{bmatrix}\boldsymbol{M}^{\sim} \quad \text{where } \boldsymbol{P} = \boldsymbol{K}\begin{bmatrix} \boldsymbol{R} & \boldsymbol{t} \end{bmatrix} \tag{2.9}$$

2.3 Epipolar Geometry

The epipolar geometry exists between a two camera systems [18]. With reference to Figure 2.3, the two cameras are represented by C_1 and C_2. Point $\boldsymbol{m}_1$ in the first image and point $\boldsymbol{m}_2$ in the second image are the imaged points of the 3D point $\boldsymbol{M}$. Points e_1 and e_2 are the so-called *epipoles*, and they are the intersections of the line joining the two cameras C_1 and C_2 with both image planes and the projection of the cameras in the opposite image.

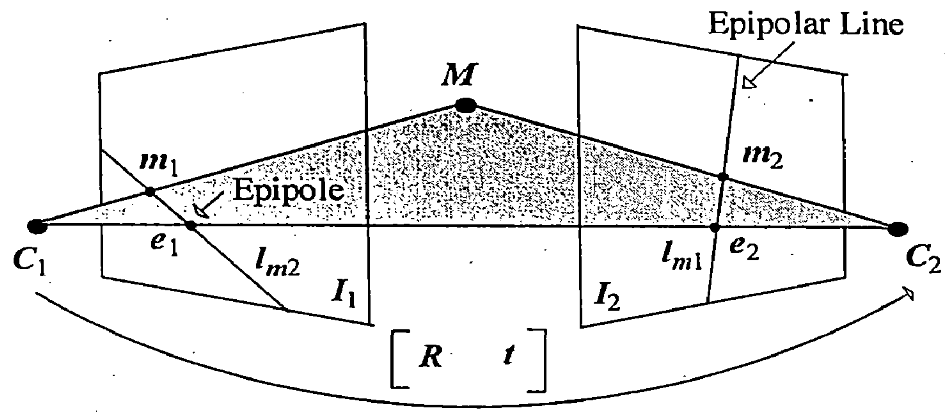


Figure 2.3: Epipolar Geometry

The plane formed with the three points $<C_1MC_2>$ is called the *epipolar plane*. The lines lm_1 and lm_2 are called the *epipolar lines* and are formed when the epipoles and image points are joined.

The point $\boldsymbol{m}_2$ is constrained to lie on the epipolar line lm_1 of point $\boldsymbol{m}_1$. This is called the *epipolar constraint*. To visualize it differently: the epipolar line lm_1 is the intersection of the epipolar plane mentioned above with the second image plane I_2. This means that image point $\boldsymbol{m}_1$ can correspond to any 3D point (even points at infinity) on the line $<C_1M>$ and that the projection of $<C_1M>$ in the second image I_2 is the line lm_1. All epipolar lines of the points in the first image pass through the epipole e_2 and form thus a pencil of planes containing the baseline $<C_1C_2>$.

The above definitions are symmetric, in a way such that the point of $\boldsymbol{m}_1$ must lie on the epipolar line lm_2 of point $\boldsymbol{m}_2$.

Expressing the epipolar constraint algebraically, the following equation needs to be satisfied in order for $\boldsymbol{m}_1$ and $\boldsymbol{m}_2$ to be matched:

$$\boldsymbol{m}_2^{\sim T} \boldsymbol{F} \boldsymbol{m}_1^{\sim} = 0 \tag{2.10}$$

where F is a 3 x 3 matrix called as the *fundamental matrix*. The following equation also holds:

$$l_{m1} = Fm_1^{\sim} \tag{2.11}$$

since the point m_2 corresponding to point m_1 belongs to the line lm_1 [25]. The role of the images can be reversed and then:

$$m_1^{\sim T} F^T m_2^{\sim} = 0 \tag{2.12}$$

which shows that the fundamental matrix is changed to its transpose.

Making use of equation (2.9), if the first camera coincides with the world coordinate system then

$$\begin{aligned} s_1 m_1^{\sim} &= K_1 \begin{bmatrix} I_{3x3} & 0_3 \end{bmatrix} M^{\sim} \\ s_2 m_2^{\sim} &= K_2 \begin{bmatrix} R & t \end{bmatrix} M^{\sim} \end{aligned} \tag{2.13}$$

where K_1 and K_2 are the camera calibration matrices for each camera, and R and t describe a transformation (rotation and translation) which brings points expressed in the first coordinate system to the second one. The fundamental matrix can then be expressed as follows:

$$F = K_2^{-T} [t]_x R K_1^{-1} \tag{2.14}$$

where $[t]x$ is the antisymmetric matrix.

Since $\det([t]x) = 0$, $\det(F) = 0$ and F is of rank 2. The fundamental matrix is also only defined up to a scalar factor, and therefore it has seven degrees of freedom (7 independent parameters among the 9 elements of F).

A note on the fundamental matrix: if the intrinsic parameters of the camera are known, such as in equation (2.14), then the fundamental matrix is called the *essential matrix* [25]. Another property of the fundamental matrix is derived from equations (2.10) and (2.11):

$$Fe_1^{\sim} = F^T e_2^{\sim} = 0 \tag{2.15}$$

Clearly, the epipolar line of epipole e_1 is $F^{\sim} e_1$.

2.4 VRML Representation

VRML (Virtual Reality Modeling Language) is an object oriented standard for the representation of 3D-model [10]. More specifically, it is a scene-description language in which set of objects is described hierarchically with their appearance information, such as placement of object, events, transformation etc. VRML-code is a simple ASCII type code.

Objects in VRML scenes can be broken into two components: their geometry, i.e. the shapes which make them up, and their qualities such as colors, materials, textures, and position or orientation. VRML uses the left-handed coordinate system; x is the width, y is the height, and z is the depth.

After the file has been created, it could be easily published in the Web by placing it at a Web site. And VRML models are easily viewed by a web browser that supports the prevalent VRML file. Specifications to the file construction of VRML could be sourced freely from [31].

The general structure that comprised the essential elements used in this work is shown below. The files generated are called world files and have an extension '.wrl'.

```
#VRML V2.0 utf8
Shape {
      appearance Appearance {
            material Material {
                        .........
}     }
geometry IndexedFaceSet  {
      solid TRUE
      coord Coordinate {
                 point [
                        .........
                  ]
            }
          coordIndex [
                        .........
                 ]
          }
}
```

Figure 2.4: General Format of VRML file.

2.5 Texture Mapping

Estimating surface properties, like color etc, from an image and a 3D model can be thought of as the inverse of creating an image from the model with known appearance [19] [21]. In image formation, or rendering, the image plane samples the light rays entering the camera from the scene. Two basic techniques are used to simulate these light rays: forward projection of surfaces from the scene into the camera, and back projection of rays from the camera into the scene, called ray tracing. In the forward projection of surfaces, each surface in the model is projected into the image, updating the color of the image if that surface is visible. In ray tracing, each pixel is mapped to a ray (or set of rays) that intersect the scene. The color of the pixel is set based on the appearance of that surface. In either case, the surface color can be determined by texture mapping, which models the surface color with an image, called the texture map.

Texture mapping in VRML is basically the same as texture mapping in all other areas of 3D graphics. It is all based on the same fundamental concepts.

The Texture Nodes in VRML 2.0 are:

ImageTexture: defines a still texture map using an image file.

MovieTexture: defines a moving texture map using a movie file.

PixelTexture: defines a still texture map made from explicit pixel values.

TextureTransform: defines a 2D transformation applied to texture coordinates.

Appearance: where the texture nodes live.

Shape: where the Appearance node lives.

TextureCoordinate: defines a set of 2D coordinates to be used to map textures to the vertices of subsequent geometry nodes like IndexedFaceSet or ElevationGrid.

In the VRML 2.0 format, the Texture node exists as part of an Appearance node. Material, texture, and texture transform are always related to one another (see [28] [15]). Also, the Appearance node exists inside of a Shape node. This associates a specific appearance with a specific geometrical object (in the example below, a cube). No other object in the file will have this appearance unless specified by the programmer.

2.6 Rigid Body Transformation

This is an assumption that an object's size and shape are *invariant* to its translation or rotation in the Euclidean space. We have assumed this to be true and valid in the entire course of the work. This fundamental assumption, if it breaks down, would result in high degrees of complexity in 3D-model reconstruction. For instance, the task of modeling the deformity of a fluid object that changes in shape after a transformation needs consideration of the fluid mechanics principles. Figure 2.5 shows an example of a rigid body object.

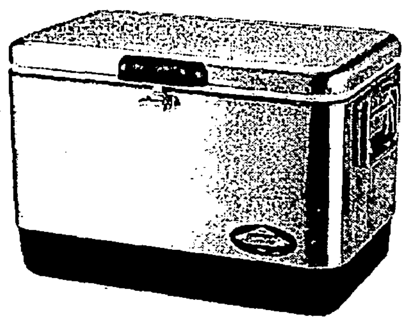


Figure 2.5: A Steel Cooler

3.1 Overview

Recently, the trends of image-based modeling and rendering to reconstruct 3D-models have been reported in [1] [2] [4] [6] [8] [11]. Image-based modeling is a passive technique that relies primarily on a sequence of images to build its virtual mode.

The basic idea is to take the views of an object from different angles (usually at least 3 cameras are used as shown in Figure 3.1). However, depth information is lost during the image formation process when 3D structures in the world are projected onto 2D images. Multiple images from different viewpoints can be used to resolve this problem. Then, geometric information of the object of interest is extracted from each of these views and finally a 3D-model representation of this object is reconstructed by using computer graphic techniques.

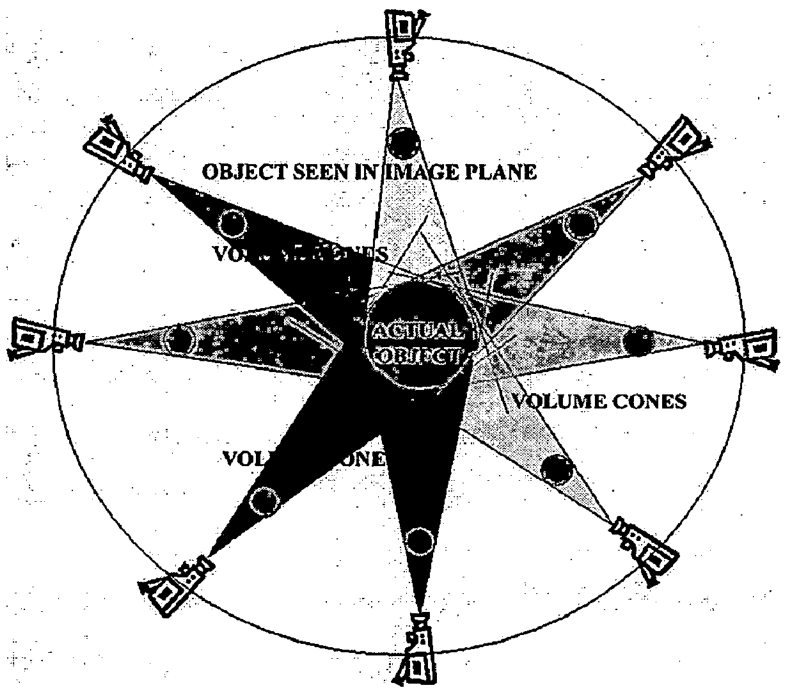


Figure 3.1: Images of object are captured from different views with known angle.

However, this approach may also require high processing power, long training time and large memory requirements. But it is generally deemed that image processing and analysis of images in 2D domains are far easier than processing problems pertaining to 3D model-based rendering. And we will not require specialized equipments; a simple and cheap digital camera can be used to take images.

3.2 Related Work

This chapter introduces and reviews some of the existing techniques of computer vision in the area of computing a 3D Euclidean reconstruction using images of a scene taken by a standard camera.

The reconstruction of a complex 3D object from multiple images has been a fundamental problem in the field of computer vision. Given a set of images of a 3D object, in order to recover the lost third dimension, depth, it is necessary to compute the relationship between images through correspondence. By finding corresponding primitives such as points, edges or regions between the images, such that the matching image-points all originate from the same 3D object point, knowledge of the camera geometry can be combined in order to reconstruct the original 3D surface.

Some of the research contributions in the field have proposed fully working systems for specific applications, some other have instead mostly focused on one or some of the involved aspects but provided a general application context. For example, Moezzi et al. [22], [23], propose an entire specific system for image-acquisition, model-construction and play-back interactive rendering, while Ofek et al. [21], mostly focus on extraction of textures from a generic video sequence for high-fidelity model-based texture mapping.

There can well be different ways of generating 3D models from single or multiple images [1] [3] [6] [13] [17]. In this chapter, three of such approaches are discussed briefly. Section 3.2.1 discusses the formation of 3D structure from multiple images using volume intersection technique. A survey of image-based volumetric scene reconstruction can be found in the works of Slabaugh et al. [11].

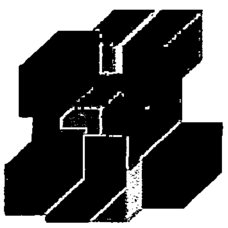
In section 3.2.2, a marching cube technique is discussed which is used for smoothing of the surface generated in volumetric reconstruction technique.

Section 3.2.3 contains brief introduction of metric 3D reconstruction and texture acquisition of surfaces of revolution from a single uncalibrated view.

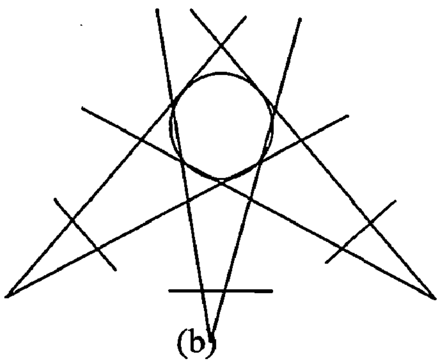
3.2.1 Volume Intersection

This is a class of methods of converting the geometric information obtained from images of an object into a set of cubes that best represents it [4] [8] [12]. This technique exploits the idea that, a bigger component can be obtained by using number of small basic components. This process is very much similar to the process of building house using number of bricks.

Using number of small cubes as basic component, we can construct a complete approximate 3D-model (see Figure 3.2). In computer vision literature such cubes are called *voxels* and the process of finding such cube is called *voxelization.*



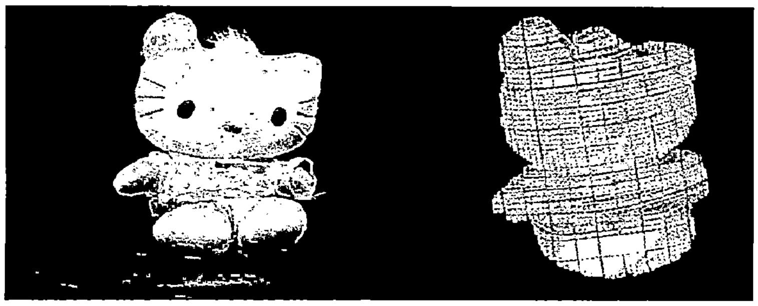
(a)



(b)

Figure 3.2: (a) Number of cubes as building blocks, (b) Volume Intersection

Intuitively, one would assume that a proper voxelization simply ensures all voxels are inclusive of the object body. Those that are not, are discarded away thus mimicking an effect of "carving" the shapes and curves that resembles that of an original object from independent views. Figure 3.3 shows result of this method.



a) b)

Figure 3.3: (a) Actual image of Toy, (b) Extracted 3D-model of Toy using voxelization

3.2.2 Marching Cubes

Marching Cubes is an algorithm for rendering isosurfaces in volumetric data. It was designed by William E. Lorensen and Harvey E. Cline to extract surface information from a 3D field of values [29].

The basic notion is that we can define a voxel by the pixel values at the eight corners of the cube. Therefore a potential 256 possible combinations of corner status is obtained. By considering rotation, mirroring and Inverting the state of all corners it is found that out of this 256 corner status combinations only 15 are required.

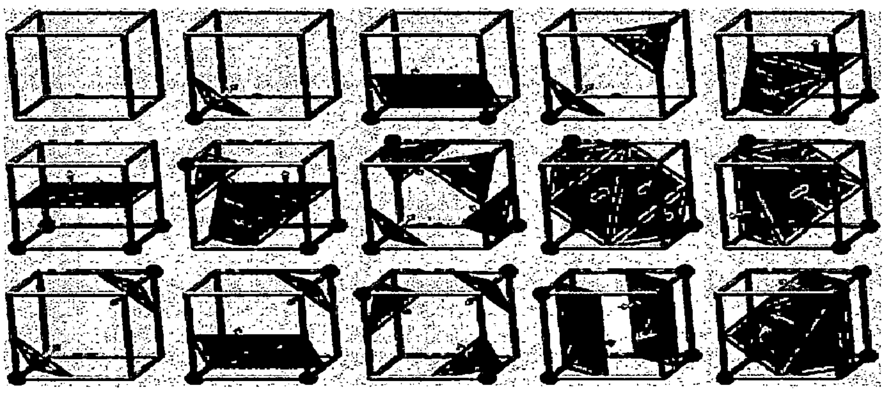


Figure 3.4: 15 combinations of corner status.

As shown in Figure 3.4, if one or more pixels of a cube have values less than the user-specified isovalue, and one or more have values greater than this value, we know the voxel must contribute some component of the isosurface [26]. By determining which edges of the cube are intersected by the isosurface, we can create triangular patches which divide the cube between regions within the isosurface and regions outside. By connecting the patches from all cubes on the isosurface boundary, we get a surface representation, as shown in Figure 3.5.

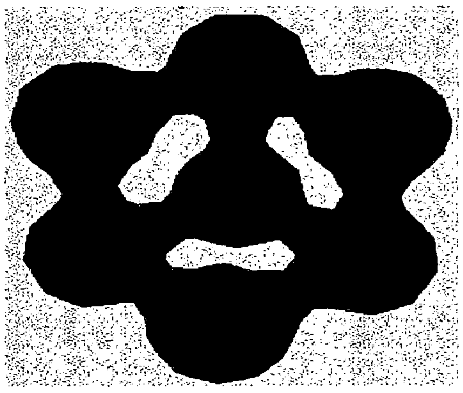


Figure 3.5: Reconstructed triangular patched surface using marching cube.

3.2.3 Metric 3D Reconstruction From SORs

Metric 3D reconstruction from *surface of revolution* (SOR) is very recent contribution to image-based modeling. It was proposed by Carlo Colombo, Alberto Del Bimbo, and Federico Pernici [1] [2]. They addressed a method for solution to the problem of metric 3D reconstruction of a generic object and its texture acquisition from a single uncalibrated view of SOR.

The proposed solution exploits the projective properties of imaged SORs, expressed through planar and harmonic homologies. These geometric constraints induced in the image by the symmetry properties of the SOR structure are used for camera calibration. The required parameters for camera calibration are directly obtained from the analysis of the visible elliptic segments of two imaged cross sections of the SOR.

The same elliptic segments are used together with the SOR apparent contour, to reconstruct the 3D structure and texture of the SOR object, which are thus obtained from calculations in the 2D domain.

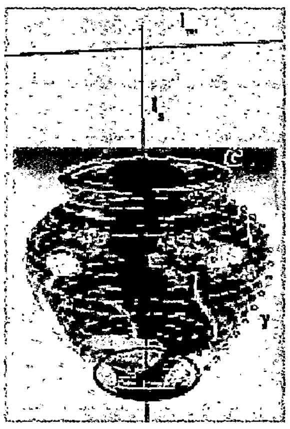


Figure 3.6: Recovery of 3D structure.

In Figure 3.6, it is shown that, elliptic imaged cross sections of the SOR can be used to recover the surface of generic imaged object. Since the homology constraints are of general applicability, the solution can be applied under full perspective conditions to any type of surface of revolution with at least two partially visible cross sections.

This Chapter will guide through the steps of solving problem for recovering the 3D shape of object using surface of revolution from multiple uncalibrated perspective views.

In first two sections basic geometry of Single axis motion and surface of revolution is discussed. This is important in order to understand the underlying situation (geometry and complexity) so that the problem can be simplified and a concise algebraic solution computed.

4.1 Single axis motion

Given a static camera, and a generic object rotating on a turntable (as shown in Figure 4.1), single axis motion (SAM) provides a sequence of different images of the object.

Now onward we will use the world coordinate system defined as follows: the positive X-direction is pointing to the right, the positive Y-direction is pointing upwards, and the positive Z-direction is pointing into the page.

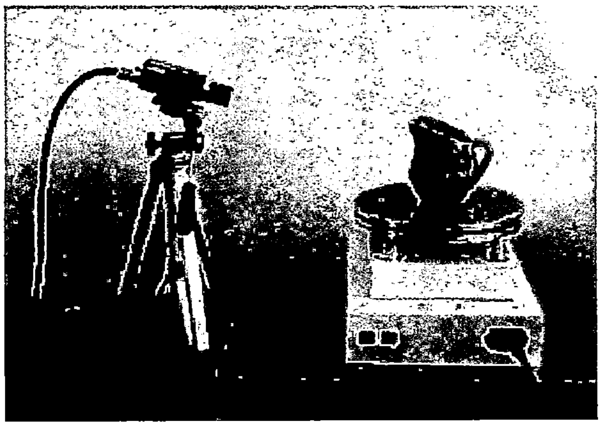


Figure 4.1: Image acquisition system consists of a turn table, a camera and a computer.

This sequence can be imagined as being produced by a camera that performs a virtual rotation around the turntable axis while viewing a fixed object.

Single axis motion can be described in terms of its fixed entities –i.e., those geometric objects in space or in the image that remain invariant throughout the sequence [24]. In particular, the imaged fixed entities can be used to express orthogonality relations of geometric objects in the scene by means of the *image of the absolute conic* (IAC) ω –

an imaginary point conic directly related to the camera matrix K as $\omega = K^{-T}K^{-1}$ [5]. Important fixed entities for the SAM are the imaged circular points i_π and j_π of the pencil of planes π orthogonal to the axis of rotation, and the horizon $l_\pi = i_\pi - j_\pi$ of this pencil. The imaged circular points form a pair of complex conjugate points which lie on ω:

$$i_\pi^T \omega\, i_\pi = 0;\; j_\pi^T \omega\, j_\pi = 0 \tag{4.1}$$

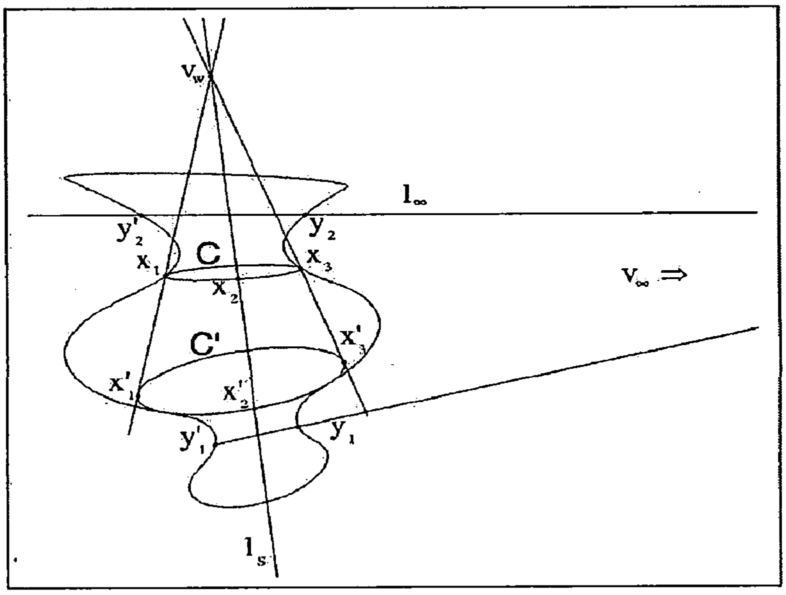


Figure 4.2: Basic projective properties for an imaged SOR.

In practice, as i_Π and j_Π contain the same information, the two equations above can be written in terms of the real and imaginary parts of either point. Other relevant fixed entities are the imaged axis of rotation l_a and the vanishing point v_n of the normal direction to the plane passing through l_a and the camera center (see Figure 4.2). These are in pole-polar relationship with respect to ω:

$$l_a = \omega v_n \tag{4.2}$$

Equations (4.1) and (4.2) were used separately in the context of approaches to 3D reconstruction from turntable sequences. In particular, (4.1) was used in [1] and in [5] to recover metric properties for the pencil of parallel planes π given an uncalibrated turntable sequence. In both cases, reconstruction was obtained up to a 1D projective ambiguity, since the two linear constraints on ω provided by (4.1) were not enough to calibrate the camera. On the other hand, (4.2) was used in [7] to characterize the epipolar geometry of SAM in terms of l_a and v_n given a calibrated turntable sequence. Clearly, in this case, the a priori knowledge of intrinsic camera parameters allows one to obtain an unambiguous reconstruction. In the case of an SOR object, assuming that its symmetry axis coincides with the turntable axis, the apparent contour remains unchanged in every frame of the sequence. Therefore, for an SOR object, the fixed entities of the motion can be computed from any single frame of the sequence. According to this consideration, an SOR image and a single axis motion sequence share the same projective geometry: the fixed entities of SOR geometry correspond to the fixed entities of single axis motion. In particular,

1. l_a corresponds to l_s;

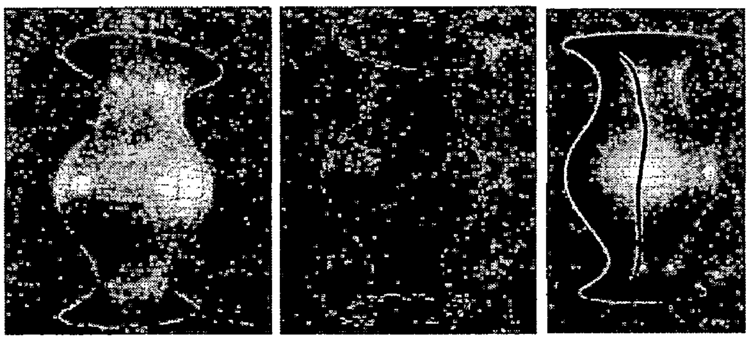
2. v_n corresponds to v_∞;

3. (i_π, j_π) correspond to (i, j);

4. l_π corresponds to l_∞ . i x j, where i and j denote the imaged circular points of the SOR cross sections.

4.2 SOR geometry

Being a subclass of SHGC, SOR enjoy all of their properties [2]. A SOR can be parameterized as $\sigma(\theta, y) = (\rho(y) \cos\theta, y, \rho(y) \sin\theta)$, where y is the (straight) axis of revolution. In 3D space, all *parallels* (i.e., cross-sections with planes y = constant orthogonal to the axis) are circles. The curves θ = constant, called *meridians*, are obtained by cutting the surface with planes passing through the axis, and characterize the specific SOR shape through the *scaling function* $\rho(y)$. Parallels and meridians are locally mutually orthogonal in 3D space, but not in a 2D view (see Figure 4.3).



(a) (b) (c)

Figure 4.3: (a) Parallels and meridians on a SOR, (b) Apparent contour, (c) Image of the Meridian

Typically observable curves in a SOR image are imaged parallels (which are always ellipses, being the perspective images of circular curves) and *apparent contours* (see Figure 4.3 (b)): the latter should not be confused with imaged meridians. In fact, while meridians are planar 3D curves, an apparent contour is the image of the (usually non planar) 3D curve of all the points at which the projection rays are tangent to the surface, referred to as *generating contour*. Figure 4.3 (b) and (c) remarks the difference between apparent contours and imaged meridians.

4.3 Approach

In this section we will describe the procedure of finding surface properties of the object. First we will define some assumption which will ease the process of camera calibration and surface formation. Following are the requirements that should be considered.

1. Generic object to be modeled must be a rigid body (i.e. shape of object must be invariant to its translation or rotation in the Euclidean space, as section 2.6).
2. The axis of rotation must exactly coincide with the middle of the image.
3. Angle $\theta = 10°$ should be constant between each pair of adjacent camera.
4. We will have T = 36, where T is the total no images.
5. We assume that, the aspect ratio and skew properties of natural camera to be known and remains to be invariant.
6. Input sequence is the sequence of silhouetted image of the object taken by natural camera. Suppose, we want reconstruct a ball then Figure 4.4 (a) shows one instance of the silhouetted input image and Figure 4.4 (b) shows resultant 3D-model.

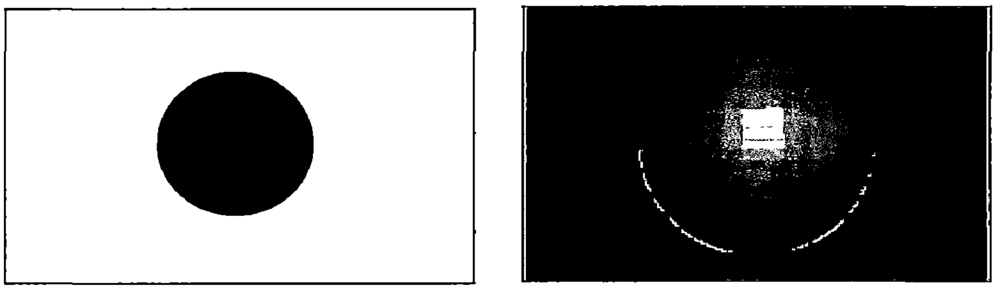


Figure 4.4: (a) Silhouetted Input Image, (b) Image of resultant object.

7. The vertical and horizontal sides of input image must exactly fit to the Retinal Image Plane delineated in Figure 2.1. Such alignment simplifies the procedure for projection and back projection between the 3D space and the image planes.

The aforementioned assumptions 2, 3, 5, and 7 form a constrained environment by which we can avoid the need of actual camera calibration. Following section will discuss the issues pertaining to the avoidance of camera calibration.

4.3.1 Avoiding a need for Camera Calibration

From equation 2.9, P matrix can be split into two matrices, M_{int} and M_{ext}, which depend on the intrinsic and extrinsic camera parameters respectively.

$$\text{i.e. } P = M_{int} M_{ext}, \quad \text{where. } M_{int} = K, \text{ and } M_{ext} = \begin{bmatrix} R & t \end{bmatrix} \tag{4.3}$$

Since we have cameras at different angles of rotation, θ then for the k^{th} view, we manipulate Equation (4.3) using equation (2.5) as:

$$P_k = \begin{bmatrix} c & 0 & u_0 \\ 0 & c & v_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos k\theta & 0 & \sin k\theta \\ 0 & 1 & 0 \\ -\sin k\theta & 0 & \cos k\theta \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & X_0 \\ 0 & 1 & 0 & Y_0 \\ 0 & 0 & 1 & Z_0 \end{bmatrix} \tag{4.4}$$

$$\text{where, } R_{k\theta} = \begin{bmatrix} \cos k\theta & 0 & \sin k\theta \\ 0 & 1 & 0 \\ -\sin k\theta & 0 & \cos k\theta \end{bmatrix} \text{ and } t = [X_0, Y_0, Z_0]^T$$

Now, we will see the impact of changing X_0, Y_0, and Z_0 one by one. First, let us see the impact of changing the Y_0 value. It can be seen clearly that by changing the Y_0 value, our camera coordinate system can be shifted up or down along that axis. This does not affect the shape integrity of our extracted 3D-model. Thus, we set the Y_0 value equal to 0.

Next, we align our set up in such a way that our X_0 is also set to 0. This can be done easily by intersecting the rotation axis with the center of the image. Thus, both of the coordinate systems are aligned with one another and are translated along the z direction only, i.e. into the page or outward to the page.

Consider, any imaged point (u_1, v_1) in input image taken from camera which is rotated by an angle θ from its previous view. Now, we want to estimate its coordinate in the virtual world coordinate system, and then from equation (2.1), we have

$$\begin{bmatrix} s_1 u_1 \\ s_1 v_1 \\ s_1 \end{bmatrix} = M_{\text{int}} R_\theta M_{ext} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{4.5}$$

solving equation (4.5), we get

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} \cos(\theta) s_1 (u_1 - u_0)/c + (Z_0 - s_1)\sin(\theta) \\ s_1 (v_1 - v_0)/c \\ \cos(\theta)(s_1 - Z_0) + \sin(\theta) s_1 (u_1 - u_0)/c \end{bmatrix} \tag{4.6}$$

As s_1, c, and Z_0 are constants, we can set them to any suitable value. The most suitable assignment will be taking the focal distance to be equal to the displacement along the Z-direction.

$$Z_0 = s_1 = c$$

Therefore, equation (4.6) will be simplified as

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} \cos(\theta)(u_1 - u_0) \\ (v_1 - v_0) \\ \sin(\theta)(u_1 - u_0) \end{bmatrix} \tag{4.7}$$

Now, the only remained unknown in equation (4.7) is (u_0, v_0), which can be easily computed. For this, we will consider the principal point (u_0, v_0) to be coincided with the image center whose value is half the dimensions of the images being captured (i.e. input images). In equation (4.7), all unknown can be derived from the restricted and controlled environment discussed above. This equation can be used further for the derivation of feature points. Hence, we have avoided the need for camera calibration.

4.3.2 Derivation of the Feature Points

In order to compute geometric information we first extract interest points in the image. Usually, we use high curvature points as they can be easily manipulated and represented in projective geometry. To extract these points, we will trace the *meridian* left to the axis of rotation. We may obtain the set of imaged feature points $X_{i,j}$ as shown in Figure 4.5.

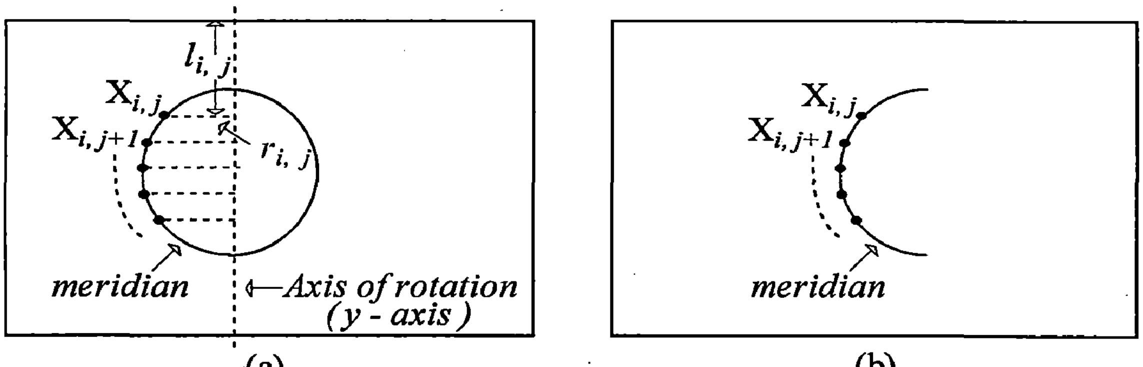


Figure 4.5: (a) Computation of feature points along meridian. (b) Extracted Feature Points.

These points are estimated in all images independently. Once these points are found, we will use direct correspondence of points from image to the virtual 3D Space.

As shown in Figure 4.5, the only entity to be derived is $X_{i,j}$, where

$$X_{i,j} = (x_{i,j}, y_{i,j}, z_{i,j}) \tag{4.8}$$

$X_{i,j}$ will be the surface points and they can be integrated to form a complete surface of 3D-model. From the geometry of SAM, SORs, and equation (4.7), we can derive the unknowns as bellow.

1. $x_{i,j} = r_{i,j} \times \cos(\theta)$ (4.9)
2. $y_{i,j} = l_{i,j}$ or (Image height / 2 – $l_{i,j}$) (4.10)
3. $z_{i,j} = r_{i,j} \times \sin(\theta)$ (4.11)

where,

$i = 0$ to m – 1 (i.e. 0th image, 1st image, so on),

$j = 0$ to n – 1,

m = total number of imaged points along *parallels*, (m = T).

n = total number of imaged points along *meridian*.

As we are using $r_{i,j}$ (as a vector) to calculate the coordinates of $X_{i,j}$, it will confirm that the imaged axis of rotation will pass through the origin and parallel to the y-axis of our world coordinate system. Therefore, imaged axis of rotation will be the projection of y-axis. The entire process of finding feature points can be algorithmically summarized as bellow.

```
for ( im = 0 to Total_No_Images)
{
   for ( i =0 to m – 1)
      {
         for ( j = 0 to n – 1)
         {
           Find ($X_{i,j}$)   // from ith image
         }
      }
}
```

Figure 4.6: Algorithmic summary of process of extracting feature points.

4.3.3 Surface Formation

In this step, we construct matched triangular meshes from the extracted feature points. Luiz et al. [20] has proposed a piecewise linear approximation method of adaptive polygonization of regular surfaces of the Euclidean 3D space. In which it is suggested that triangulation is the best technique in order to form smooth surface. We too, try to find the triangular pieces of the surface which are then integrated to form 3D-model.

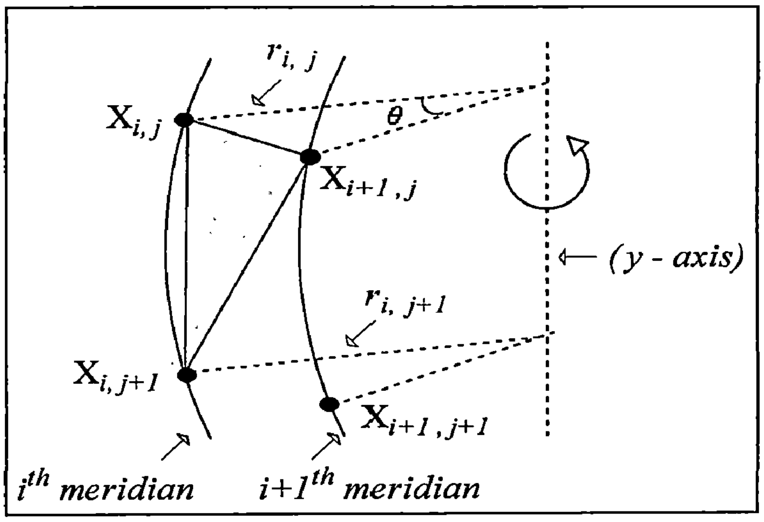


Figure 4.7: Surface formation using feature points.

In Figure 4.7, four extracted feature points are shown. These four points will constitute the surface of resultant virtual object [20]. We can construct a portion of surface using triangulation, for example, points $X_{i,j}$, $X_{i,j+1}$ and$X_{i+1,j}$ will form one triangle and so others.

This process of surface formation can be easily visualized and understood by using epipolar geometry, as shown in Figure 4.8.

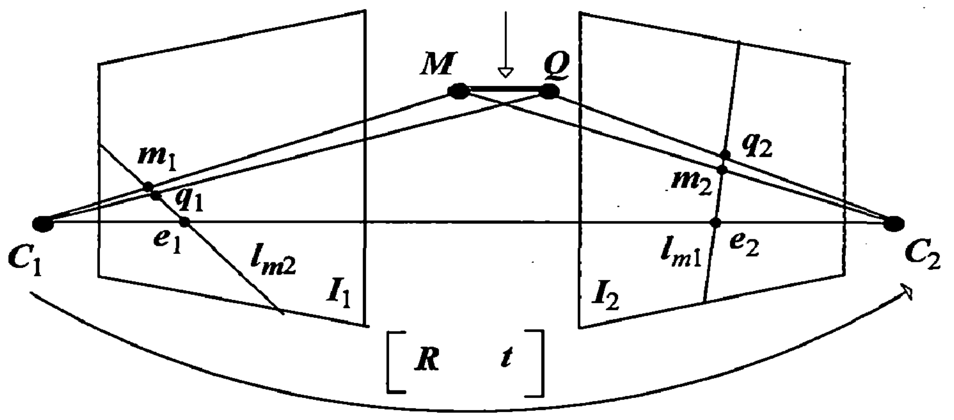


Figure 4.8: Epipolar geometry of two feature points from camera C_1 and C_2.

In above figure, there are two points M and Q (at same height) from two images taken by two adjacent cameras C_1 and C_2. Points M and Q are equivalent to the feature points $X_{i,j}$ and $X_{i+1,j}$ of Figure 4.7 respectively. As these two points are at same height and according to the epipolar constraints discussed in section 2.3, the line segment $<MQ>$ of two feature points will constitute a surface indicated by a vertical arrow.

We will use this simple and straightforward approach to form the surface. This will also ease the process of texture acquisition and its mapping (discussed in 4.3.4).

4.3.4 Texture Acquisition

Texture mapping is a shading technique for image synthesis in which a texture is mapped onto a surface in a three dimensional scene, much as wallpaper is applied to a wall [9]. A view dependant texture acquisition technique used to acquire surface properties of object. If **I** is the image space then we will try to map it as

$$X_{i,j} \rightarrow \mathbf{I}\,(u_i,\, v_i)$$

We will consider imaged points in pairs (i.e. point on the ith meridian and point on the i+1th meridian). We will make use of all T = 36 images. For texture acquisition following formulae can be used.

For points on ith meridian

$X_{i,j} = (x_{i,j}, y_{i,j}, z_{i,j}) \rightarrow (u_i,\, v_i)$ where,

$u_i = (\text{Image width} / 2 - r_{i,j}) + \Delta$

$v_i = y_{i,j} + \Delta$

For points on i+1th meridian

$X_{i+1,j} = (x_{i+1,j}, y_{i+1,j}, z_{i+1,j}) \rightarrow (u_{i+1}, v_{i+1})$

Where,

$u_{i+1} = (\text{Image width} / 2 - r_{i,j})$

$+ \text{sqrt}\,((x_{i+1} - x_i)^2 + (z_{i+1} - z_i)^2) + \Delta$

$v_{i+1} = y_{i+1,j} + \Delta$

Δ = Error due to camera zooming/skew properties and/or misplacement of axis of rotation.

The pinhole camera model assumes that the imaging process is a perfect perspective projection from world to image coordinate frames i.e. from 3D to 2D. However, real cameras are not perfect perspective projections [24] (especially when used with a short focal length lens) and non-linear distortions are introduced into the imaging process. There are several different forms of non-linear distortion, where the error is a radial displacement proportional to an even power of the distance from the centre of the image.

CHAPTER 5. DESIGN AND IMPLEMENTATION

In this chapter designing and software implementation of the project is discussed. Input to the project is a sequence of T =18 or 36 digital images, 576 x 768 pixel dimension, taken at θ =20° or 10° radial graduations. They are read and stored in a 2D array and its individual elements are accessed using iterations.

The main software component is required to do calculations and tests to locate all intersecting feature points. The implementation details of finding feature points are described in section 5.2.

Section 5.3 and 5.4 describe the process of surface formation and texture mapping respectively. Section 5.1 contains the information about the representation of input image format.

The source code listing is given in APPENDIX - B. The output is a 3D model in VRML format.

5.1 Image Reading

Before the images can be computationally processed, a thorough understanding of image format type is needed so as to extract intensity data from such files. The input going into the image-reading module is 8-bit grayscale TIF file and the output is a 2D integer array holding the intensity values of the image.

TIF is an image file format and it begins with an 8-byte image file header that points to an Image File Directory (IFD). An IFD contains information about the image as well as pointers to the actual image data.

Byte 0-1 : The byte order used within the file. The legal values are "II" and "MM". In the "II" format, byte order is always from the least significant byte, for both 16-bit and 32-bit integers. In the "MM" byte order, the converse is true.

Bytes 2-3 : An arbitrary but carefully chosen number (42) that further identifies the file as a TIFF file.

Bytes 4-7 : The offset in bytes of the first (IFD). The directory may be at any location in the file after the header. It is always used to refer to the beginning of the TIF file. The structure of.TIF file is described in Figure 5.1.

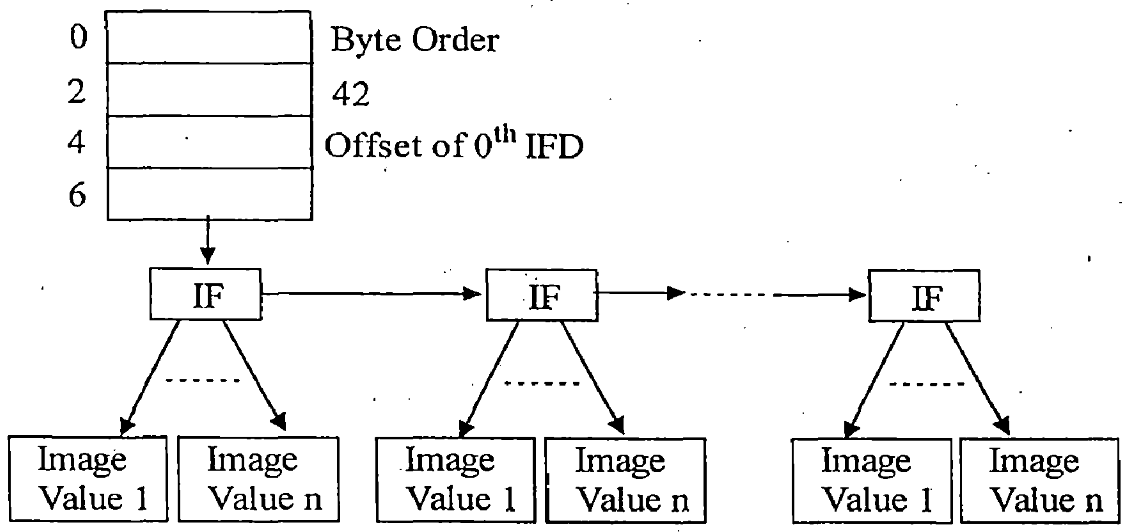
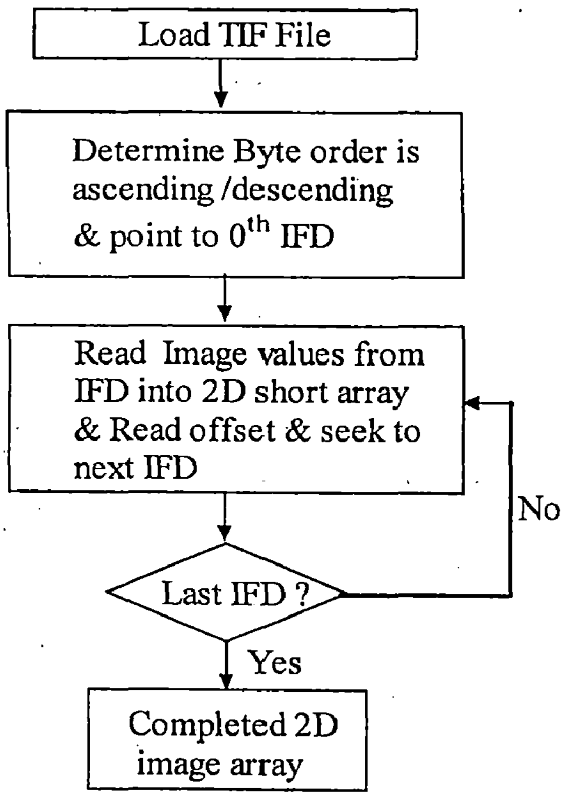


Figure 5.1: File Structure of the Tagged Image Format (TIF)

With understanding of the tiff file format, the image-reading algorithm is devised as shown in Flowchart 1.


Load TIF File
Determine Byte order is ascending /descending & point to 0th IFD
Read Image values from IFD into 2D short array & Read offset & seek to next IFD
No
Last IFD ?
Yes
Completed 2D image array

Flowchart 1: Image Reading module

5.2 Edge Finding

The edges of an object in the image are nothing but the surface outlines. Edges have special benefit in 3D reconstruction, as they provide the most reliable information about the whole object. The most popular edge detection technique is Canny's edge detection [30].

As the input images to the proposed method are binary images (i.e. black and white), there is no difficulty in finding the edge points. We can follow the iterative approach to find these surface points. Following are the steps involved in this process.

Step 1. The input image is read as discussed in section 5.1. The pixel dimensions are of 576 x 768 sizes. The pixels comprising the object body are assigned to value 1 and other pixels to 0.

Step 2. Now it is required to search for the pixel elements with values 1. This can be done by using iterative algorithm row-wise or column-wise. We follow the row-wise approach as shown in Figure 5.2.

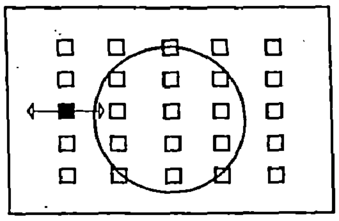


Figure 5.2: Iterative step of finding edge points.

In above figure there are 5 x 5 pixel elements depicted and a circular shape is shown. We go on searching the pixel elements row-wise until we get the pixel element with value 1.

Step 3. This process is repeated for the other rows. Not all rows are needed to be searched. We can skip a known number of rows (three or four); but that number should be constant throughout the process in order to get better results.

These extracted image points are called interest points or feature points. Once these points are found they need to be represented in a suitable mathematical form in order to analyze the captured information. This is given in section 5.4.

5.3 Surface formation

In this step the extracted feature points are integrated with each other in such a way that they are made to form the surface of 3D-model. These points are joined in the form of uniform continuous grid of triangles as discussed in section 4.3.3. Intuitively, a grid is a mesh of piecewise linear surface, consisting of triangular faces pasted together along their edges. For our purposes it is important to maintain the distinction between the connectivity of the mesh and its geometry. This piecewise curvilinear grid formation results into the surface of the body and it will also ease the process of texture acquisition.

In following two sub-sections we will define the representation of feature points in the form of 2D and 3D interpretation.

5.3.1 2D Point

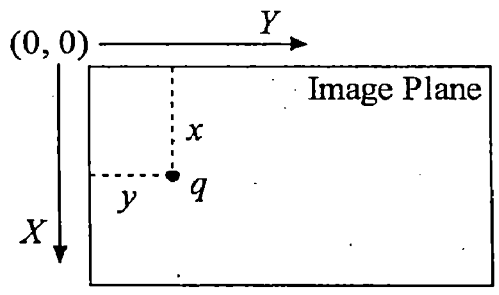


Figure 5.3: 2D Image Coordinate System.

In Figure 5.3, a 2D image coordinate system is shown. We have to transform this 2D coordinate system into the 3D world coordinate system. The 2D points in the image plane are referenced from top-left corner which will have coordinated (0, 0). With reference to this point, every point in this plane can be represented by vertical downward displacement as X-coordinate and horizontal rightward displacement as Y-coordinate.

Therefore, the point p shown in Figure 5.3 can be represented as

$$q \equiv [x, y]$$

5.3.2 3D Point

The 2D point is transformed into the 3D point using equation (4.9), (4.10), and (4.11).

$$Q \equiv [x, y, z]$$

5.3.3 Surface stitching

By a *surface* we mean a “compact, connected, orientable two-dimensional manifold, possibly with boundary, embedded in 3D coordinate system” [27]. Figure 5.4 depicts the process of surface formation, in which six feature points are shown. These feature points are used to form a piecewise triangular continuous grid of partial surfaces.

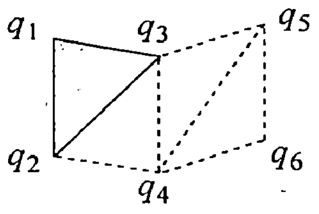


Figure 5.4: Triangular grid formation from feature points.

The order of the formation of triangles is kept as discussed in section 4.3.3. This can be done using following algorithm.

```
for ( im = 0 to Total_No_Images)
{
   for ( i =0 to m – 1)
      {
         for ( j = 0 to n – 1)
         {
           Triangle_Formation ( )
         }
      }
}
```

Figure 5.5: Algorithm describing the process of surface formation.

5.4 Texture Mapping

To allow texture-mapping, a surface must be parameterized onto a texture domain by assigning texture coordinates to its vertices. Generally, most of the texture mapping techniques involves the parameterization of a 3D surface onto the 2D domain for the purpose of texture-mapping [28].

In VRML, texture mapping is achieved by assigning a 2D texture map coordinate to each 3D vertex. The texture is then interpolated between vertices. The simplest approach is to use a whole camera image as a texture map and provide the correspondences between 3D model points and image points.

The experiments are performed on a personal computer with 256 MB of RAM, Intel PIV 2.00GHz CPU and 32MB frame buffer. The images are captured with a 2/3" Color Progressive scan CCD camera at a resolution of 1280x960.

The algorithm to find feature points, surface formation, and texture mapping are successfully implemented in Visual C++. The experiments have been performed on various shapes of objects. The obtained results are compared with the 3D-models derived using Voxelization.

6.1 Using Homologous images

Homologous images are the best suitable candidates for input to the proposed method. Homologous images preserve the similarity in position, structure, etc. They can be placed exactly bilateral symmetrical to the axis of rotation. There can be many families of such homologous images. Figure 6.1 shows some homologous images of simple geometrical shapes as inputs and their corresponding outputs.

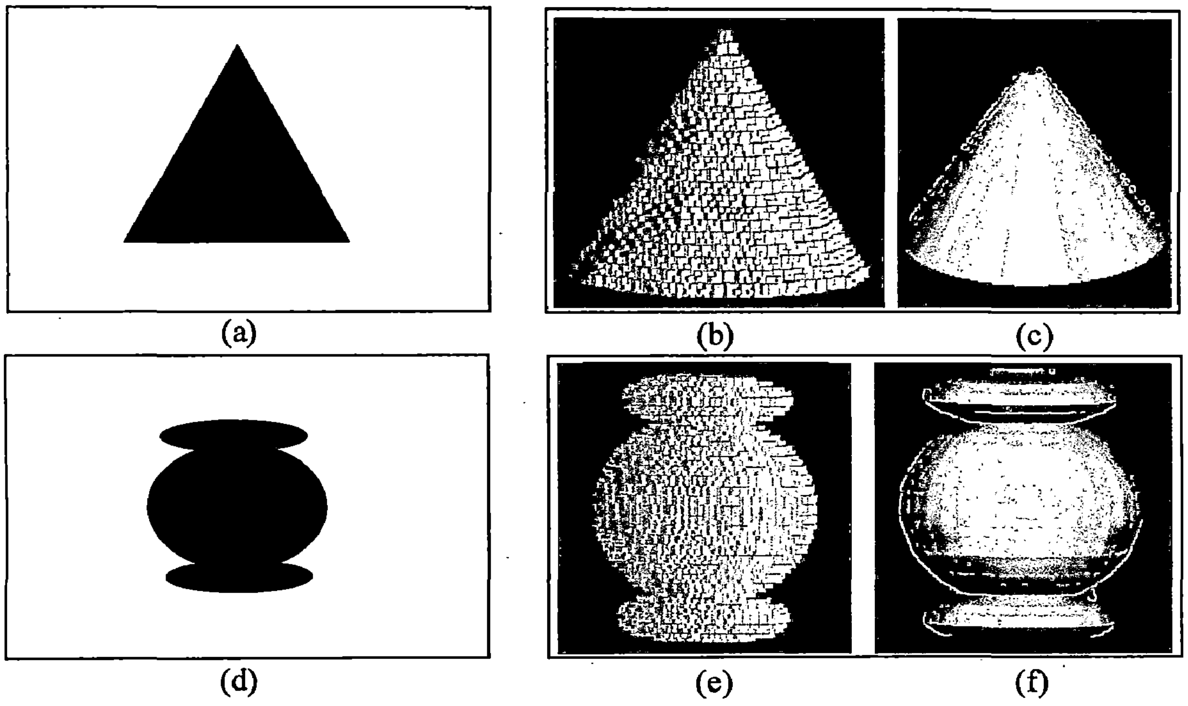


Figure 6.1: (a), (d) Input Images; (b), (e) Output using Voxelization; (c), (f) Output using Proposed Method.

In proposed method, texture acquisition can be done very easily and effectively as discussed in section 4.3.4. Following figures show the texture mapped output.

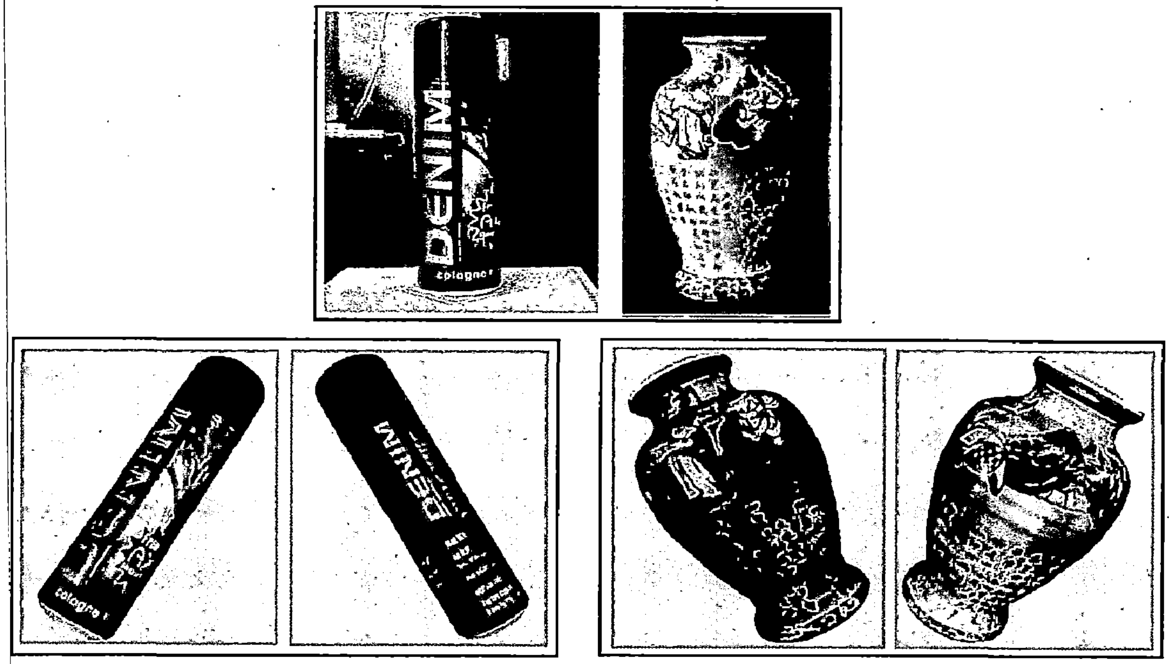


Figure 6.2: Top – Input: Homologous imaged objects, Bellow – Their corresponding reconstructed 3D-models.

6.2 Using Reduced Number of Images

Up till now, we have used all T = 36 or 18 images for 3D acquisition. We can make an arrangement to use reduced number of images. This can be done by extracting feature points on the both sides of axis of rotation. The point on the right of the axis of rotation can be thought of as being extracted from the image which is 180° ahead of current image. In this way, by using first T/2 number of images we can construct complete 3D object. But for texture acquisition we have to use all images in order to capture all information. Figure 6.3 shows object which is reconstructed using T = 4 number of views (i.e. $\theta = 90°$). Only first two images are used to extract feature points and all four images are used to acquire texture information.

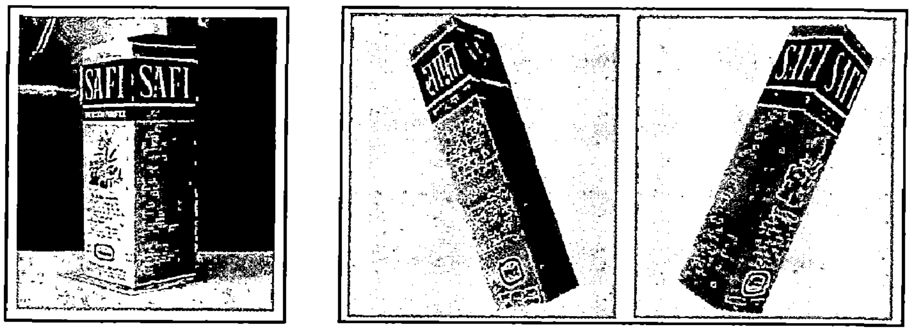
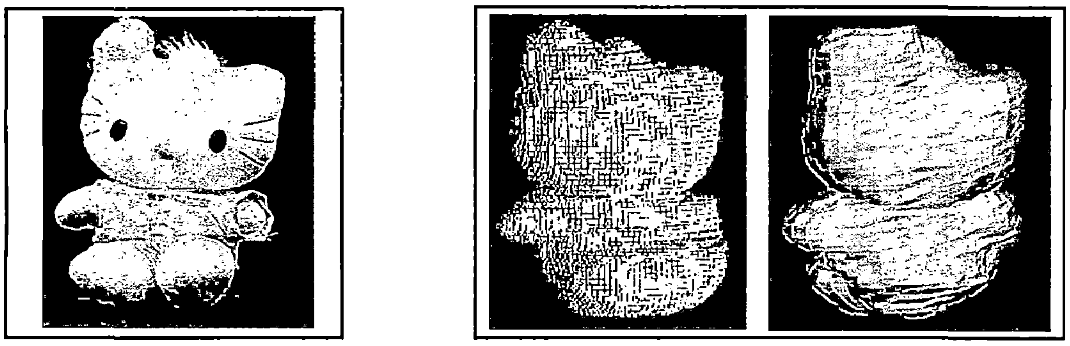


Figure 6.3: 3D-model of object is reconstructed using 4 views.

6.3 Using Non-homologous images

The non-homologous images exhibit structural differences amongst them. In APPENDIX-A some sample of non-homologous images are shown. Though there are structural differences they can be used to reconstruct 3D-models.

As these images are the resultant of image acquisition process discussed in section 4.1 and due to the inherent nature of single axis motion, these differences between two images taken by adjacent cameras are not too much if we consider large number of views. i^{th} and $i \pm 1^{th}$ images slightly differ in their structure as the camera is rotated by an angle θ. Usually, T = 36 number of images are enough to reconstruct complete 3D-model as shown in Figure 6.4.



(a) (b) (c)

Figure 6.4: (a) Input image of Toy, (b) Output using Voxelization, (c) Output using Proposed Method.

6.4 Limitations and Accuracy

Limitations

1. As we have avoided camera calibration, we are able to derive 3D-models up to an unknown scale factor. But it is found that object maintains shape integrity.
2. As the case with other image-based modeling, this method also suitable for convex surfaced objects only.
3. This technique is extremely view as well as image dependent.
4. Texture mapping is highly sensitive to camera zooming and skew factor.

Accuracy

The accuracy of any reconstruction depends on several factors:

1. The number of images containing views of the same points increases the number of rays back-projected when estimating the position of a point. This should reduce the effect of errors introduced by image noise.

2. The distance between the camera centers, known as the baseline. If the baseline is small, the angle between the back-projecting rays will be small, and image noise can produce a large error in back-projection. However, if there is a large baseline, the back-projecting rays are generally well-conditioned and the image noise has a smaller effect.

3. The accuracy of the information known about the camera, the motion involved, and the objects in the scene, including:

   – The camera motion. If for example, the camera is assumed to only translate, with no rotation, then how close is the actual motion to this assumption?

   – Assuming a set of points lies on a planar surface in the scene.

   – The camera calibration.

CHAPTER 7. CONCLUSION AND FUTURE WORK

7.1 Conclusion

This thesis has explored the 3D reconstruction of real-world generic objects using multiple homologous and view dependent images obtained by natural hand held camera. An algorithm for finding feature points has been successfully implemented. Furthermore, these points are used for surface formation with its surface properties, in order to implement texture mapping.

Following conclusions can be drawn:

- To avoid the need for camera calibration the constrained and controlled environment can be provided by choosing aspect ratio to unity, ignoring skew factor and choosing principal point to coincide with the centre of image.
- Therefore, the proposed method allows a computer to automatically generate a realistic 3D model when provided with a sequence of images of an object or scene.
- The results show that the 3D-models are fairly accurate and can be obtained from homologous as well as non-homologous images.
- The technique employed is simple and straightforward.

7.2 Future work

While much research has been conducted in 3D reconstruction and reconstructions are becoming increasingly photorealistic, improvements are still needed in order to accurately and efficiently recover the 3D object from images.

The approach presented in this thesis can be extended in a number of ways. Geometric accuracy, realistic surface reflectance and methods to account for complex large-scale dynamic environments, and real-time 3D reconstruction from video sequences remain areas of research and development in the field of 3D scene reconstruction.

REFERENCES

[1] Carlo Colombo, Alberto Del Bimbo, and Federico Pernici, "*Metric 3D Reconstruction and Texture Acquisition of Surfaces of Revolution from a Single Uncalibrated View*", IEEE Transactions on Pattern Analysis And Machine Intelligence, Vol. 27 (1), January 2005.

[2] C. Colombo, D. Comanducci, A. Del Bimbo, and F. Pernici, "*Accurate Automatic Localization of Surfaces of Revolution for Self-Calibration and Metric Reconstruction*", Proc. IEEE Workshop Perceptual Organization in Computer Vision, 2004.

[3] Kristen Grauman, Gregory Shakhnarovich, and Trevor Darrell, "*Inferring 3D Structure with a Statistical Image-Based Shape Model*", MIT Computer Science and Artificial Intelligence Laboratory, pp. 483–484, 2004.

[4] Kristen Grauman, Gregory Shakhnarovich, and Trevor Darrell, "*Virtual Visual Hulls: Example-Based 3D Shape Inference from Silhouettes*", MIT Computer Science and Artificial Intelligence Laboratory, pp. 485–486, 2004.

[5] G. Jiang, H. Tsui, L. Quan, and A. Zisserman, "*Geometry of Single Axis Motions Using Conic Fitting*", IEEE Trans. Pattern Analysis and Machine Intelligence, Vol. 25 (10), pp. 1343-1348, October 2003.

[6] Yen-Hsiang Fang, Hong-Long Chou, and Zen Chen, "*3D shape recovery of complex objects from multiple silhouette images*", Pattern Recognition Letters, Vol. 24, pp 1279–1280, June 2003.

[7] K.-Y. K.Wong, P. R. S. Mendonca, and R. Cipolla, "*Camera calibration from surfaces of revolution,*" IEEE Trans. on Pattern Analysis and Machine Intelligence, Vol. 25 (2), pp. 147–161, February 2003.

[8] Kuzu Yasemin, and Sinram O, "*Volumetric Reconstruction of Cultural Heritage Artifacts*", CIPA XIXth International Symposium, Antalya, Turkey, pp. 93–98, 2003.

[9] Vitor Sequeira, and Joao G.M. Goncalves, "*3D Reality Modelling: Photo-Realistic 3D Models of Real World Scenes*", Proc. of the First International Symposium on 3D Data Processing Visualization and Transmission, 2002.

[10] Dinesh K. Pai, Kees van den Doel, Doug L. James, Jochen Lang, John E. Lloyd, Joshua L. Richmond, and Som H. Yau, "*Scanning Physical Interaction Behavior of 3D Objects*", ACM SIGGRAPH, August 2001.

[11] G. Slabaugh, W. B. Culbertson, T. Malzbender, and R. Schafer, "*A survey of volumetric scene reconstruction methods from photographs*", Proc. of Joint IEEE TCVG and Eurographics Workshop, pp. 81–100, June 2001.

[12] Kuzu Yasemin, and Rodehorst Volker, "*Volumetric modeling using shape from silhouette*", Fourth Turkish-German Joint Geodetic Days, pp. 469–476, 2001.

[13] Caleb Lyness, Otto-Carl Marte, Bryan Wong, and Patrick Marais, "*Low-Cost Model Reconstruction from Image Sequences*", First International Conference on Computer Graphics, Virtual Reality and Visualization, pp 131–132, 2001.

[14] Arne Henrichsen, "*3D Reconstruction and Camera Calibration from 2D Images*", Dissertation University of Cape Town, pp. 8–30, December 2000.

[15] Arzu Coltekin, Jussi Heikkinen, Petri Ronnholm, "*Studying Geometry, Color and Texture in VRML*", Journal Article (English), Surveying Science in Finland, Vol. 17 (1), pages 65-90, October 1999.

[16] M. Pollefeys, "*Self-Calibration and Metric 3D Reconstruction from Uncalibrated Image Sequences*", PhD thesis, ESAT-PSI, K.U. Leuven, 1999.

[17] David P. Gibson, Neill W. Campbell, and Barry T. Thomas, "*The Generation of 3-D Models Without Camera Calibration*", Computer Graphics And Imaging, ACTA Press, pp 146–149, June 1998.

[18] Z. Zhang, "*Determining the Epipolar Geometry and its Uncertainty: A Review*", The International Journal of Computer Vision, Vol. 27 (2), pp.161–195, March 1998.

[19] Frederick M. Weinhaus Venkat Devarajan, "*Texture Mapping 3D Models of Real-World Scenes*", ACM Computing Surveys, Vol. 29 (4), December 1997.

[20] Luiz Velho, Luiz Henrique de Figueiredo, and Jonas Gomes, "*A Methodology for Piecewise Linear Approximation of Surfaces*", Computer Graphics and Image Processing, pp. 2–7 ,September, 1997.

[21] E. Ofek, E. Shilat, A. Rappoport, and M. Werman, "*Highlight and reflection-independent multiresolution textures from image sequences*", IEEE Computer Graphics and Applications, Vol. 17 (6), March–April 1997.

[22] S. Moezzi, L. Tai, and P. Gerard, "*Virtual view generation for 3d digital video*", IEEE Multimedia, Vol. 4 (1), pp.18–26, January–March, 1997.

[23] S. Moezzi, A. Katkere, D. Kuramura, and R. Jain, "*Reality modeling and visualization from mulitple video sequences*", IEEE Computer Graphics and Applications, Vol. 16 (6), pp. 58–63, November 1996.

[24] M. Armstrong, "*Self-Calibration from Image Sequences,*" PhD thesis, Univ. of Oxford, England, 1996.

[25] Q.-T. Luong and O. Faugeras, "*The Fundamental matrix: theory, algorithms, and stability analysis*", The International Journal of Computer Vision, Vol.1 (17), pp.43–76, 1996.

[26] C. Montani, R. Scanteni, and R. Scopigno, "*Discretized Marching Cube*", IEEE conference, pp. 281–287, 1994.

[27] Hugues Hoppe, "*Surface Reconstruction from Unorganized Points*", PhD thesis, University of Washington, pp. 15–16, 1994.

[28] P. Heckbert, "*Fundamentals of Texture Mapping and Image Warping*", MS thesis, CS Division, Univ. of California, Berkeley, 1989.

[29] Lorensen, W.E. and Cline, H.E., "*Marching Cubes: a high resolution 3D surface reconstruction algorithm,*" Computer Graphics, Proc. of SIGGRAPH, Vol. 21 (4), pp. 163–169, 1987.

[30] Canny j., "*A computational approach to edge detection*", IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 8 (6), pp. 679–698, 1986.

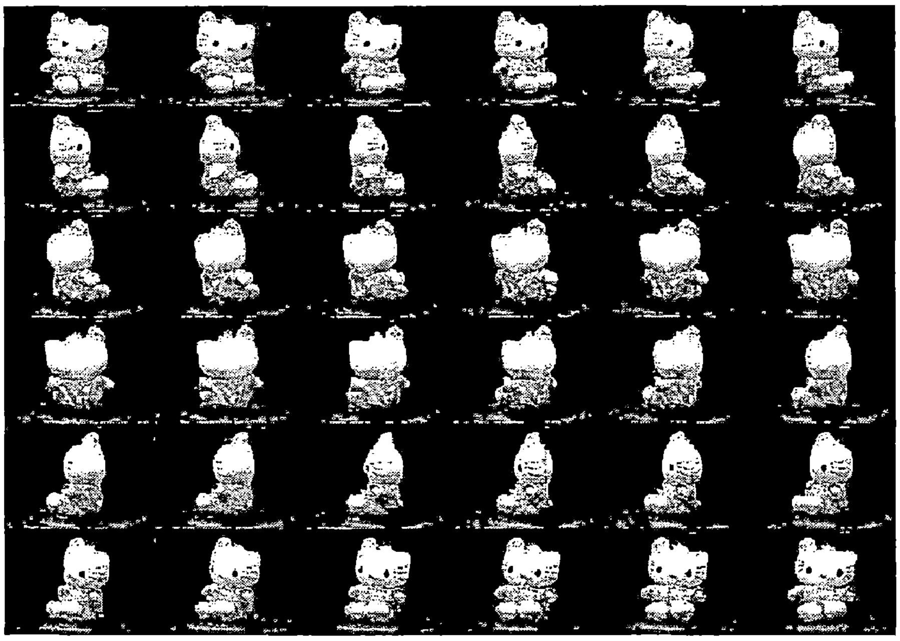
[31] Web 3D Consortium, http://web3d.org

APPENDIX

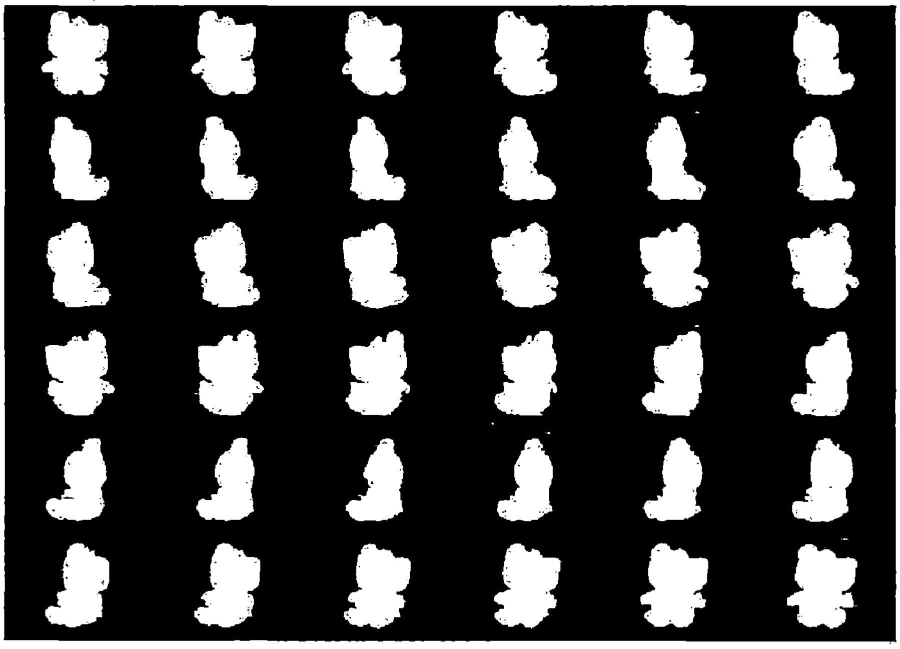
APPENDIX - A

Samples of Non-homologous Images and their silhouettes

Calibration images



Calibration images



APPENDIX – B

File Name: image_open.cpp
This file contains the code listing required to open image file and to read it

```
/////////////////////////////////////////////////////////////////////////
/*The input file is in the form of TIF format.
This code listing contains the implementation of storing and retrieving information
contained in the input file.
*/
/////////////////////////////////////////////////////////////////////////
#include "Cips.h"
#include <stdio.h>
short **allocate_image_array(long length,long width)
{
  int i;
  short **the_array;
  the_array = malloc(length * sizeof(short  *));
  for(i=0; i<length; i++){
    the_array[i] = malloc(width * sizeof(short ));
    if(the_array[i] == '\0'){
      printf("\n\tmalloc of the_image[%d] failed", i);
    } /* ends if */
  } /* ends loop over i */
  return(the_array);
} /* ends allocate_image_array */

read_tiff_image(char   image_file_name[],short   **the_image)
{
  char  *buffer, /* CHANGED */
      rep[80];
  int  bytes_read,
      closed,
      position,
      i,
      j;
  FILE  *image_file;
  float a;
  long  line_length, offset;

  struct tiff_header_struct image_header;
  read_tiff_header(image_file_name, &image_header);

   /***********************************************
   *   Procedure:
   *  Seek to the strip offset where the data begins. Seek to the first line you want.
   *  Loop over the lines you want to read. Seek to the first element of the line.
   *  Read the line. Seek to the end of the data in that line.
   ***********************************************/
  image_file = fopen(image_file_name, "rb");
  if(image_file != NULL){
    position = fseek(image_file,
                image_header.strip_offset,
                SEEK_SET);
    for(i=0; i<image_header.image_length; i++){

      bytes_read   = read_line(image_file, the_image,
                        i, &image_header,
                        0, image_header.image_width);
    } /* ends loop over i  */
    closed = fclose(image_file);
  } /* ends if file opened ok */
  else{
    printf("\nRTIFF.C> ERROR - cannot open ravi2 "
         "tiff file");
  }
} /*  ends read_tiff_image */

 /*********************************************
  *  read_line(...
  *  This function reads bytes from the TIFF. file into a buffer, extracts the numbers
  *    from that buffer, and puts them into a  ROWSxCOLS array of shorts. The
process
  *  depends on the number of bits per pixel used in the file (4 or 8).
  *********************************************/
read_line(FILE  *image_file,short  **the_image,int  line_number,
       struct tiff_header_struct *image_header,int  ie,int  le)
{
  char  *buffer, first, second;
  float a, b;
  int bytes_read, i;
  unsigned int bytes_to_read;
  union short_char_union scu;
  buffer = (char  *) malloc(image_header->image_width * sizeof(char ));
  for(i=0; i<image_header->image_width; i++)
    buffer[i] = '\0';
     /*******************************************
      *  Use the number of bits per pixel to
      *  calculate how many bytes to read.
      *******************************************/
 bytes_to_read = (le-ie)/
              (8/image_header->bits_per_pixel);
  bytes_read    = fread(buffer, 1, bytes_to_read,
                  image_file);
   for(i=0; i<bytes_read; i++){
     /*******************************************
      *  Use unions defined in cips.h to stuff bytes into shorts.
      *******************************************/

     if(image_header->bits_per_pixel == 8){
      scu.s_num        = 0;
      scu.s_alpha[0]      = buffer[i];
   the_image[line_number][i] = scu.s_num;
  }  /* ends if bits_per_pixel == 8 */
  if(image_header->bits_per_pixel == 4){
   scu.s_num           = 0;
   second              = buffer[i] & 0X000F;
   scu.s_alpha[0]      = second;
   the_image[line_number][i*2+1] = scu.s_num;
   scu.s_num           = 0;
   first               = buffer[i] >> 4;
   first               = first & 0x000F;
   scu.s_alpha[0]      = first;
   the_image[line_number][i*2] = scu.s_num;
  }  /* ends if bits_per_pixel == 4 */
 }  /*  ends loop over i  */
 free(buffer);
 return(bytes_read);
}  /* ends read_line  */

read_tiff_header( char file_name[],struct tiff_header_struct *image_header)
{
 char buffer[12], response[80];
 FILE *image_file;
 int  bytes_read, closed, i, j, lsb, not_finished, position;
 long  bits_per_pixel,  image_length,  image_width,length_of_field,  offset_to_ifd,
strip_offset, subfile, value;
 short  entry_count,  field_type,  s_bits_per_pixel,s_image_length,s_image_width,
s_strip_offset, tag_type;
 image_file = fopen(file_name, "rb");
 if(image_file == NULL)
 {              printf("\n Warning %s" , file_name);
                exit(0);
 }
 if(image_file != NULL){
    /*************************************
    *  Determine if the file uses MSB first or LSB first
    *************************************/
 bytes_read = fread(buffer, 1, 8, image_file);
 if(buffer[0] == 0x49)
   lsb = 1;
 else
   lsb = 0;

    /*************************************
    * Read the offset to the IFD       *
    *************************************/
 extract_long_from_buffer(buffer, lsb, 4, &offset_to_ifd);
 not_finished = 1;
 while(not_finished){
    /*************************************
    *      *  Seek to the IFD and read the  entry_count, i.e. the number of
    *   entries in the IFD.      *
 *************************************/
position  = fseek(image_file, offset_to_ifd, SEEK_SET);
bytes_read = fread(buffer, 1, 2, image_file);
extract_short_from_buffer(buffer, lsb, 0,&entry_count);
 /*************************************
 **  Now loop over the directory entries. Look only for the tags we need.  These
 *   are: ImageLength ImageWidth BitsPerPixel(BitsPerSample) StripOffset
 *****************************************/
for(i=0; i<entry_count; i++){
 bytes_read = fread(buffer, 1, 12, image_file);
 extract_short_from_buffer(buffer, lsb, 0, &tag_type);
 switch(tag_type){
   case 255: /* Subfile Type */
     extract_short_from_buffer(buffer, lsb, 2,&field_type);
     extract_short_from_buffer(buffer, lsb, 4, &length_of_field);
     extract_long_from_buffer(buffer, lsb, 8, &subfile);
     break;
   case 256: /* ImageWidth */
     extract_short_from_buffer(buffer, lsb, 2, &field_type);
     extract_short_from_buffer(buffer, lsb, 4, &length_of_field);
     if(field_type == 3){
      extract_short_from_buffer(buffer, lsb, 8,&s_image_width);
      image_width = s_image_width;
     }
     else
      extract_long_from_buffer(buffer, lsb, 8,&image_width);
     break;
   case 257: /* ImageLength */
     extract_short_from_buffer(buffer, lsb, 2, &field_type);
     extract_short_from_buffer(buffer, lsb, 4, &length_of_field);
     if(field_type == 3){
      extract_short_from_buffer(buffer, lsb, 8, &s_image_length);
      image_length = s_image_length;
     }
     else
      extract_long_from_buffer(buffer, lsb, 8, &image_length);
     break;

   case 258: /* BitsPerSample */
     extract_short_from_buffer(buffer, lsb, 2, &field_type);
     extract_short_from_buffer(buffer, lsb, 4,&length_of_field);
     if(field_type == 3){
      extract_short_from_buffer(buffer, lsb, 8, &s_bits_per_pixel);
      bits_per_pixel = s_bits_per_pixel;
     }
     else
      extract_long_from_buffer(buffer, lsb, 8,&bits_per_pixel);
     break;
   case 273: /* StripOffset */
     extract_short_from_buffer(buffer, lsb, 2, &field_type);
     extract_short_from_buffer(buffer, lsb, 4,  &length_of_field);
```

```
        if(field_type == 3){
         extract_short_from_buffer(buffer, lsb, 8,  &s_strip_offset);
         strip_offset = s_strip_offset;
        }
        else
         extract_long_from_buffer(buffer, lsb, 8, &strip_offset);
        break;
      default:
        break;
    } /* ends switch tag_type */
   } /* ends loop over i directory entries */
   bytes_read = fread(buffer, 1, 4, image_file);
   extract_long_from_buffer(buffer, lsb, 0,&offset_to_ifd);
   if(offset_to_ifd == 0) not_finished = 0;
  } /* ends while not_finished */
  image_header->lsb            = lsb;
  image_header->bits_per_pixel = bits_per_pixel;
  image_header->image_length    = image_length;
  image_header->image_width     = image_width;
  image_header->strip_offset    = strip_offset;
  closed = fclose(image_file);
  } /* ends if file opened ok */
  else{
   printf("\n\nTIFF.C> ERROR - could not open ravi1"
       "tiff file");
  }
} /* ends read_tiff_header */


/****************************************
 * *  extract_long_from_buffer(...
 *  *   This takes a four byte long out of a buffer of characters. It is important to
know the byte order LSB or MSB.
 * ***************************************/
extract_long_from_buffer(char buffer[],int lsb,int start,long *number)
{
  int i;
  union long_char_union lcu;
  if(lsb == 1){
   lcu.l_alpha[0] = buffer[start+0];
   lcu.l_alpha[1] = buffer[start+1];
   lcu.l_alpha[2] = buffer[start+2];
   lcu.l_alpha[3] = buffer[start+3];
  } /* ends if lsb = 1 */
  if(lsb == 0){
   lcu.l_alpha[0] = buffer[start+3];
   lcu.l_alpha[1] = buffer[start+2];
   lcu.l_alpha[2] = buffer[start+1];
   lcu.l_alpha[3] = buffer[start+0];
  } /* ends if lsb = 0     */
  *number = lcu.l_num;
```

```
} /* ends extract_long_from_buffer */
 /***************************************
  * *   extract_short_from_buffer(...This takes a two byte short out of a buffer of
characters.  *  It is important to know the byte order  *  LSB or MSB.
  *  ****************************************/
extract_short_from_buffer(char buffer,int lsb, int start,short* number)
{  int i;
  union short_char_union lcu;
  if(lsb == 1){
    lcu.s_alpha[0] = buffer[start+0];
    lcu.s_alpha[1] = buffer[start+1];
  } /* ends if lsb = 1 */
  if(lsb == 0){
    lcu.s_alpha[0] = buffer[start+1];
    lcu.s_alpha[1] = buffer[start+0];
  } /* ends if lsb = 0     */
  *number = lcu.s_num;
 } /* ends extract_short_from_buffer */
```

File Name: My_Binary_Matrix.h
This file contains code listing to store and retrieve binary matrix of 0s and 1s.

```
class My_Binary_Matrix
{
	private:
		unsigned int * matrix;
		int col_size;
	public:
		My_Binary_Matrix();
		My_Binary_Matrix(int,int);////to get size of matrix
		void set_val_of_row_col(int no_of_rows,int no_of_cols);
		unsigned int *get_address(int r,int c);
		void assign_value_at(int r,int c,int value);
		int get_value_from(int r,int c);
		unsigned int get_u_int_value(int r,int c);
};
```

File Name: My_Binary_Matrix.cpp
This file contains code listing to store and retrieve binary matrix of 0s and 1s.

```
#include "My_Binary_Matrix.h"
#include <stdlib.h>
#include <iostream.h>
#include <string.h>

My_Binary_Matrix::My_Binary_Matrix()
{
	matrix = NULL;
	col_size = 0;
}

My_Binary_Matrix::My_Binary_Matrix(int no_of_rows,int no_of_cols)
{
	if((no_of_cols % (8 * sizeof(unsigned int))) == 0)
		col_size = no_of_cols / (8 * sizeof(unsigned int));
	else
		col_size = no_of_cols / (8 * sizeof(unsigned int)) + 1;

	matrix = new unsigned int[no_of_rows * col_size];
	for(int i = 0;i < 3;i++)
		for(int j = 0;j < 3;j++)
		{
			matrix[i * col_size + j] = 0;
		}
}
void My_Binary_Matrix::set_val_of_row_col(int no_of_rows,int no_of_cols)
{
	if((no_of_cols % (8 * sizeof(unsigned int))) == 0)
```

```
                col_size = no_of_cols / (8 * sizeof(unsigned int));
        else
                col_size = no_of_cols / (8 * sizeof(unsigned int)) + 1;
        matrix = new unsigned int[no_of_rows * col_size];
        for(int i = 0;i < 3;i++)
                for(int j = 0;j < 3;j++)
                {
                        matrix[i * col_size + j] = 0;
                }
}
void My_Binary_Matrix::assign_value_at(int r,int c,int value)
{
        if(value == 0 ||value == 1)
        {
                unsigned int * address;
                address = get_address(r,c / (8 * sizeof(unsigned int)));
                if (value == 0)
                {
                        unsigned int temp = 2147483648;//32768;
                        temp = temp >> c % (8 * sizeof(unsigned int));
                        temp = ~ temp;
                        *address = *address & temp;
                }
                else
                {
                        unsigned int temp = 2147483648;
                        temp = temp >> c % (8 * sizeof(unsigned int));
                        *address = *address | temp;
                }
        }
        else
        {
                cout<<endl<<"Error : Binary Matirix Element can't be other than 0 or
                1"<<endl;
                exit(0);

        }
}

int My_Binary_Matrix::get_value_from(int r,int c)
{
        ///check for valid r, c - here is no check made in this function

        unsigned int temp;
        temp = get_u_int_value(r,c);
        temp = temp << c % (8 * sizeof(unsigned int));
        temp = temp >> (8 * sizeof(unsigned int)) - 1;
        return temp;

}
unsigned int * My_Binary_Matrix::get_address(int r,int c)
```

```
{
        return &matrix[r * col_size + c];
}
unsigned int My_Binary_Matrix::get_u_int_value(int r,int c)
{
        return matrix[r * col_size + c/ (8 * sizeof(unsigned int))];
}
```

File Name: global.h
This file contains code listing for the global parameters

```
#define PI 3.14159265358979


#define HEIGHT 576
#define WIDTH 768

#define h_DIST 1
#define w_DIST 1
#define SIZE ((h_DIST)*(w_DIST))
#define INC_SIZE ((h_DIST+1)*(w_DIST+1))
#define PERCENTAGE 1

#define ANGLE 10
#define No_Of_Views 36

#define h_RANGE 9
#define w_RANGE 6

//#define Scale 40 -- safi
#define Scale 0

int Total_Pieces_wrt_Height;
```

File Name: main.cpp
This file contains code listing for extracting feature points and formation of surfaces

```
#include "rvrml.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "global.h"

#include "image_open.h"
#include "My_Binary_Matrix.h"
//#include "octree_node.h"
//#include "process_image.h"
int h_Next_of_first_img,h_Next_of_Next_img = 0;


//void find_matrix(short **image_i,char  matrix[][WIDTH]);
void find_matrix(short **image_i,My_Binary_Matrix  &matrix);

void main()
{
        int im,i,j;
        My_Binary_Matrix matrix[No_Of_Views];
        char s[20];
        char filename[100];
        short **image_i;

        image_i = allocate_image_array(HEIGHT, WIDTH);
        for(im = 0;im < No_Of_Views; im++)
        {
                strcpy(filename,"G:\\Try\\image_source\\New_Folder_61\\bw_shoe");
                sprintf(s,"%d",((im % No_Of_Views)*ANGLE));  //*N
                strcat(filename,s);
                strcat(filename,".tif");
                read_tiff_image(filename, image_i);
                matrix[im].set_val_of_row_col(HEIGHT, WIDTH);
                find_matrix(image_i,matrix[im]);
        }
        free_memory_of_image_array(image_i,HEIGHT, WIDTH);

        point first_point,second_point,third_point,fourth_point;

        int c= 0,Point_Flag = 0, Write_Flag = 0;

        rvrml v;


        for(im = 1; im <= No_Of_Views; im++)
        //im = 1;
        {
```

```
        /// for texture map

strcpy(filename,"G:\\martch_cube\\Try_marching\\output\\temp_file.txt");
        v.set_temp_file_name(filename);
        strcpy(filename,"colored_shoe");
        strcat(filename,".jpg");
        v.new_shape(filename);

        ////texture
        cout << "Processing "<<im<<" image"<<endl;

        for(i = 0;i < HEIGHT; i+= h_RANGE)
        {
                if(i >= HEIGHT)
                        break;
                for(j = 0;j <= WIDTH/2; j++)
                {
                        if(matrix[im%  No_Of_Views].get_value_from(i,j)  ==
1)
                        {
                                first_point.p = i;
                                first_point.q = j;
                                first_point.view = im% No_Of_Views;
                                Write_Flag = 1;
                                Total_Pieces_wrt_Height++;
                                break;
                        }
                }
                for(j = 0;j <= WIDTH/2; j++)
                {

  if(matrix[(im+1)%No_Of_Views].get_value_from(i,j)== 1)
                        {
                                second_point.p = i;
                                second_point.q = j;
                                second_point.view = (im+1) % No_Of_Views;
                                Write_Flag = 1;
                                break;
                        }
                }
                if(Point_Flag == 0 && Write_Flag == 1)
                {
                        ////first ponit
                        v.write(convert_2d_to_3d_point(first_point,-1));
                        ////second ponit
                        v.write(convert_2d_to_3d_point(second_point,-1));
                }
                else
                {
                        if(Write_Flag == 1)
                        {
```

```
                                        ////third ponit
                                        v.write(convert_2d_to_3d_point(first_point,-1));

                                        ////fourth point
                                        v.write(convert_2d_to_3d_point(second_point,-
1));
                                }
                        }
                        Point_Flag = (Point_Flag + 1) % 2;
                        Write_Flag = 0;
                }
                v.end_shape(im);
                Total_Pieces_wrt_Height = 0;
        }
        cout<<endl<<"c = "<<c;

}

void find_matrix(short **image_i,My_Binary_Matrix  &matrix)
{
        int i,j;
        for (i=0;i<HEIGHT;i++)
        {
                for(j=0;j<WIDTH;j++)
                {
                        matrix.assign_value_at(i,j,0);
                        if (image_i[i][j] == 255) ///for back black & object white
                        //if (image_i[i][j] == 0) ///for back white & object black
                        {
                                image_i[i][j]=1;

                                //v.write('1');
                        }
                        //else
                                //v.write(' ');

                }
        }

        int k=0,l=0,sum=0;
        //for(i = 0;i < HEIGHT ;i += h_DIST)//64
                //for(j = 0; j <WIDTH  ;j += w_DIST)//256
        for(i = 0;i < HEIGHT -h_DIST;i += h_DIST)//64
                for(j = 0; j <WIDTH -w_DIST ;j += w_DIST)//256
                {
                        sum=0;
                        for(k = i; k< i+h_DIST; k++)
                                for(l = j;l< j+w_DIST;l++)
                                {
                                        if(image_i[k][l] >= 1)
                                                sum++;
                }
    if(((double)sum/SIZE)>=PERCENTAGE||((double)sum/SIZE)<= 0.0 )
    {
                sum=0;
                for(k = i; k< i+h_DIST+1; k++)
                for(l = j;l< j+w_DIST+1;l++)
                {
                        if(image_i[k][l] >= 1)
                                sum++;
                }
    }
            if(((double)sum / INC_SIZE) >= PERCENTAGE )
                continue;
            else
            if(((double)sum / INC_SIZE) <= 0.0)
                continue;
            else
            {
                //cout<<endl<<((double)sum/SIZE);
                for(k = i;k < i+h_DIST+1;k++)
                {
                        if(image_i[k][j] >= 1)
                                matrix.assign_value_at(k,j,1);
                                //cout<<"ravi";
                        if(image_i[k][j+w_DIST-1] >= 1)
                                matrix.assign_value_at(k,j+w_DIST-
1,1);
                }
                for(l = j;l < j+w_DIST+1;l++)
                {
                        if(image_i[i][l] >= 1)
                                matrix.assign_value_at(i,l,1);
                                //cout<<"ravi";
                        if(image_i[i+h_DIST-1][l] >= 1)
                                matrix.assign_value_at(i+h_DIST-1,l,1);
                }
            }

            {
                //cout<<endl<<((double)sum/SIZE);
                for(k = i;k < i+h_DIST;k++)
                {
                        if(image_i[k][j] >= 1)
                                matrix.assign_value_at(k,j,1);
                                //cout<<"ravi";
                        if(image_i[k][j+w_DIST] >= 1)
                                matrix.assign_value_at(k,j+w_DIST,1);
                }
                for(l = j;l < j+w_DIST;l++)
                {
                        if(image_i[i][l] >= 1)
                                matrix.assign_value_at(i,l,1);
                                //cout<<"ravi";
                        if(image_i[i+h_DIST][l] >= 1)
                                matrix.assign_value_at(i+h_DIST,l,1);
                    }
                }
            }
    }
```

File Name: vrml.h
This file contains code listing for VRML file creation

```
// rnode2.h: interface for the rnode class.
//
//////////////////////////////////////////////////////////////////////
#include <iostream.h>
#include <fstream.h>

struct point
{
        int p,q;///p - h direction , q - w direction
        int view;
};
struct point_3d
{
        double x,y,z;
};
point_3d convert_2d_to_3d_point(struct point,int d);
class rvrml
{
private:
        int count;

        fstream ofile,temp_file;

public:
        //ObNode rnd[20193];
        //int index ;
        //int view[20193];///// 1,-1  for +ve,-ve X
                              ///// 2,-2  for +ve,-ve Y
                              ///// 3,-3  for +ve,-ve Z
        rvrml();
        ///void write(float,float,float,int);////x,y,z,size
        void write();
        void write(char);
        void write(point_3d);
        void new_shape(char str[]);
        void end_shape(int view);
        void set_temp_file_name(char str[]);
        void write_to_temp_file(point & one,point & two,int height,int width);
        void wrt_form_temp_to_main_file();
        void close_temp_file();
        virtual ~rvrml();

};
```

File Name: vrml.cpp
This file contains code listing for VRML file creation

```
// vrml.cpp: implementation of the vrml class.
//
//////////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////////
// Construction/Destruction
//////////////////////////////////////////////////////////////////////

point_3d convert_2d_to_3d_point(point dot,int d)///d - direction form left or right
{
        point_3d temp;

        if(d == -1)
        {
                temp.x = -((dot.q - WIDTH/2 - 1)*sin((dot.view *ANGLE)*PI/180));//
                +( WIDTH-HEIGHT) );
                temp.z  =  ((dot.q  -  WIDTH/2  -  1)*cos((dot.view
                *ANGLE)*PI/180));///+( WIDTH-HEIGHT) );
                temp.y = dot.p;
        }
        else
        {
                temp.x  =  -(dot.q  -  WIDTH/2  -  1)*sin((dot.view  *ANGLE  +
180)*PI/180);
                temp.z  =  (dot.q  -  WIDTH/2  -  1)*cos((dot.view  *ANGLE  +
180)*PI/180);
                temp.y = dot.p;
        }


        return temp;
}
rvrml::rvrml()
{
        count = 0;
        //index = 0;
        ofile.open("G:/martch_cube/Try_marching/output/test_out.wrl",ios::in|ios::out
);
        ///ofile is used to as Read/Write file
        if(!ofile)
        {
                cout<<endl<<"unable to open file "<<endl;
        }

        ofile << "#VRML V2.0 utf8\n";

}
```

```
void rvrml::new_shape(char str[])
{
        ofile << "Shape { \n";
        ofile << "\t appearance Appearance { \n";
        ofile << "\t\t  material Material { \n";
        ofile << "\t\t              diffuseColor 0.81 0.71 0.23\n";
        ofile << "}";

        ofile << "\t texture ImageTexture { \n";
        ofile << "url ["<<"\""<<str<<"\"]\n";

        ofile << "\n\t\t}                     \t\t} \n";

        ofile << "geometry IndexedFaceSet  {\n";
        ofile << "\t solid TRUE\n";
        ofile << "\t coord Coordinate { \n";
        ofile << "\t              point [\n";
}
void rvrml::end_shape(int view)
{
        ofile << "                         ]            #end point\n ";
        ofile << "                     }\n";

        ofile << "texCoord TextureCoordinate {\n";
   ofile << "point [\n";

        temp_file.seekg(0000L);

        for(int j = 0; j <Total_Pieces_wrt_Height;j++)
        {
                ofile  <<  (double)((view-1)  *  WIDTH/No_Of_Views)/WIDTH  <<"
"<<1-((double)(j * HEIGHT/(Total_Pieces_wrt_Height-1))/HEIGHT)<<"           ";
                ofile     <<     (double)((view-1)     *     WIDTH/No_Of_Views    +
WIDTH/No_Of_Views)/WIDTH              <<"              "<<1-((double)(j          *
HEIGHT/(Total_Pieces_wrt_Height-1))/HEIGHT);
                ofile <<endl;
        }

        close_temp_file();

        ofile << "] \n";
        ofile << "} # end textcoord\n";

        ofile << "                 coordIndex [ \n";


        for(int im = 0; im < count; im += 2)
        {
                ofile<<endl<<im+0<<","<<im+2<<","<<im+1<<",-1, ";
```

```
            ofile<<im+1<<","<<im+2<<","<<im+3<<",-1, ";
            //ofile<<im+0<<","<<im+4<<","<<im+5<<",-1, ";
            //ofile<<im+0<<","<<im+5<<","<<im+1<<",-1, ";
        }

        ofile << "                              ]\n";
        ofile << "texCoordIndex[\n";

        for( im = 0; im < count; im += 2)
        {
            ofile<<endl<<im+0<<","<<im+2<<","<<im+1<<",-1, ";
            ofile<<im+1<<","<<im+2<<","<<im+3<<",-1, ";
        }

        ofile << "] # end\n";
        ofile << "} # end geometry\n";
        ofile << "} # end shape\n";

        count = 0;
}

void rvrml::write(char ch)
{
        ofile << ch;
}

void rvrml::write(point_3d dot)
{
        ofile<<dot.x<<" "<<dot.y<<" "<<dot.z<<", "<<endl;
        count++;
}

void rvrml::write()
{
        double x1 = 0,y1 = 100, z1 = 100;
        double x2 = 0,y2 = 0, z2 = 100;

        for(int im = 0; im < 36; im++)
        {
            ofile << x1 << " " << y1 << " " << z1 << ",";
            ofile << x2 << " " << y2 << " " << z2<< ",\n";
        }
}

void rvrml::set_temp_file_name(char str[])
{
        temp_file.open(str,ios::in|ios::out);
        if(!ofile)
        {
            cout<<endl<<"unable to open temp file :"<<str<<endl;
            exit(0);
        }
}
void rvrml::write_to_temp_file(point & one,point & two,int height,int width)
{
        point_3d temp_dot1,temp_dot2;
        temp_file <<(double)one.q / width<<" "<<1- (double)one.p / height<<" ";

        temp_dot1 = convert_2d_to_3d_point(one,-1);
        temp_dot2 = convert_2d_to_3d_point(two,-1);

        temp_file    <<(double)(one.q    +    Scale    +    sqrt(pow((temp_dot1.x    -
temp_dot2.x),2)    +    pow((temp_dot1.z    -    temp_dot2.z),2)))/width<<"    "<<1-
(double)one.p / height<<" ";
}
void rvrml::close_temp_file()
{
        temp_file.close();
}
void rvrml::wrt_form_temp_to_main_file()
{
        double temp;
        while(!temp_file.eof())
        {
                for(int i =0 ;i<1;i++)
                {
                        temp_file >> temp;
                        ofile << temp<<" ";
                        temp_file >> temp;
                        ofile << temp<<", "<<endl;
                }
        }
}
rvrml::~rvrml()
{
        ofile << "NavigationInfo{ \n";
        ofile << "type \"EXAMINE\" \n";
        ofile << "} # end NavigationInfo";
}
```