Construction of a 3D Surface Model from a Series of 2D Digital Pictures of a Solid Object

By

Tarik Ahmed Chowdhury, B. Sc. Eng.

This thesis is submitted as the fulfilment of the Requirement for the award of degree of

Master of Engineering (M.Eng.)

to

Dublin City University

January 2003

Research Supervisor: Professor M. S. J. Hashmi

School of Mechanical & Manufacturing Engineering

DECLARATION

I hereby certify that this material, which I now submit for assessment on the programme of study leading to the award of Master of Engineering is entirely my own work and has not been taken from the work of others save to the extent that such work has been cited and acknowledged within the text of my work

Signed: [illegible] (Candidate)

ID No.: 50162543

Date: 06-02-2003

Acknowledgements

First, I would like to express my sincere thanks and gratitude to Professor M.S.J. Hashmi for his constant encouragement and support. Without his guidance and continuous advice throughout this research it would not have been possible to finish this thesis. I am honoured to have him as my supervisor for this research.

I would also like to express my sincere thanks to Liam, school Senior Technician for his kind cooperation for the Design of my system. Without his assistance it would have been very difficult for me to complete the Design of my System.

Thanks to Keith Hicky, school IT system administrator, for his ongoing support. And also thanks to Alan Meehan, school Technician, for his kind help during the design of my system and other works.

Thanks to all members of Mechanical Engineering workshops who offered their time and valuable expertise when required.

Thanks to my fellow graduate students for their spontaneous supports, comments and particularly their joyfulness that made my hard times smoother.

Finally, I make special mention of my grand mother, who always motivated me for higher studies. She supported and understood me always and everywhere. Special thanks to her for everything she has done and given to me.

Construction of a 3D Surface Model from a series of 2D Digital Pictures of a Solid Object

by

Tarik Ahmed Chowdhury, B.Sc. Eng.

ABSTRACT

As the virtual world demands ever more realistic 3D models, attention is being increasingly focused on systems that can acquire graphical models from real objects. In this research, a system has been developed to create 3D surface model from a series of 2D digital pictures of a solid object. A rotating table and a calibration rail have been designed and constructed as part of the system, to capture images of a solid object at certain rotational angles and calibrate the camera respectively. A tripod is used to fix the camera in a certain position and background colour is selected depending on the colour of the object during image capturing. *3D Model* software was developed by the author using Visual C++, as part of the proposed system, and was used to create 3D surface models of solid objects automatically from captured digital images. 3D surface models are created based on a "silhouette based" approach. In this process, quadtree is created from the silhouette of the object image and octree is created from the voted refined quadtree. Texture mapping was performed on the octree of the object using surface particles. Surface particles are created from the outer surface of the edge octree and their colours are captured by projecting them on the proper original image. For most objects without concavities, a fairly good approximation of the surface of the objects can be obtained using the proposed system. But the proposed system still has the same limitation as previous silhouette based volumetric construction algorithm, that is, a limited precision and an inability to detect concavities. The effect of shadows and spotlights were not considered in the proposed system.

Table of content

List of Figures

List of Tables

Definitions

Camera Calibration:

Camera calibration is the process of determining the internal camera geometry and optical characteristics (intrinsic parameters) and the 3-D position of the camera frame relative to a certain world coordinate system (extrinsic parameter).

Octree:

Octree was generated by dividing a universe (generally a cube) into octants and subdividing the octants into suboctants until all the voxels (volume elements) in each octant lie entirely within an object or outside object.

Quadtree:

A quadtree is generated by dividing an image into quadrants and repeatedly subdividing the quadrants into subquadrants until each quadrant has uniform colour (e.g. “1” or “0” in a binary image). This yields a representation of an image as image as a tree of out degree four. The root node of the tree corresponds to the entire image. It is a hierarchical data structures used for compact representations of two-dimensional images

Silhouette:

The silhouette of an image is the projection of the object or objects of the image as a binary image.

Texture Mapping:

Texture mapping means the mapping of a function onto a surface in 3-D. The domain of the function can be one, two, or three-dimensional, and it can be represented by either an array or by a mathematical function. For example, 1-D texture can simulate rock strata; a 2-D texture can be represent waves, surface bumps; a 3-D texture can represent clouds, wood, marble.

Voting:

The voting means representing the 2D images into three dimensional voxel spaces with the cylindrical coordinate system.

Visual Hull:

The visual hull of an object S, is the envelop of all of the possible circumscribed cones of S. An equivalent intuition is that the visual hull is the maximum object that gives the same silhouette of S from any sample object.

Voxel:

A voxel is a unit of graphic information that defines a point in three-dimensional space.

Chapter One
Introduction

Numerous methods for acquiring 3D models of real objects have been proposed and implemented in the last decade. These methods can roughly be divided into two distinct groups based on whether they acquire the 3D shape using an active or a passive sensor. Laser range scanners and encoded light projecting systems use active triangulation to acquire precise 3D data. However they remain expensive and require special skill for the acquisition process itself. Furthermore, they work well only on matt surfaces, and it is necessary to sprinkle the surface with white powder if the object has either brilliant parts or is made of materials such as fur or velvet. Only a few scanners are capable of recording concurrently the 3D shape information with the colour texture. With those having this capability, the colour information is incorporated either by an RGB camera or by using three different laser wavelengths during the 3D acquisition process. In the first case, the colour acquisition device being generally different from the 3D one, the colour texture and the 3D data have to be registered. In the second case, the colours and the 3D information of perfectly registered but the colours obtained are not colorimetrically faithful since the spectral reflectance of the object surface is sampled at only 3 wavelengths.

Compared to active scanners, passive methods work in an ordinary environment with simple devices. The target object is pictured by a digital RGB camera from different view points, for example as it rotates on a turntable. The 3D information is then extracted from the sequence of 2D colour images by using various techniques such as multi-stereo, shape from motion or shape from silhouette, shape from shading and shape from occluding contour. There are a few other methods from that. Binocular stereo algorithms, for example, recover the third dimension (depth) from corresponding image features in two images. In static stereo, ambiguities occur during the determination of corresponding image elements. For a selected pixel of one image in general there exists more than one pixel in the other image which is a possible candidate for correspondence.

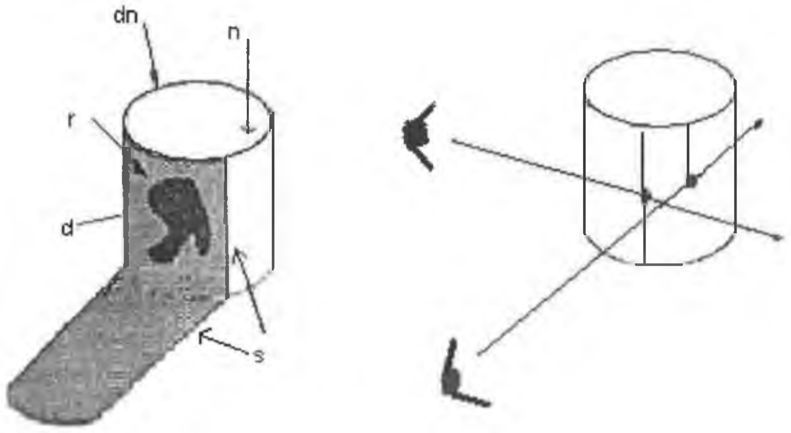


Figure 1.1. The physical origin of edges.

In the Figure 1.1 the edge labelled **d** produces the depth discontinuity because it shows instability when viewer moves. For examples, Fig. 1.1 shows that the 3-D points on the edge of the cylinder, which will produce a smooth discontinuity edge (d) in the image, moves across the surface of the cylinder as the viewer moves. The reduction of these ambiguities can be based, for example on a priori- knowledge about the parameters of the scene objects, or about value ranges of the object parameters. Shape from shading (SFS) methods can potentially determine the three dimensional shape from its irradiance in a single image by using its reflection properties. But there are a number of factors that influence the measured irradiance of an image. Firstly, the image which is acquired by the sensor depends on the geometrical and spectral distribution of the light source which illuminates the observed source. The individual objects of the scene are characterized by their geometry and by their reflection properties. The geometry and the reflection properties affect the light falling on the imaging sensor. The imaging system converts the light to measured image irradiances. Therefore, it also has a considerable influence on the measured image irradiance values. So in the SFS method of illumination, reflection, geometry and sensor are the scene factors which interact with each other. A few assumptions are required about the scene factors to reconstruct the image from a single image. However, it has to be stressed that many SFS methods, even with very restrictive assumptions, conceptually do not allow the reconstruction of an ambiguous surface. Occluding contour and silhouette are the other ways used to create the 3D image from 2D images. An occluding contour edge is formed by projecting the extreme boundary of a smooth surface. On it the viewing ray is tangent to the surface. Standard reconstruction technique such as stereo vision system perform poorly in occluding contour edges, since the geometry in three dimensions which produce these edges is unstable: the three-dimensional points from the tangency between the visual ray and the surface change with any movement in viewing position. Thus the physical origin of the edges that are produced in an image of a three-dimensional object is of great importance. In particular, external boundary points change dynamically as the viewpoints changes. The work reported in references [1-4] use the occluding contour for 3-D model construction. On the other hand, in silhouette base approach, silhouettes are taken from the sequence of images. These silhouettes are projected to create 3D conic volumes depending on the viewpoints. Hierarchical octree structures are used to represent and to process 3D volume data efficiently.

The objective of this research project was to develop a very simple and cheap system including a development of a software to reconstruct 3D surface model of real object from a series of 2D image sequences. To make system simple and cheap, this project uses the passive method and uses the shape from silhouette approach. In this method, a target object is pictured by a digital RGB camera from different view points, for example as it rotates on a turntable. The 3D information is then extracted from the sequence of 2D colour images by using shape from silhouette approach. This thesis is organized as follows: In chapter two related works are briefly reviewed. Chapter three overviews the proposed system and software developed. While in chapter four algorithm developed is described. Chapter five describes the result and discussion, and Chapter six describes the conclusion.

Chapter Two
Literature Review

2.1 Silhouette based 3D construction Review:

Schmit and Yemez [5] presented an automatic system for 3D colour reconstruction of real objects from 2D image sequences captured with a calibrated camera. The target object is pictured by a digital RGB camera from different view points, for example as it rotates on a turntable. The 3D information is then extracted from shape from silhouette approach. This approach is used to curve an octree structure and to model the object with a triangular mesh [6]. The resulting 3D triangle mesh is obtained by a marching cube technique [7] followed by a mesh decimation method [8]. The main focus of this work was the colour related task of the whole system, such as object extraction and texture mapping. In texture mapping, the most limiting factors come from the original 2D images where the colour of a surface point varies from one image to another. The colour variations are mainly due to the physical reflectance properties of the underlying materials and the geometrical variations in the viewing and lighting conditions of the object surface. In view of this colour variation problem, Schmit and Yemez [5] improved the texture mapping process by taking into account the highlight problem which is often ignored, and introduced a texture mapping strategy that avoids texture discontinuities both at triangle edges and within triangles themselves.

Matsumoto, et al [6] presented a portable three-dimensional digitizer using monocular camera. The digitizer automatically acquires the shape of a target object as well as its texture. This digitizer system consists of only three major components: a turntable, a monocular camera and a personal computer. The algorithm of this digitizer uses the "Shape-from-Silhouette" approach. This algorithm consists of five steps. In the first step, the camera calibration parameter was calculated using the calibration method reported in reference [9]. In the second step, the object is placed on the turntable and its features are taken from various viewpoints by rotating the turntable stepwise. In the third step, a silhouette image corresponding to each object image is generated based on subtraction operation. In the fourth step, the shape modelling has two processes: voting and surface representation conversion. In the voting process, the target object is represented as a set of voxels. In the surface representation conversion, the set of voxel is converted to a set of triangular patch. In the second process, a Voroni diagram, and consequently a Delaunay net [10], are generated on the object surface in the voxel space to determine which vertices should be connected. The fifth step is called texture acquisition. In this step, the texture of a patch is taken from multiple object images because a patch can be seen from various viewpoints. Therefore, an object image should be specified for each patch to take its texture. Advantages of the digitizer are:

a) compact and inexpensive

b) skill-free 3D image acquisition and

c) handles a wide range of objects of various materials.

The disadvantage of the digitizer is that some errors may arise in modelling 3D objects with concavities. This is because the concave feature can not observed as a silhouette contour.

Niem and Wingbermuhle [11] presented a novel method, which is characterized by a simple measurement environment that allows a free movement of the camera around the object and an independent choice of the focus for each camera view. This is achieved by simultaneous acquisition of the object and a newly developed calibration pattern. The new calibration pattern is placed below the object as like the following figure.

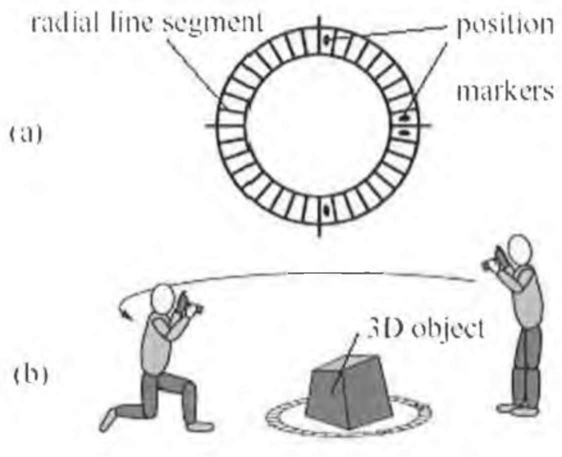


Figure 2.1 Calibration pattern and its usage within the measurement environment

In this work [11] camera calibration is done in three steps. In the first step, object, background and calibration patterns are separated in the input images. For this purpose, a colour segmentation technique in combination with a repeated erosion-dilation is used. In the second step, the 2D calibration points are estimated by intersecting the line segments and the projected circles of the calibration pattern in the image. The correspondence of the non occluded points to a fixed world coordinate system is determined by evaluation of position markers. In the third step, from the extracted 2D calibration point coordinates in the images and their corresponding 3D world coordinates, the external and internal camera parameter for each view are estimated by using an algorithm as proposed by Tsai [12]. After camera calibration, shape reconstruction is done by "shape from silhouette" approach using volume intersection technique. The principle of this approach can be divided into two steps. In the first step, a bounding pyramid is constructed using the focal point of the camera and the silhouette. In the second step, the pyramids are intersected and form an approximation of the bounding volume. Then the volume representation is transform in to surface model. In this step, the volume model is approximated by a triangular mesh, which generate the 3D-wireframe model. For natural looking of 3D models the texture is of great importance. To meet the goal, a texture algorithm is used which estimate the texture for each triangle from the input image sequence [13].

This system offers a significant gain in flexibility for the user and a drastic reduction of costs for technical equipments; hence it provides a new option for future 3D reconstruction techniques. In this proposed system one need to calibrate camera for every view. This makes the system time consuming and slow.

Martin and Aggarwal [14] presented a method to generate a volumetric description of objects from multiple views occluding contours (silhouettes). In their work, the orientation of a view is specified by a point and three mutually perpendicular directions that form a right-handed, three-axis coordinate system with the given point as origin. The y axis of each coordinate system is considered to be the direction of the line of sight with the (x,z)-plane being the image plane, i.e., the silhouette is projected onto the image plane by lines parallel to the line of sight. Thus, an occluding contour is generated by the lines that are parallel to the y-axis and that object, but does so only at points which are not on the object surface. The contours are displayed in their respective image planes and some representative contour generating lines are indicated by equal length sections of lines that are parallel to the appropriate y-axis. The set of contour generating lines for a given view defines a volume which bounds the actual object. However, this volume is by itself infinite. So the second view also defines a infinite volume that contains the object, and if the second view is distinct as like the following Figure 2.2, i.e. the two lines of sight are not parallel, then the intersection of the two volumes will still encompasses the object and will be of finite extent. So using multiple image views a bounding volume for the object can be created. The system offers a good approximation of three-dimensional object with orthogonal projection. This kind of approach can handle arbitrary shapes, provided that the object surface does not contain cavities or “holes”. Another problem with this approach is, images are taken with perspective camera, the orthogonal projection is an assumption which induces inaccuracies in the shape depending on the creditability of the assumption. It is shown in [15] that the ratio between the distance from the camera and the height of the object must be greater than 10 for the assumption to be reasonable.

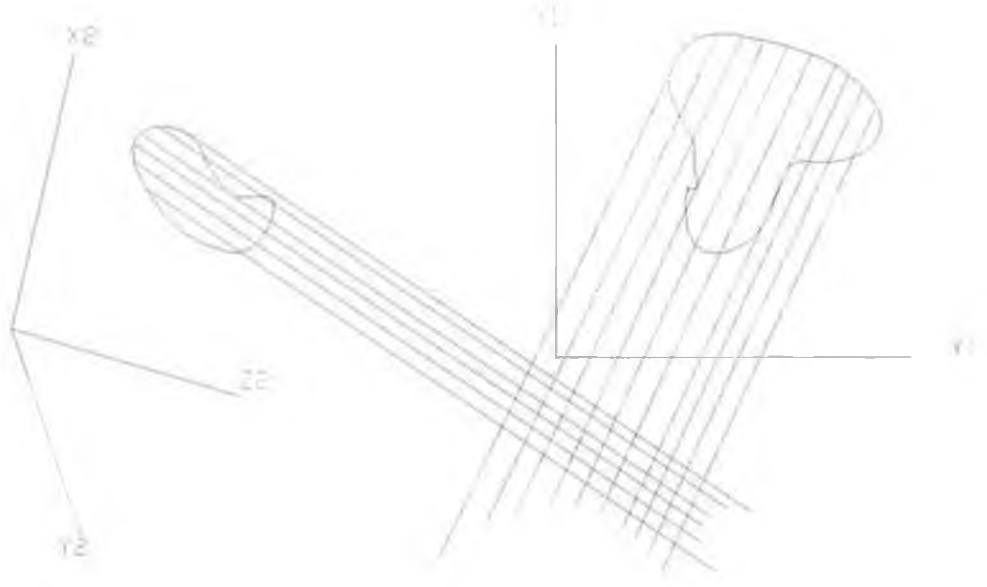


Figure 2.2: Silhouette and contour generating lines for two views.

Zheng and Kishino [16] presented a method to explore shape from contour method for acquiring a 3D model using a continuous sequence of images taken as an object rotates. Their direct goal was to establish a 3D model of a human face. They used occluding contours to recover the shape. They deal with area that is invisible when viewed only in silhouette. They found that the unexposed an area, which might be a concave or planer surface, is numerically related to the non smoothness of the contour distribution. They detect these areas by filtering contour distribution according to the model resolution required. This is very important for an algorithm in qualitative understanding of the visual world.

Chein and Aggarwal [17] proposed a new technique for shape recognition from the occluding contours of 3D objects. For shape recognition, model construction and object matching are two important issues. For model construction they used quadtrees and octrees to describe 2D silhouettes and 3D objects, respectively. They produced the quadtree structure using the technique proposed in references [18,19]. Then octree of an object was generated from three non-coplanar views using a technique known as volume intersection [20]-[22]. Given three noncoplanar views, each of them was projected onto an image plane determined by the remaining two viewing directions, which specifies the two coordinate axes of the image plane. A quadtree was generated for each image plane. Each quadtree was then copied along the associated viewing direction to obtain a pseudo-octree describing the (oblique) cylinder, which was the swept volume of the projected image along the viewing direction. Previously they proposed [14] a method called multilevel boundary search (MLBS) algorithm to determine the orientations of the surface normal at the interfaces between the object and the surrounding space from the quadtree. To improve the overall efficiency, they proposed another approach in which quadtree had four types of node: gray node, interior nodes, interior nodes and contour nodes; where nonobject nodes corresponds to nonobject nodes, and interior and contour nodes corresponds to object nodes in a regular quadtree. This quadtree is called RC(region/contour) quadtree (Figure 2.3).

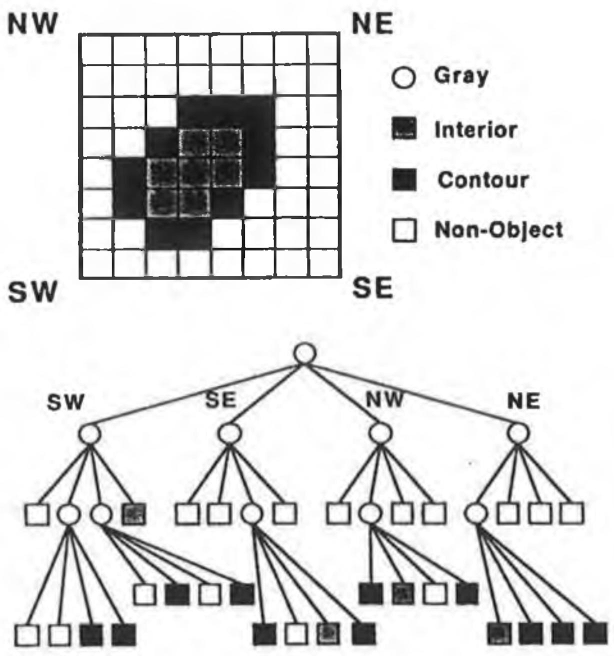


Figure 2.3 An Image and its RC quadtree representation

After creating RC octree from RC quadtree, surface information was computed according the proposed method reported in reference [20]. In the next step, they used feature points to object recognition. They extracted 2D feature points from an unknown object silhouette and match else points with the existing 3D model feature points for object recognition. They also verified this process by a 2-D contour matching. But in their proposed recognition algorithm, the underlying assumptions were:

a) the given silhouette of 3-D objects were generated using the scaled orthographic projection, and

b) there were at least four feature points present in the silhouette.

Potmesil [23] presented a method for generating octree models of 3D solid objects from their silhouettes obtained in a sequence of images. In this approach, he firstly generated 3D conic volumes from the silhouette of objects visible in an image and the projection center for each image plane. He used octree model to construct conic volumes. These conic volumes, computed in each of a sequence of views, are then intersected to generate a 3D model of the object. Each processed view of a sequence progressively refined this model. After the final 3D model had been obtained from a sequence of images, it was further improved by labeling contiguous solid volumes as individual objects, adding surface-normal vectors to the modeled objects, and mapping intensities on the surfaces of the modeled objects. The algorithm of the system is as follows:

a) accurate object-to-image space transformation (camera calibration) for each image.
b) A 3D voxel universe is created in the object coordinate system as shown in Figure 2.4.

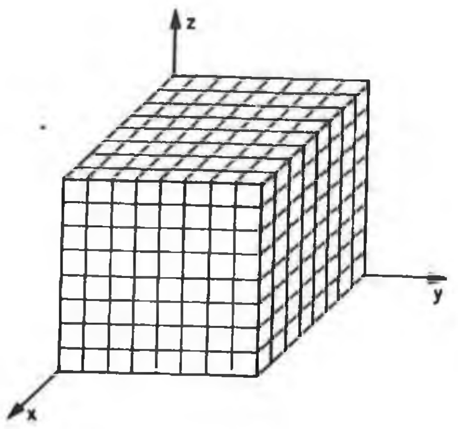


Figure 2.4 An array of voxels in object coordinate space O(x,y,z,w)

c) A conic volume is created as like the Figure 2.5 for each image.

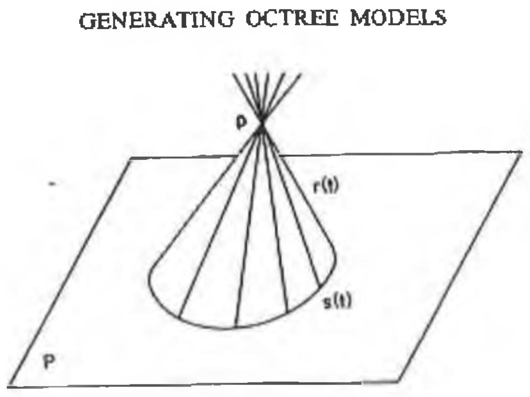


Figure 2.5: A 3D cone is a ruled surface generated by a curve s(t) ⊂ P and rulings r(t) which all pass through a point p ⊄ P, where s(t) is a silhouette, p is the center of projection, and P is the image plane.

d) octree nodes are projected on the image plane to find the intersection of the projected node with the object region. This is applied for each image.

e) The direction of a surface normal vector to a surface leaf node is estimated from the shape of a local neighborhood.

f) Texture mapped on surface nodes is done by projecting the light intensities from the image pixels into the corresponding surface nodes of the objects.

This proposed approach has some disadvantages:

a) Unable to process the concave part.

b) Camera calibration was processed for each image.

Szeliski [24] proposed a model to process each image on-line and to produce a coarse model quickly and refine it as more images are seen. Their approach was similar to the Potmesil's [23] approach, but instead of building a separate octree for each view, he intersected each new silhouette with the existing model. Their hierarchical octree construction technique was faster because of the following reason.

a) their algorithm test cubes at the finest level.

b) Their algorithm tested cubes in top-down traversal to the current finest level. The children of cubes that fall within the current silhouette were not tested.

c) Their hierarchical construction algorithm tested only cubes at the finest level whose parent's inclusions were "ambiguous" in the current silhouette. This technique was applicable when the same set of silhouettes was being recycled each resolution level.

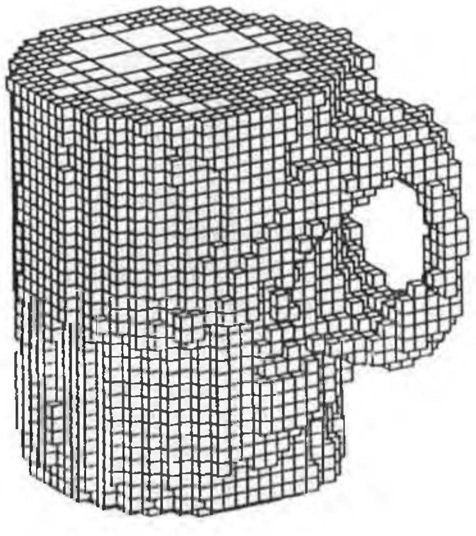


Figure 2.6: Octree model derived from a real cup images

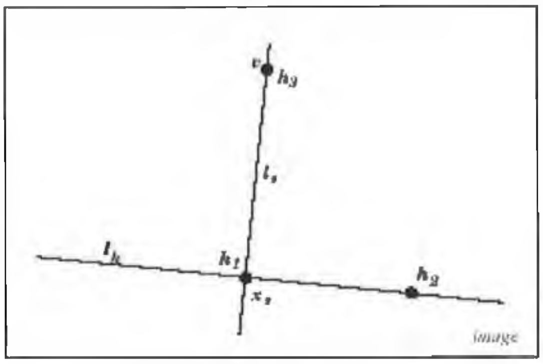
Their proposed method still has the limitation as previous silhouette-based volumetric construction algorithms, such as a limited precision and the inability to detect concavities in the object.

Fitzgibbon [25] developed the projective geometry of single axis of rotation and described its automatic and optimal estimation. It is shown that 3D structure and cameras can be estimated up to an overall two-parameter ambiguity. He has shown that the fundamental matrix F, trifocal tensor τ and camera matrices P all have additional properties, and that the multiple view tensors (F and τ) determine a two-parameter family of camera matrices. His proposed algorithm was:

1. For each pair of views the planar-motion fundamental matrix.

$$F = \mu[x_a] + \tan\frac{\Delta\theta}{2}(l_s l_h{}^{\mathrm{T}} + l_h l_s{}^{\mathrm{T}}) \quad \text{with } x_a{}^{\mathrm{T}} l_h = 0 \quad \dots\dots\dots\dots\dots\dots(2.1)$$

here θ is the rotation angle, l_s is the vertical line and l_h is the horizontal line of camera rotation.



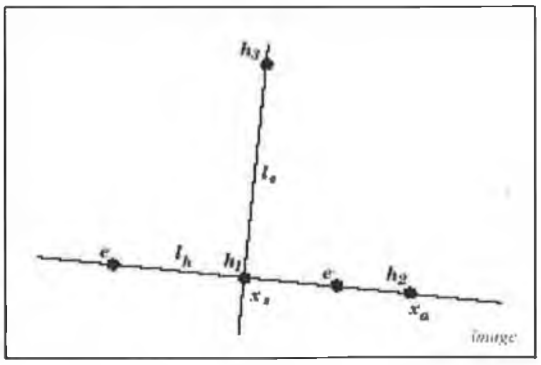


Figure 2.7 A Fixed image entities Figure 2.7 B Two-view entities

F can be expressed in terms of camera internal parameter H and Δθ.

$$F = [h_2] - \frac{1}{(\det H)}\tan\frac{\Delta\theta}{2}\left((h_1 \times h_3)(h_1 \times h_2)^{\mathrm{T}} + (h_1 \times h_3)(h_1 \times h_3)^{\mathrm{T}}\right) \dots\dots (2.2)$$

and

$$H = [h_1 \quad h_2 \quad h_3] = [x_s \quad \mu x_a \quad v x_s + \omega d] \quad \dots\dots\dots\dots\dots\dots\dots\dots\dots\dots (2.3)$$

In the case of three-view μ was produced accurately and hence Δθ was recovered uniquely. The only remaining ambiguity was in the third column h_3 of H. And this ambiguity could not be reduced by the single axis motion constrain alone.

2. From each τ_i determine μ and the two angles $\Delta\theta_i$ and $\Delta\theta_{i+1}$.

3. Average μ over the sequence, and angles from overlapping triplets.

4. Bundle adjust, varying H, θ_i and 3D points X_j to minimize reprojection error

$$\sum d^2(x_{ij}, H[R_Z(\theta_i) \mid t] X_j). \qquad \text{..............................} (2.4)$$

This algorithm provides an uncalibrated structure recovery systems based on the single-axis motion constraint. But the visual hull effect severely limits the range of models that can be acquired by the system.

Chein and Aggarwal [20] proposed a one-pass algorithm to compute the volume/surface from occluding contours and silhouette of multiple views of 3D objects. The surface information was computed directly from the occluding contours and multiple views. They used modified quadtree structure called RC (region/contour) quadtree to create octree. It had four nodes *(*Figure 2.3*)*. With the contour information encoded, each of the RC quadtree of three views was converted into RC pseudo-octree and RC pseudo-octree was used to create the VS (volume/surface) octree. Their proposed algorithm for surface computation was as follows:

If a corresponding node in either of the three quadtree is a contour node and the remaining two are interior nodes, then the surface normal $\overrightarrow{N}_s$ is in the same direction as that of the contour Normal $\overrightarrow{N}_c$.

$$\overrightarrow{N}_s = \overrightarrow{N}_c = n_x \overrightarrow{T}_x + n_y \overrightarrow{T}_y. \qquad \text{.........................} (2.5)$$

Where $\overrightarrow{T}_x$ and $\overrightarrow{T}_y$ are unit vector along X and Y direction. $\lfloor n_x, n_y \rfloor$ is the contour normal in the image coordinate system.

It is clear from the above discussion that their proposed algorithm can compute surface information at the time of octree creation. So they called it one pass algorithm. They proposed another efficient algorithm for surface refinement, if additional views of the object are available.

As this algorithm belongs to the volume intersection technique, concavities in the object cannot be constructed. It also uses parallel projection but contours are taken by perspective projection.

Lavakuja et al [26] proposed a new algorithm for constructing a 3D approximation of an object from three orthogonal 2D silhouettes. Previously the work reported in reference [22] generated octree in three major steps and they were:

1) generation of the quadtrees and pseudo octrees
2) intersection of swept volumes
3) condensation of the resulting representation to obtain an octree.

In this work, their proposed algorithm for volume intersection directly generates the linear octree of the 3D object and hence condensation was not required and made the algorithm faster than the previous algorithm reported in reference [22]. But they had the same problem as like the reference [22]. It is also restricted to three orthogonal views, unable to detect concavities, and use parallel projection.

Hiroshi et al [27] proposed algorithm was based on the volume intersection technique and used multiple polyhedral view cones to approximate the shape of the target object. The target object was approximately represented by common region, which was defined as the intersection of all view cones and called the parent region and was represented by octree. They created the octree in four stages. In the first stage, the proposed algorithm checks and classifies the eight subregions as inside, intersecting or outside each convex cone. This was done by their proposed Cone_Making procedure. It manages each convex view cone to make its classification, and also manages each convex cone partitioning a nonconvex view cone to make its classification (Figure 2.8). In the second step, using information from the first stage,

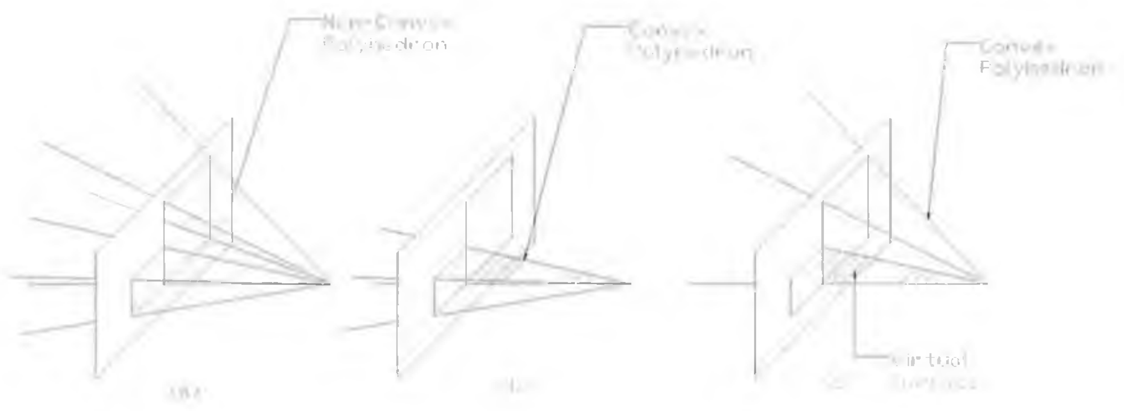
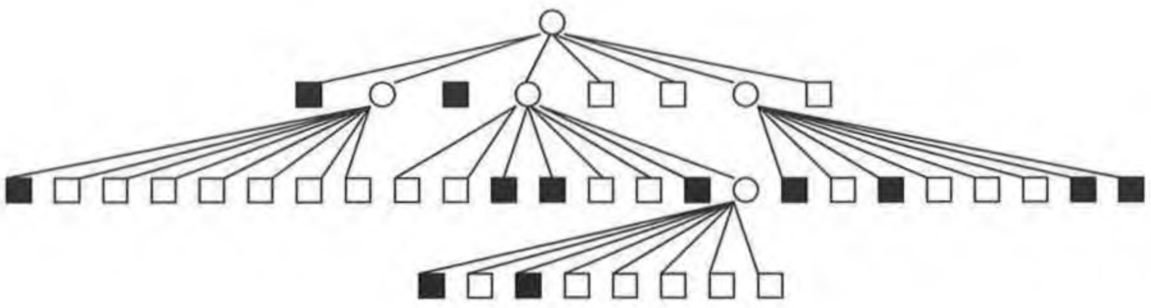


Figure 2.8 a) view cone b) and c) convex cones for the view cone.

the eight subregions were classified as inside, intersecting, or outside each nonconvex view cone. This was done by a cone Unifying Procedure. In the third step, using information from stages one and two the eight subregions were classified as inside, intersecting, or outside the common region. This was done by Cone-Intersecting procedure. In the forth step, using representation from stage three, the Depth First (DF) representation for eight subregions was generated. This was done by cone DF_Making Procedure (Figure 2.9).



■	Balck node
□	White node
○	Mix node

(a)

(1(10000000)1(0011001(10100000))00(10100011)0)

(b)

Figure 2.9 a) Octree of an object b) its DF-representation.

The Kim and Aggarwal[28] octree generation had two difficulties. The first one is, the views were limited to three orthogonal views and the other is, parallel projection was used. The algorithm of Hong and Sheneier [29] solved these difficulties by:

1) constructing the octree in an arbitrary coordinate system,
2) defining the cones by the perspective transformation, and
3) having an unlimited number of views.

The proposed algorithm in this work offers all the facility like the Hong and Sheneier algorithm but faster than the Hong and Sheneier algorithm.

As this method used volume intersection, concavities of the object cannot be detected in the created octree model.

Sanjay at el [30] presented an efficient algorithm for generating octree from multiple perspective views of an object. In the first step of their algorithm they threshold the image to get a binary silhouette. Then, the binary of the silhouette was approximated by a polygon. They then composed the polyhedral silhouette into convex components. For each convex component, the semi-finite pyramid illustrated in Figure 2.10 was constructed and then passed on the second step.

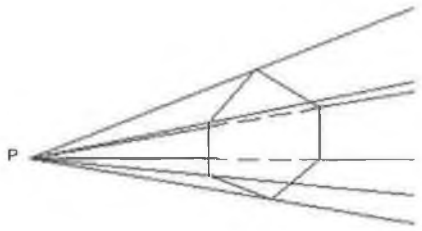


Figure 2.10: A pyramid formed by treating the view point P as its apex and the orthogonal silhouette as one of its cross section

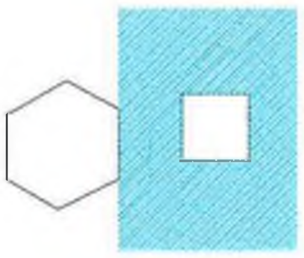


Figure 2.11: Geometry illustrating two-dimensional analog white node detected in test 3. The square will be detected to be outside the polygon if it lies in the shaded region.

In the second step the algorithm checks for intersection between a semi-infinite pyramid and upright cube corresponding to an octree node. They decomposed this intersection detection step into a period of tests, each examining a distinct geometric configuration of the cube and the pyramid. The first test approximates the pyramid by the smallest enclosing cone and the cube by the smallest enclosing sphere. If the sphere and the cone do not intersect it, the octree node is white. If they intersect, second test go on. The eight vertices of the cube are tested to see where they lie inside or outside the pyramid. If all the vertices are found to be contained inside the pyramid, the node is black; if some vertices are found to be inside and outside, the node is colored gray. If all the vertices are found to be outside, then the cube may or may not intersect with the pyramid. To resolve this problem, a third test examines the location of the cube with respect to each of the faces of the pyramid. If there is any face such that all eight vertices of the cube lie on the outside of the face, then the cube is outside the pyramid. Figure 2.11 illustrates a two-dimensional analog of this test, where a square is tested for being completely on the inside or outside of an edge polygon. Any intersections of the edges of the pyramid with the extended faces of the cube are found. If any of the points of intersection are contained inside the cube, then the node is colored gray; otherwise the node is colored white.

These steps together determine an octree for each convex component of a given polygonal silhouette. The final octree for the pyramid obtained from a single silhouette is the union of the octres obtained for all convex components of the silhouette. Such octrees obtained from the different silhouettes are intersected to obtain the final octree of the object which represents the volume of intersection of the different pyramids.

Their proposed method still has the limitation as the silhouette-based volume intersection algorithms, such as a limited precision and the inability to detect concavities in the object

2.2. Contour based 3D construction Review:

Extremal boundaries or occluding contour is one of the methods of 3D construction. They broadly fall into two classes: in the first class, the authors [31,32] assume that they have one monocular image in which they have been able to identify, by means that they do not describe, one occluding edge. They then go about describing the kind of interferences that can be drawn from this observation. Marr[31] already signalled their possible importance to infer the sign of the Gaussian curvature of the object. His observations were made more precise by Koenderink[32] who has done a fairly through job at analysing the qualitative properties of the rim and its images. In the second class, the authors assume that they have several views of the silhouette of the objects and therefore, several occluding contours. The works of Giblin and Wesis [4], Seals[2], Vaillant and Faugeras[3] and Cipolla and Blake [33] fall in this group. Giblin and Wesis [4] studies the 2-D case and gives precise equations and simulation results for reconstructing local shape from sequence of several images. They used planar motion of the camera and noise free data as input.

The work of Vaillant and Faugeras [3] was based on the multi camera rig method. They identified occluding contours from triplets of images by estimating the radius of curvature of a special planar section of the object surface: the radial curve. On the identified occluding edges, they detected the parts, if any, that are images of points on the object with zero Gaussian curvature. Then they modeled the object as the envelope of its tangent planes and used the Gauss map to compute the depth, the normal, and the second fundamental from the surface along the occluding edges. They used perspective projection and it was not restricted to orthographic projection. Their proposed method can provide a false result if there were erroneous matches or if the observed occluding contour did not corresponds to a cylinder but to a more general surface.

The goal of the work Seals and Faugers [2] was to recover an accurate and complete 3D model of shapes that contain both smooth and sharp boundaries. The fundamental components of their system were camera calibration, stereo matching, edge classification and reconstruction, multiple frame fusion and global surface fitting. Main features of their work were:

a) exploration of the special geometry to produce the occluding contour and recover and use the important surface information like surface normal and curvatures.
b) reconstruction of the complete object that could be composed of smooth surface patches as well as sharp surface discontinuities and surface marking.
c) Their edge classification schemes provided the mechanism for computing the motion between frames automatically.
d) They used clustering and meshing techniques to recover global features of an object such as protrusions and holes.

In their system, they used a trinocular camera rig in front of a rotating, calibrating table. Objects were placed on the table in an arbitrary position. After getting the occluding contour they classified as fixed and external edges which is shown in Figure 1.1. Using the edges and both known and computed motion they constructed the 3D images. They used their own surface fitting algorithm to obtaining surfaces. In their system they encountered some errors for two reasons. Firstly, error occurs because of misclassification of edges. Sometimes fixed edges are classified as external edges or vice versa. Secondly, error occurs in registering many frames of data into a common frame.

Bro-Nielsen[34] used a 3D triangle as a primitive in the bounding cone generated by the occluding contour in an image. The arcs in a polygonal approximation to the occluding contour were used to generate the individual triangles. The first step of the algorithm registered how triangles from different views collide. The second step used this information to determine quadrangles that combined to produce the surface of the minimal bounding volume. Finally, texture is mapped onto the resulting model.

In the first step, he extracted an occluding contour from silhouette images. They used this occluding contour as a directed closed polygon in an image. By connecting the end points of each polygon arc to the viewpoints of the camera each polygon arc determines a triangle in space (Figure 2.12 (a)). When more than one view of the same object is used, triangle cones from one view can and will collide with cones from other views. This will create a cut polygon in the triangle (Figure 2.12 (b)), where each arc in the polygon is generated by a triangle in the cutting cone.

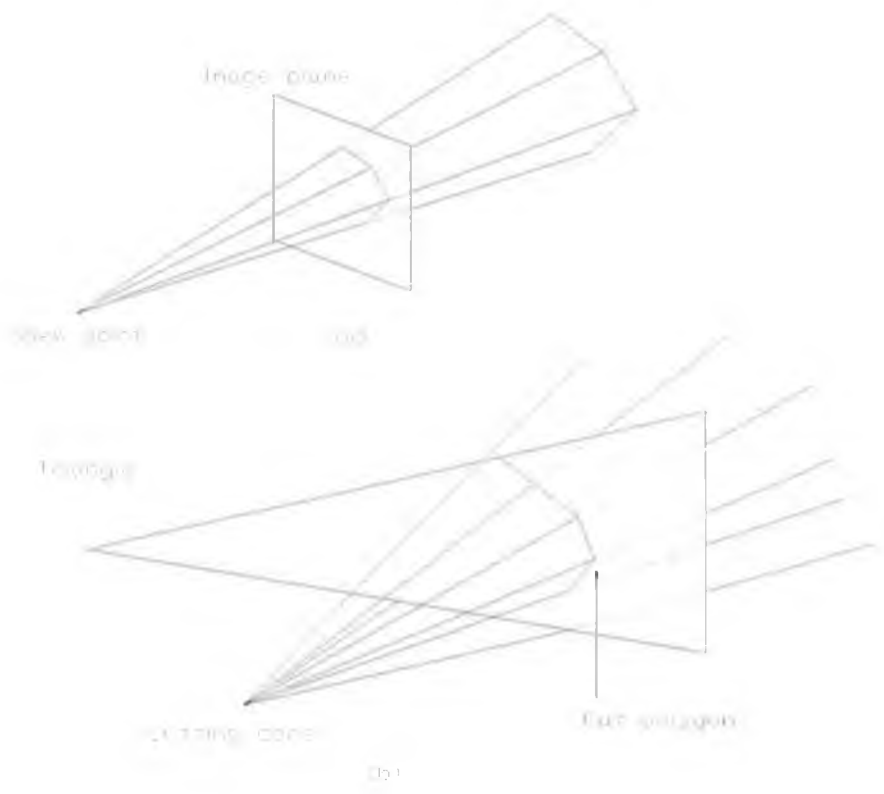


Figure 2.12: (a) Contour polygon arcs define triangles in space (b) A cut polygon is generated by a colliding triangle cone.

A triangle will usually be cut by sets of cones from many views defining many partitions on the triangle and to determine minimal bounding volume, they determine the parts of all triangles that are closet to the object. To determine minimum number of triangles, the triangle was first divided into slices defined by the nodes in the cut polygons and collision points between pairs of polygon arcs (collision nodes) (Figure 2.13). Together with the viewpoint of the triangle each node defines a line in the triangle. These lines partition the triangle into a number of slices. In the second step, quadrangular patches were produced from pairs of adjacent cut polygon arcs from these slices. The cut polygon arcs define the open bounding volume on the triangle / the slice, and these open bounding volume alone determine the object shape / the minimal bounding volume. So this method uses geometric primitives to produce bounding volume and uses orthogonal projection to make the system faster and simple.

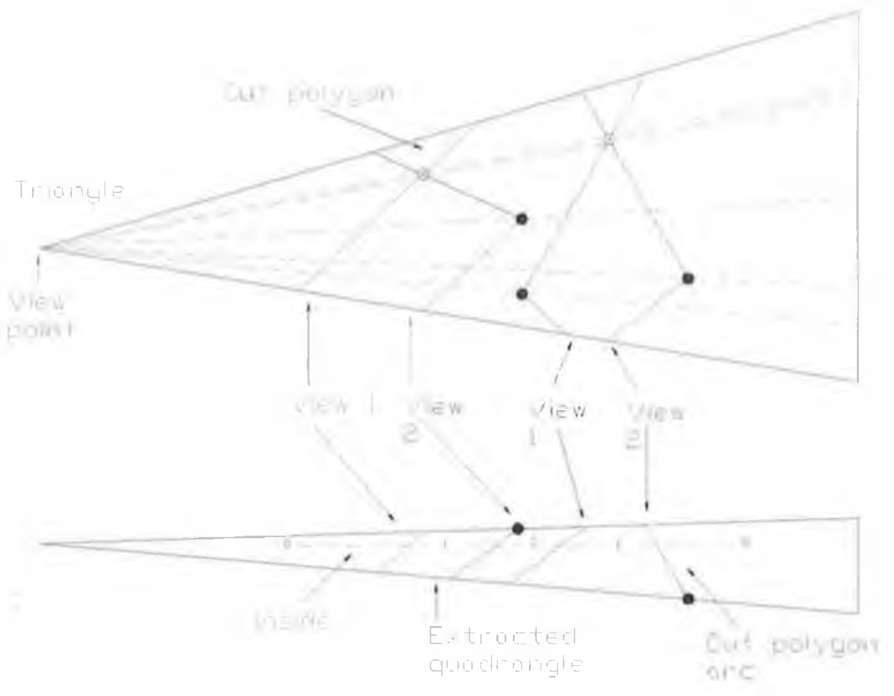


Figure 2.13: A triangle is partitioned by cut polygons. Bottom: Resulting quadrangle is extracted from a triangle slice.

Morten Bro-Nielsen proposed method still has the limitation as previous volumetric construction algorithms, such as a limited precision and the inability to detect concavities in the object.

2.3 Visual Hull based 3D construction review:

Laurentini [35] developed a general geometric tool called a visual hull for dealing in a simple and straightforward way with the questions raised for nonconvex objects by the silhouette based approach to image understanding. He proposed efficient algorithms for computing internal and external visual hull in 2D and also for computing the silhouette active surface of 3-D polygonal object, based on the 2D algorithm.

Visual Hull: Visual Hull VH(S,R) of an object S relative to a viewing region R is a region E^3 such that for each point P $\in$ VH(S,R) and each viewpoint V $\in$ R, the half line starting at V and passing through P contain at least a point of S (Figure 2.14 (a) and (b)).

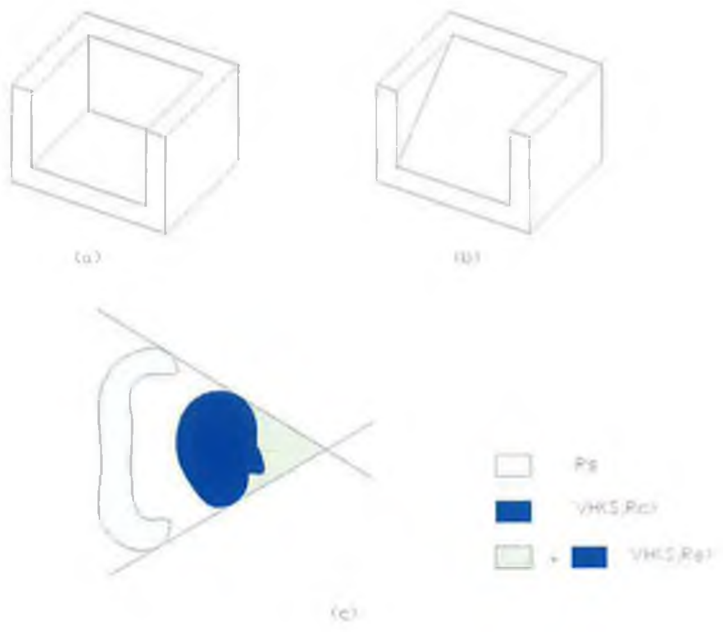


Figure 2.14: a) Silhouette of an object b) Visual Hull of the object (a) and (c) The Visual hull relative to R_s can be obtained from the visual hull relative to R_c.

External Visual Hull: VH(S, R_C) was defined as the external visual hull or simply the visual hull of S, VH(S), without any other specification. In other word, VH(S) is the largest visual hull of S whose viewing region is bounded by the visual hull itself (Figure 2.14 (c)).

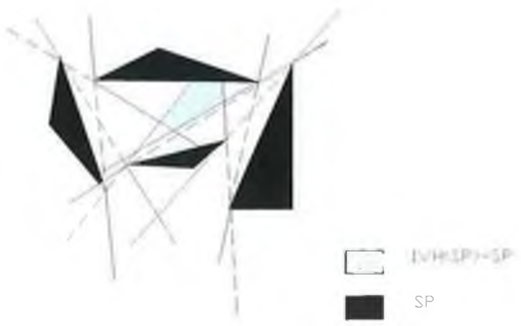


Figure 2.15: An example of internal visual hull

Internal Visual Hull: If the boundary of the viewing region is S itself, and $R_i = E^3 - S$, VH(S, R$_i$) is defined as the internal visual hull of S, IVH(S) (Figure 2.15).

He presented two efficient algorithms for computing the visual hull and the internal visual hull of 2-D objects. These algorithms divide the plane into regions that either entirely belongs to the visual hull or its complement. The time bound to both these algorithms is $O(n^3 + m \log m)$, where m is the size of the partition and n is the number of algebraic curves defining the boundary of the object. The 2-D algorithms find the visual hulls of $2\frac{1}{2} - D$ sweep solids and provide the basis for an algorithm described elsewhere for dealing with solids of revolution. He also show that for a polyhedron the external silhouette-active surface can be computed applying the 2-D algorithm in each of the planes supporting a face of S. His proposed system is only for polyhedral object not for the curved surfaces.

Laurentini [36] proposed a new technique for surface detection for 3D construction from silhouette based approach. 3D can be constructed by visual hull [35] concept or using volume intersection technique [17][26][28]. Visual Hull is relatively new concept for approximate the 3D object. It is the closest approximation of object *O* that can be constructed from its silhouettes. For surface detection of 3D object *R* which is reconstructed from original object *O*, he divided the points of the surface *R* into hard point and soft points. Hard points belong to the surface of any possible object originating *R*. Soft points may or may not belong to *O* (Figure 2.16 a, b, c). He provided theory for both hard point and soft point detection for both polyhedral surfaces and curved surfaces which were detected by Visual Hull (Figure 2.16(d)).

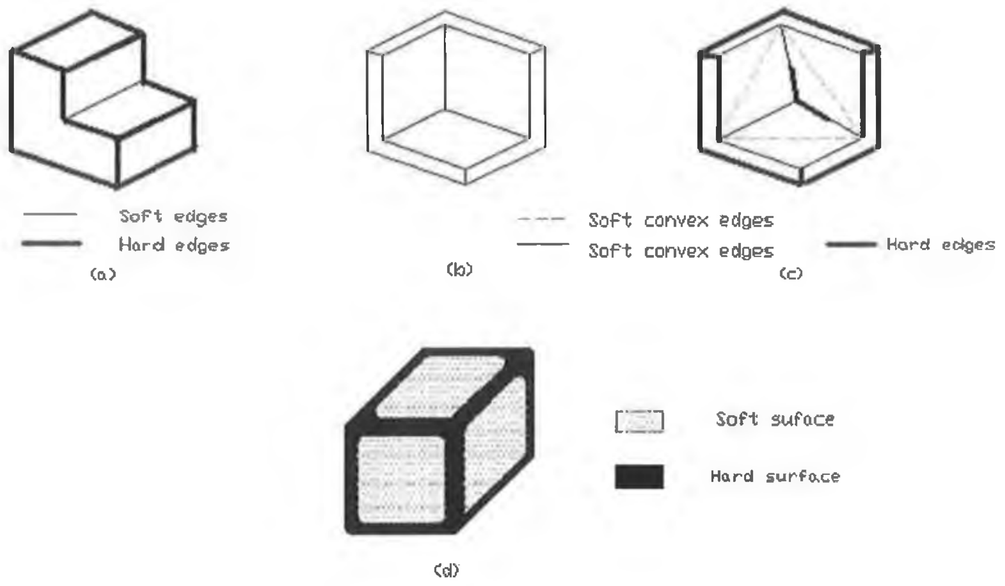


Figure 2.16: a) A polyhedral visual hull and is hard and soft edges, c) the visual hull of (b) has three soft edges and 2 partial hard convex edges, d) the hard and soft surface of disc-shaped visual hull.

Since in practice, one could be unable to obtain the visual hull of an object or to find out if the reconstructed object is visual hull or not, he addressed the problem of finding the hard points for a generic reconstructed object. Also, in this case, he had given a necessary and sufficient condition for a point to be hard.

He also shown that when, in theory, a reconstructed visual hull can also supply hard surfaces, a generic reconstructed object is able to supply only hard points or hard lines.

Chapter Three
Overview of the Developed System

3.1 Introduction:

The system developed in this research project is used to create a 3D surface model from a series of 2D images. A moderately complex object can be reconstructed using the proposed system. It consists of a turntable, a digital camera, a computer, a rail, a tripod and the software named *3D Model.* The author himself had done the turntable and the calibration rail design and the development of the *3D Model* software in Visual C++.

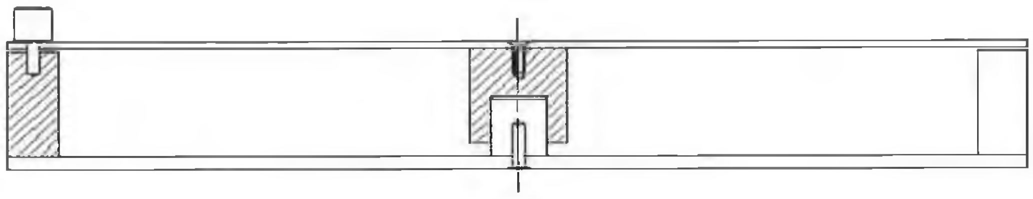
The whole system task can be divided into two distinct parts. One is camera calibration and the other is 2D to 3D conversion. Rail and calibration patterns are used in camera calibration. The turntable is used for target object image capture. A camera, tripod and software are the common elements for both calibration and 2D to 3D conversion. As will be illustrated later the background colour selection is very important for image capture.

In the first step, the calibration pattern is placed on the rail with a suitable background colour and images are taken with the camera which is fixed on the tripod. In the second step, the calibration rail is replaced with the turntable and the target object is placed in the centre of the table. Images of the target object are taken at equal intervals of a certain degree (i.e. 10 degree) of rotation of the turntable. The object and calibration pattern images are transferred in the *3D Model* software to create the 3D image and camera calibration respectively.

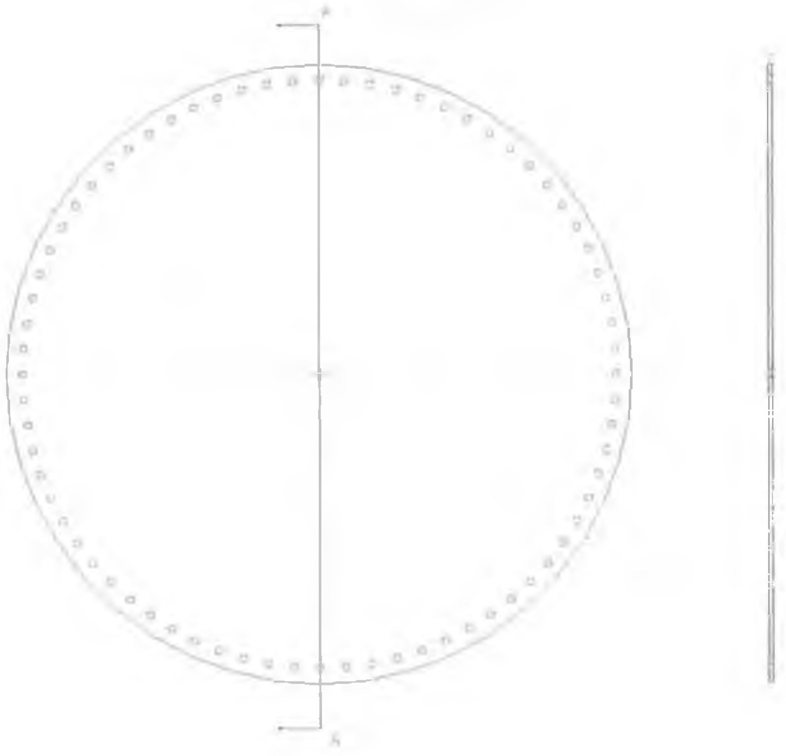
Section 3.2 briefly describes the turntable, section 3.3 deals with the camera specification, section 3.4 describes computer configuration, section 3.5 contains the rail information used in camera calibration, section 3.6 deals with the tripod and 3.7 introduces how to use the software.

3.2 Turntable:

The target object is placed on the center of the turntable. This Table has a facility to rotate the object in steps of 5 degree, 10 degree, 15 degree and so on. The table is designed in such a way that if it is required to change the rotation degree step size, it is only necessary to replace the top disc of the table with another disc that has the desired rotation degree markers / holes (Figure 3.1 a, b).



(a)



(b)

Figure 3.1: (a) Front View of the rotating table, (b) Top View of the rotating plate.

3.3 Digital Camera:

The Camera specification is very important for the *3D Model* software. It differs for coplanar and noncoplanar camera calibration. One should have the following information for noncoplanar calibration before using the camera in this project:

1. camera sensor element number.
2. sensor element center to center distance
3. image resolution

For coplanar calibration all the camera specification discussed above are needed together with the following specification.

1. camera frequency
2. A/D frequency

A Nikon coolpix digital camera was used in this project. Its specification is as follows:

1. Sensor resolution is 1600 x 1200,
2. sensor element center to center distance for both x and y directions are 0.0005 mm/sel.
3. image resolution is set to 640x480.

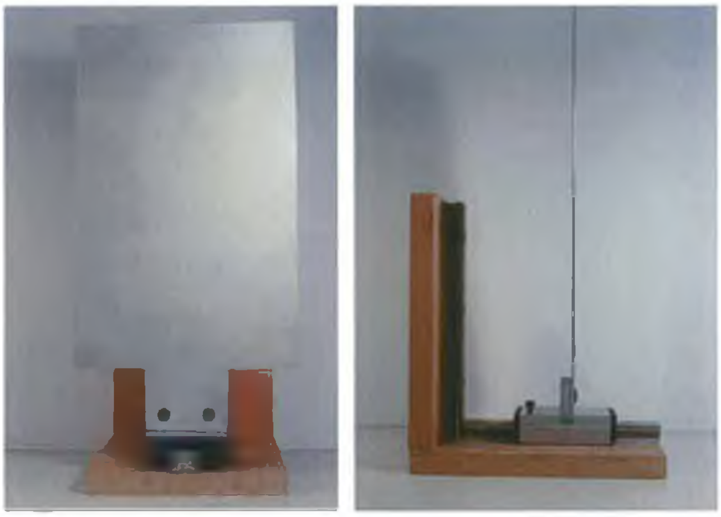
Before using the camera in the system, the camera resolution is set to the 640x640 or less than this resolution. Resolution greater than (640x640) is not supported by the developed software.

3.4 Computer configuration:

Minimum requirement is Pentium 2 450MHz processor and REM 256MB.

3.5 Rail:

It is used for calibration of the camera. Figure 3.2 a,b shows calibration rail front and side view. And Figure 3.2c shows the rail with calibration pattern.


(a) Front view (b) side view

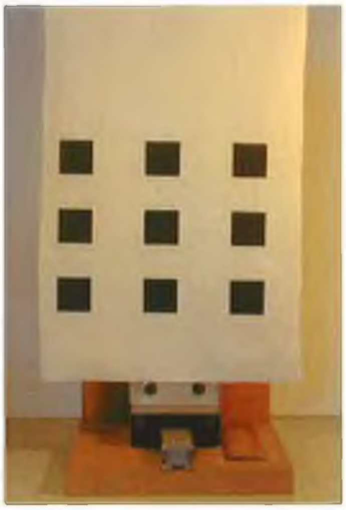

(c) rail with calibration pattern

Figure 3.2: Calibration rail and calibration pattern

Two cameras can be calibrated using the Rail. One camera being rotated in horizontal direction and other is in vertical direction. It can be used for calibrating the cameras in stereo system for future project.

3.6 Tripod:

It is used to hold the camera in a fixed position.

3.7 Software:

When the 3D Model software starts to execute it provides a single document graphical user interface (Figure 3.3). This software mainly focuses on 2D to 3D conversion and Camera Calibration. It provides 2D and 3D menus to deal with 2D to 3D conversion and a Calibration menu for the camera calibration.

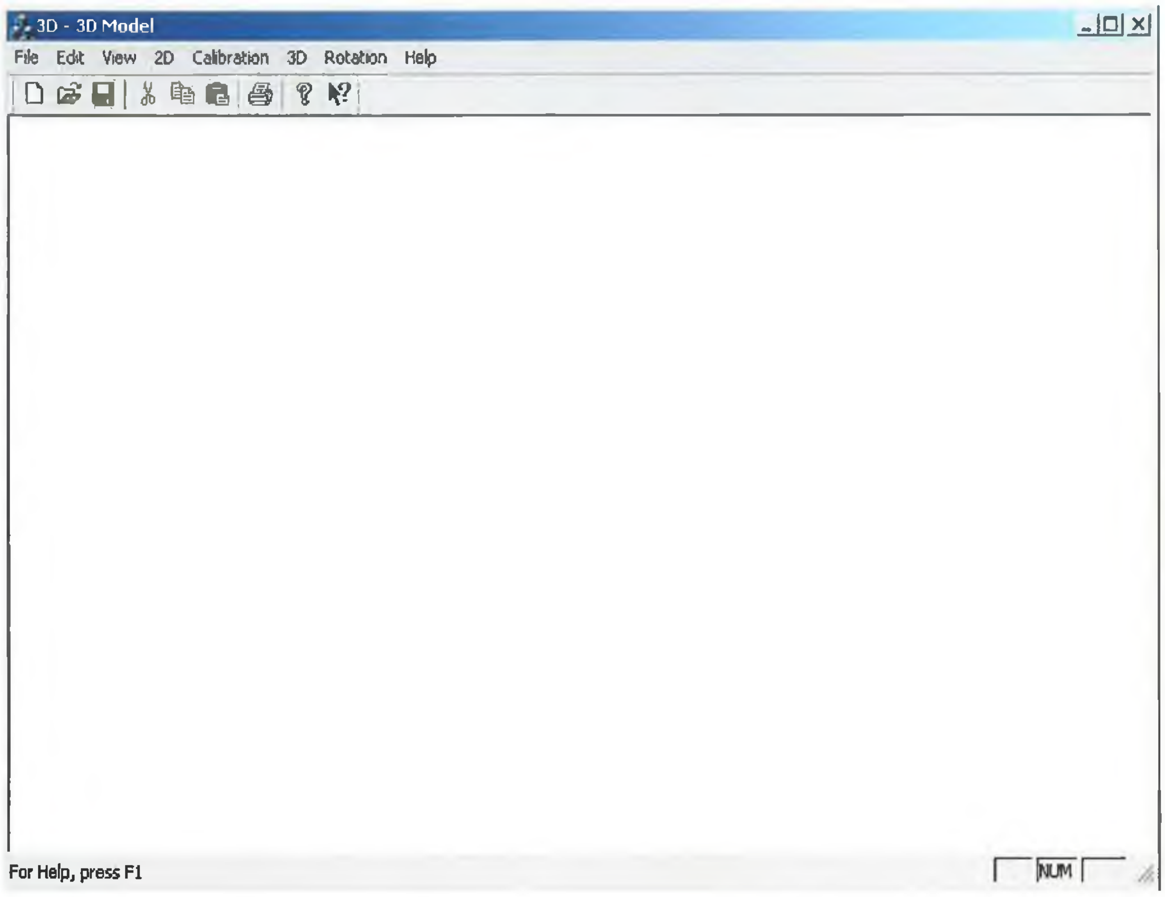


Figure 3.3: Main Window

3.7.1 Calibration:

For the camera calibration, the required information is camera parameter, world coordinate data, corner of the calibration pattern image and calibration methods. Calibration can be done in two ways. By selecting existing calibration data from a file or by doing the following steps:

a) Camera Parameter selection

b) World Coordinate Data selection

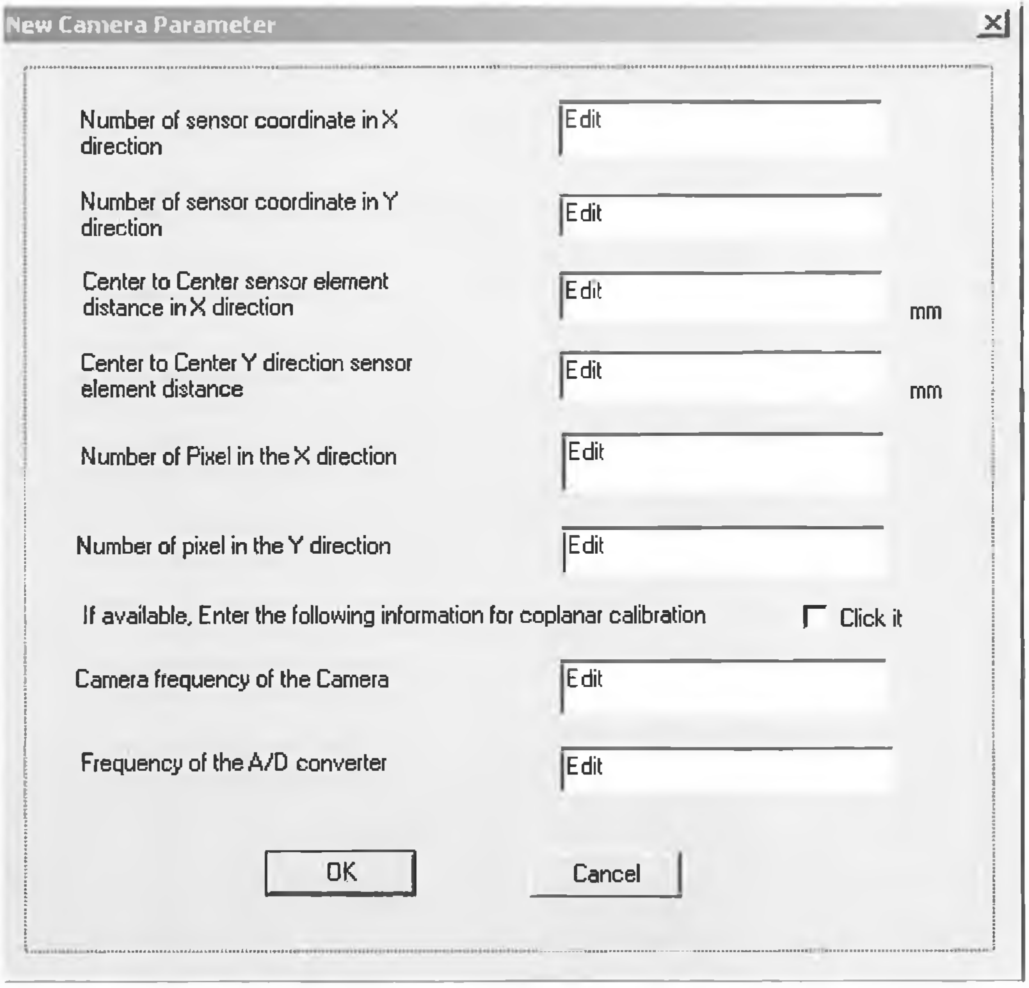
c) Corner Detection

d) Calibration method selection

e) Save Calibration

a) Camera Parameter Selection:

The camera parameter can be obtained in two ways. Firstly, using the new calibration parameter command from the Calibration menu. Secondly, selecting the camera name from the existing calibration parameter menu. The first method will provide a dialog box (figure 3.4). For noncoplanar calibration, the number of sensor elements in x and y direction, the centre to centre sensor element distances in x and y directions, and the image pixel numbers in x and y directions are required.



New Camera Parameter

Number of sensor coordinate in X direction	Edit	
Number of sensor coordinate in Y direction	Edit	
Center to Center sensor element distance in X direction	Edit	mm
Center to Center Y direction sensor element distance	Edit	mm
Number of Pixel in the X direction	Edit	
Number of pixel in the Y direction	Edit	

If available, Enter the following information for coplanar calibration ☐ Click it

Camera frequency of the Camera	Edit
Frequency of the A/D converter	Edit

Figure 3.4 New Camera Parameter Dialog box

Coplanar calibration requires all the information such as noncoplanar calibration and also camera frequency and A/D converter frequency.

The camera parameter information which are provided by the software under Existing parameter menu are:

1) Photometrics Star I
2) General Imaging MOS5300 + Matrox
3) Panasonic GP-MF702 + Matrox
4) Sony XC75 + Matrox
5) Sony XC77 + Matrox
6) Sony XC57 + Androx
7) Kodak Es310
8) Nikon CoolPix 800
9) Canon Xap Shot

b) World Coordinate Data:

The camera calibration pattern should look like that shown in Figure 3.4. Since the calibration points are on a common plane, the world coordinate system can be chosen such that z = 0, and the origin is not close to the centre of the view or y axis of the camera coordinate system. Since the (x,y,z) is user defined, and the origin is arbitrary, there is no problem setting the origin of (x,y,z) to be out of the field of view and not close to the y axis.

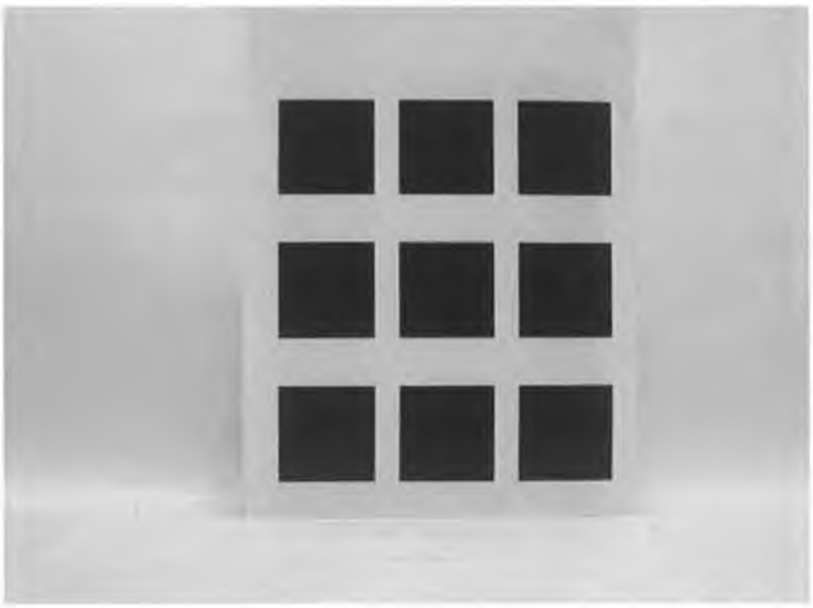
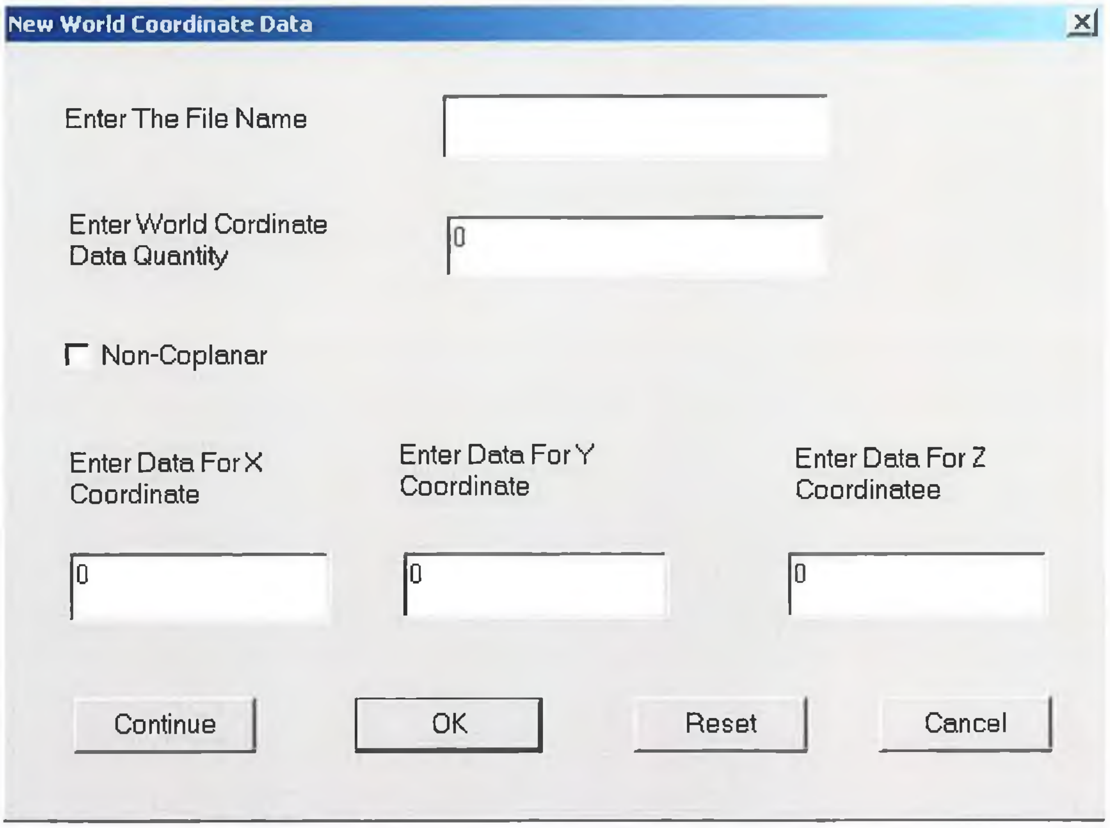


Figure 3.5: Calibration Pattern

Example: calibration pattern in Figure 3.5 has 36 calibration points. All the square boxes are of 200 mm width and distance between the boxes in the horizontal direction is 100 mm and in the vertical direction is 150 mm. Let the world coordinate be at the 100mm left bottom corner of the page. Let z=0, the four corner points of the left bottom square will be (10,10,0), (10,210,0), (210,210,0) and (210,10,0).

The world coordinate data can be inserted by using "New WC Data" command or "Existing WC Data" command from the calibration menu. "New WC Data" command will show the dialog box presented in Figure 3.6. The first edit box is for the file name in which the world coordinate data will be saved. File name extension should be ".txt". The second edit box is for total world coordinate data number, which should be the total corner point in the calibration pattern. Figure 3.5 requires 36 World Coordinate data points for coplanar calibration and for noncoplanar calibration of 3 images requires 36x3= 108 corner points. For non-coplanar calibration check non-coplanar check box. After entering three x, y z values in three edit box, the continue button should be press.


New World Coordinate Data

Enter The File Name

Enter World Cordinate Data Quantity: 0

☐ Non-Coplanar

Enter Data For X Coordinate: 0

Enter Data For Y Coordinate: 0

Enter Data For Z Coordinateе: 0

Figure 3.6: New World coordinate data dialog box.

"Existing WC Data" command for the calibration shows the open dialog box. It is required to set the type of Files in the open dialog box to "*.txt" as shown in Figure 3.7, because all the World Coordinate data file are saved in text file with extension "*.txt".

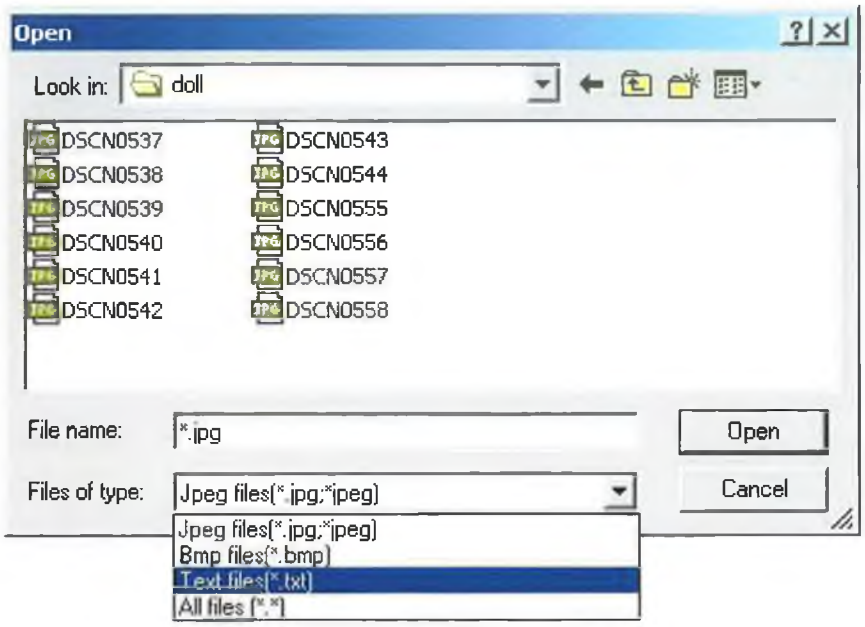


Figure 3.7: Open Dialog box

c) Corner Detection:

Corner detection from the calibration pattern image is not same for coplanar or noncoplanar calibration. So 'noncoplanar' and 'coplanar' are the two commands under Corner Detection (Figure 3.8) sub menu.

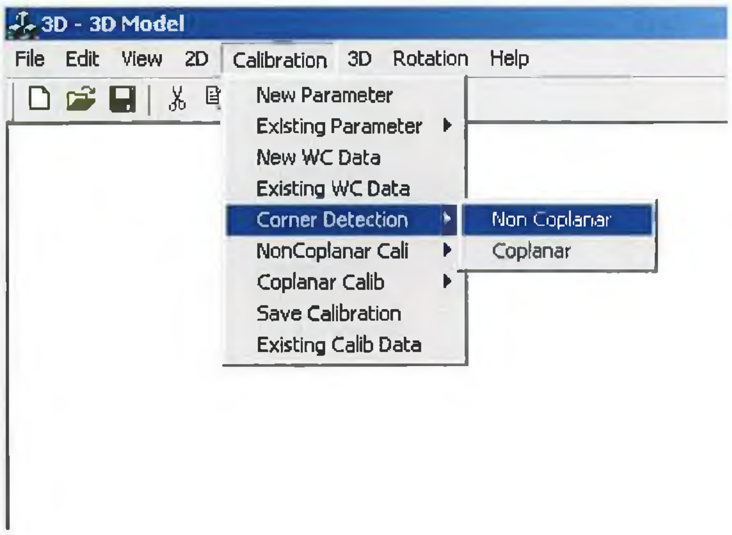
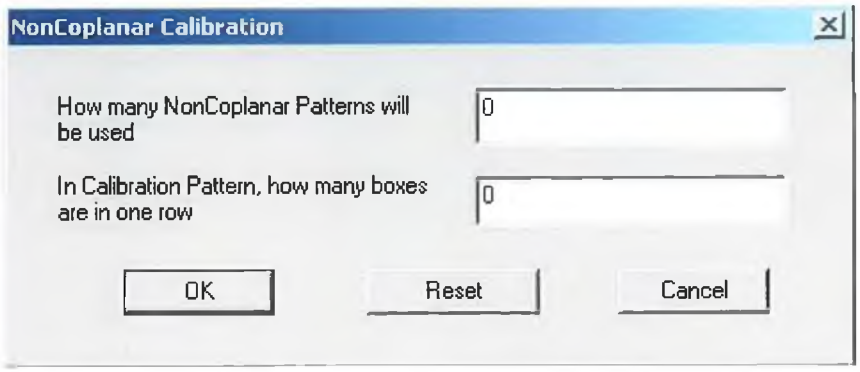


Figure 3.8: Corner Detection menu

c.1) Non Coplanar Corner Detection:

Selection of the ‘noncoplanar’ shows a dialog box (Figure 3.9) for quantity of calibration image and box number in one row of the calibration image. If in an experiment three images such as shown in Figure 3.5 are taken by moving the calibration pattern, the number of calibration patterns will be three and the box number will be three if the pattern looks like that in Figure 3.5.


NonCoplanar Calibration

How many NonCoplanar Patterns will be used: 0

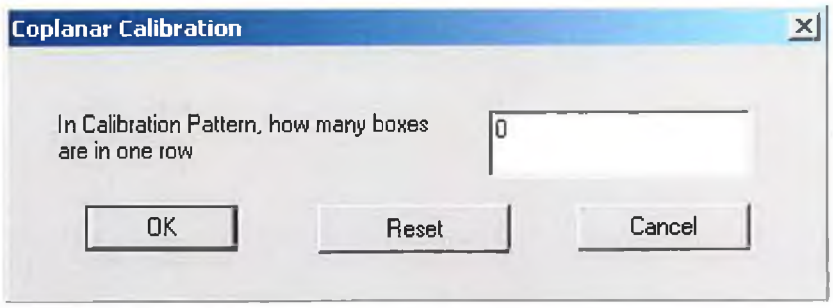
In Calibration Pattern, how many boxes are in one row: 0

Figure 3.9: Non Coplanar Corner Detection Dialog box.

The Ok button will close the window and will call the open dialog box (Figure 3.7) for loading calibration pattern image. If the calibration pattern numbers three the open dialog box will be called three times and will detect the corner point for each image (Figure 3.11).

c.2) Coplanar Corner Detection:

Selection of the ‘Coplanar’ from the menu shows a dialog box (Figure 3.10) for box number in one row of the calibration image. The Ok button will close the dialog box and will call the open dialog box once for loading the calibration image. Then the corner of the calibration image will be detected (Figure 3.11).


Coplanar Calibration

In Calibration Pattern, how many boxes are in one row: 0

Figure 3.10: Coplanar Corner Detection Dialog box

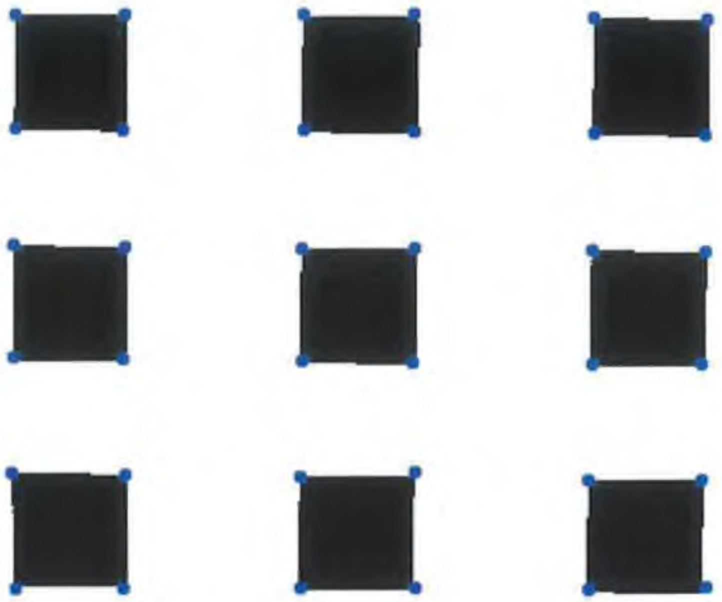


Figure 3.11: Corner point of the Calibration image of Figure 3.5

d) Calibration Method Selection:

Non coplanar calibration and Coplanar calibration can be done with non linear optimization or without non linear optimization (Figure 3.12). The "With Optimization" command in Non Coplanar calibration will run the versatile camera calibration technique with Non Linear optimization suggested by Tasi [3]. "Without Optimization" command will run the camera calibration without non linear optimization.

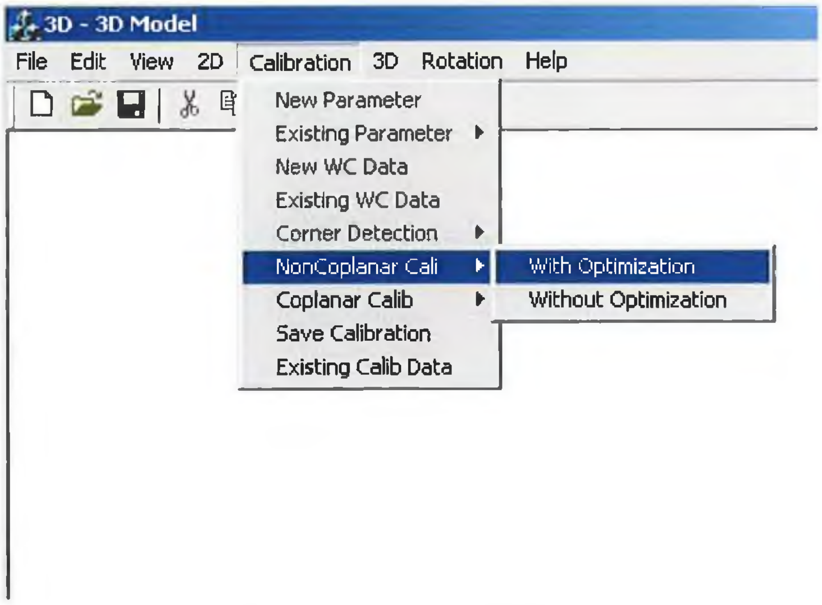


Figure 3.12: Non Coplanar Calibration menu

Coplanar calibration has also “With Optimization” and “Without Optimization” options as like Non Coplanar calibration

e) Save Calibration:

Save Calibration command from Calibration menu will call the save dialog box to save the calibration information in a text file.

f) Existing Calibration:

“Existing Calib data” command from the Calibration menu will call the open dialog box to open a text file with calibration information. This command will set the Camera class object with calibration data.

3.7.2 2D to 3D conversion:

2D and 3D menus are dedicated for converting 2D images to 3D. Background color selection, Quadtree creation, Silhouette extraction and delete option for both quadtree and original image are main features of the 2D menu. Octree creation and Texture mapping are the content of the 3D menu. Rotate menu is used for rotating the 3D object. Steps for creating 3D from to 2D images are:

i. Background selection.
ii. Silhouette Extraction.
iii. Quadtree Creation.
iv. Octree creation.
v. Texture Map.

i) Back Ground Selection:

The “Background” command in the 2D menu (Figure 3.13) offers three sub options. “Image Selection” command finds the lowest colour and highest colour value from the image in the client window. Before using this command appropriate background image should be loaded in the window using Open command from File menu. The

“One Area Selection” command can be used to select a rectangle area from an image as a background. This command will find the lowest and highest colour values from the selected area. “Two Area Selection” command is used to find the lowest colour and highest colour value from two selected rectangle areas. In this command, Right mouse button should be clicked four times to select two rectangle areas. First two clicks will be used as the two corner points of the first rectangle and third and fourth click will be used as the two corner points of the second area.

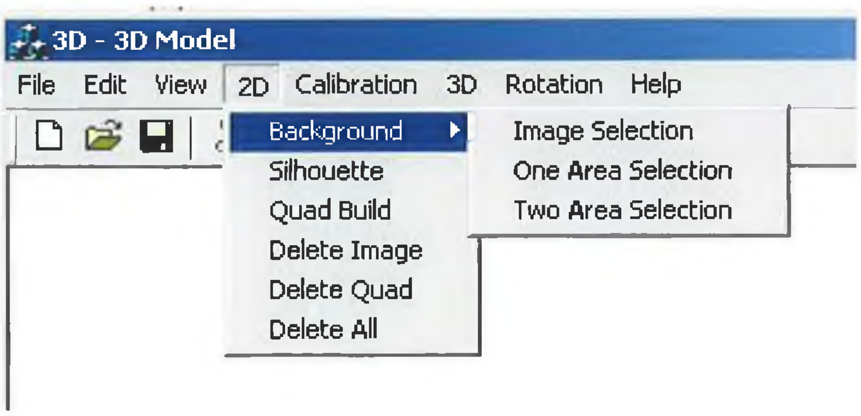
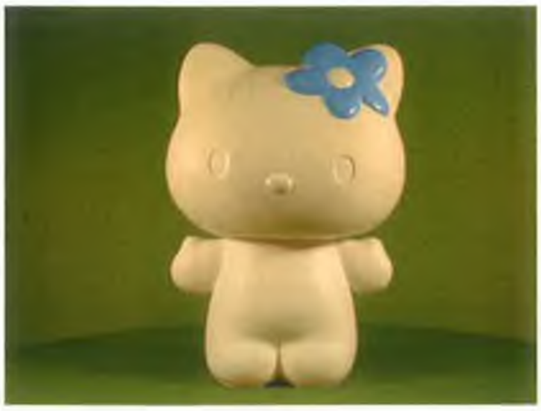
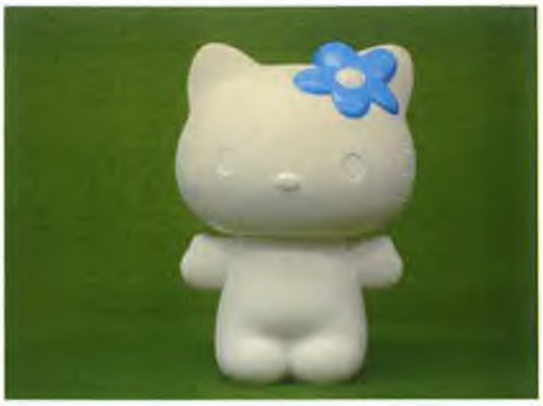


Figure 3.13: 2D Menu

With the same lighting Figure 3.14a shows the object image with green background colour and Figure 3.14b shows the green background image. Similarly, Figure 3.14c,d, 3.15 and 3.16 shows the object image and background image with different lighting and different background colour. It is clear from the Figure 3.14-3.16 that with the same lighting digital camera (Nikon Coolpix-800) images show different colours for the same background when the object is not in the image. Same effect has been found for Kodack DC-3200 and Samsung cameras. So, it can be concluded that all the commercially available digital camera used for normal photograph shows the same effect. To solve this problem “One Area Selection” and “Two Area Selection” (Figure 3.13) has been provided in the software.


(a) Object image with green background

(b) background image


(c) Object image with green background

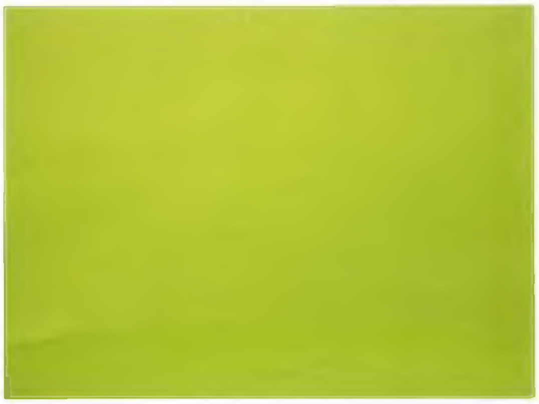
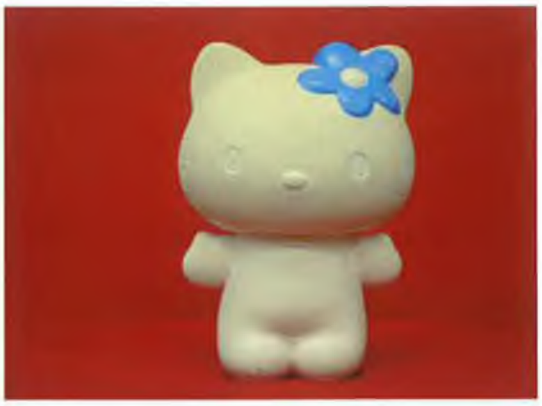

(d) background image

Figure 3.14: Object images and background images with green background in different lighting.

(a) Object image with red background

(b) Background image

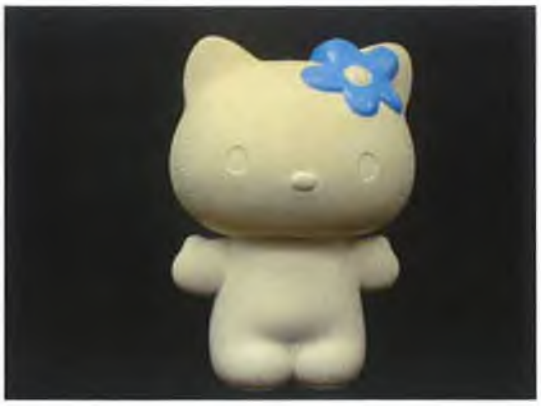

(c) Object image with red background

(d) Background image

Figure 3.15: Object images and background images with red background in different lighting.

(a) Object image with blue background

(b) background image


(c) Object image with blue background

(d) background image

Figure 3.16: Object images and background images with blue background in different lighting.

ii) Silhouette Extraction:

"Silhouette" command in 2D menu creates silhouette image from the object image (Figure 3.17), using the background color values. Target image should be loaded in the window using the Open command from the File menu before using this command.

Figure 3.17: Silhouette image

If 36 images are taken with 10 degrees interval for the 3D image creation, the silhouette extraction should be done 36 times.

iii) Quadtree Creation:

"Quad Build" command (Figure 3.12) from 2D menu creates quadtree from the silhouette images. This command should be used after the silhouette has been already created and displayed in the window. No of "Quad Build" command will be as like "Silhouette" command.

iv) Octree From Quadtree:

"Display Octree" command (Figure 3.18) will create the 3D octree from the 2D quadtrees. This command will display a dialog box (Figure 3.19) for rotation angle of the target object. This angle will be negative if the target object is rotated in anti clockwise direction on the turntable.

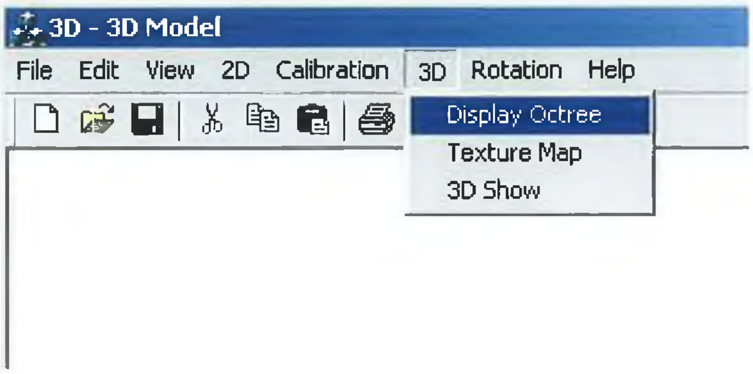
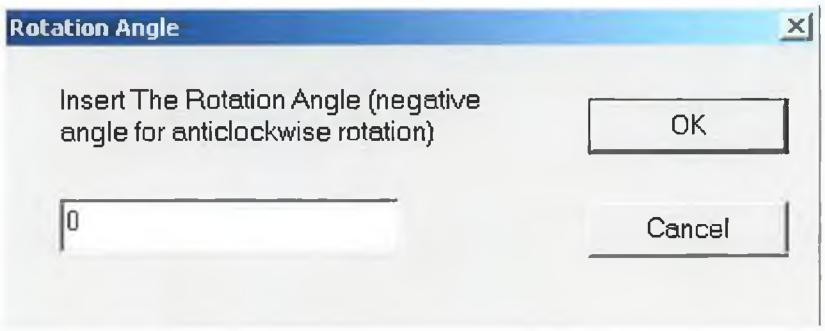


Figure 3.18: 3D menu

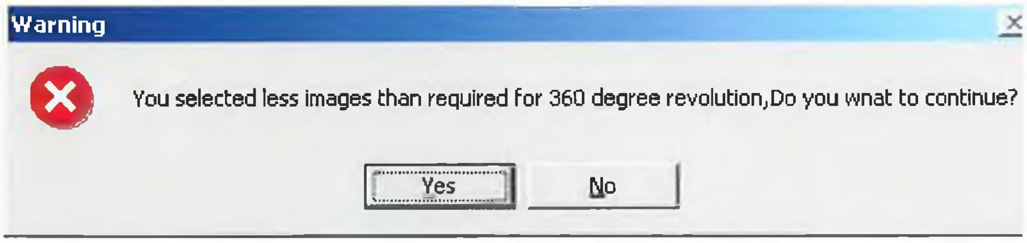


Insert The Rotation Angle (negative angle for anticlockwise rotation)

0

Figure 3.19: Rotation Angle Dialog box

If 10 degree is the rotation angle and created quadtree number is 10, "Display Octree" command will show a warning message (Figure 3.20). Selection "yes" will create partial octree from the 10 quatree.



You selected less images than required for 360 degree revolution,Do you wnat to continue?

Figure 3.20: Warning message for Octree creation.

v) Texture Map:

“Texture Map” command (Figure 3.18) creates the surface of a 3D image and display the surface image (Figure 3.21) in the window. Figure 3.18 shows the texture map of a cup object for two images.

3.21: Surface for two images of a cup

3.7.3 3D Image Display:

“3D Show” command (Figure 3.18) shows the 3D image with octree and surface point (Figure 3.22).

3.22: Octree with Surface for two images

3.7.4 Rotation:

Rotation menu (Figure 3.23) offers clockwise rotation or anti clockwise rotation command. “Clock Wise” or “Anti Clock” command offers 10 degree, 20 degree or custom rotation of the 3D image in clockwise or anti clockwise direction respectively.

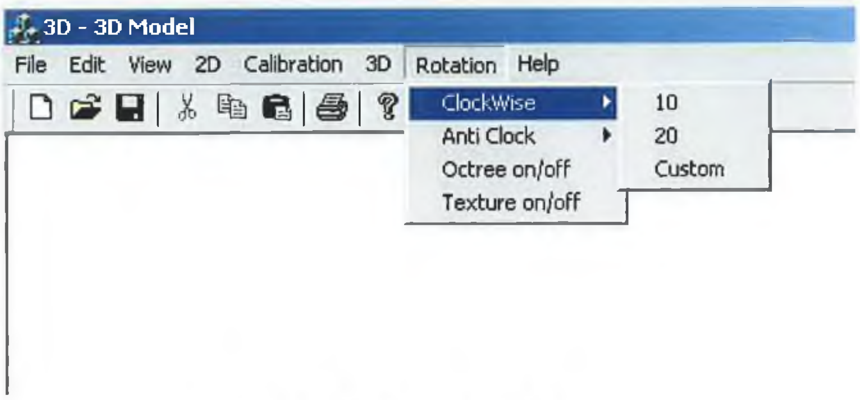


Figure 3.23: Rotation menu

Custom command under "ClockWise" or "Aniti ClockWise" submenu shows a dialog box (Figure 3.24) for a rotation angle to rotate the 3D image.

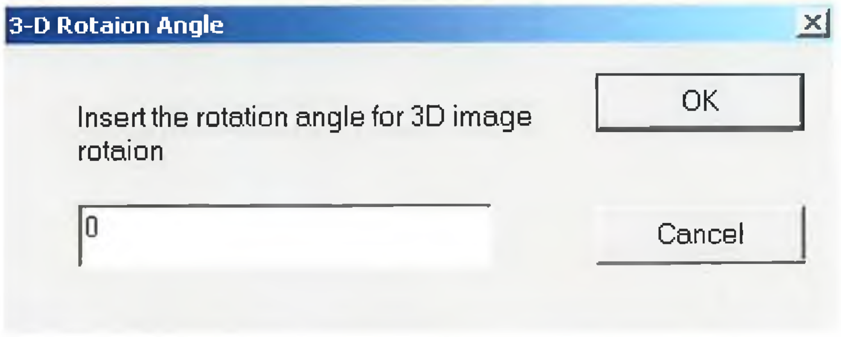


Figure 3.24: Rotation angle Dialog Box for 3D object rotation

3.7.5 Octree on / off:

Rotation menu also offers "Octree on/off" and "Txeture on/off" commands for displaying the 3D image with or without octree and Texture image.

3.7.6: Saving 3D Image:

"Save" command from the File menu is used for 3D Image saving. It shows the Save dialog box (Figure 3.25). And offers two types of file format, one is DXF format and other is "Binary Text File" format. File saved in DXF format can be open by the Auto CAD. And file saved in Text file format can only be used by this software.

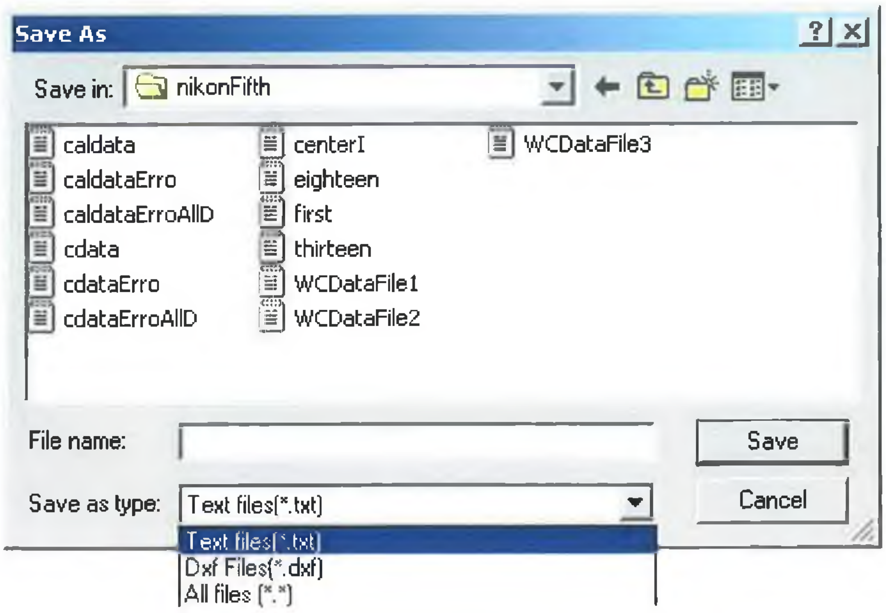


Figure 3.25: Save Dialog Box

This software can save 3D images in DXF format but cannot open the DXF file.

3.7.7: Help Menu:

Help menu (Figure 3.26) offers “Help Topics” and “About 3D Model” commands.

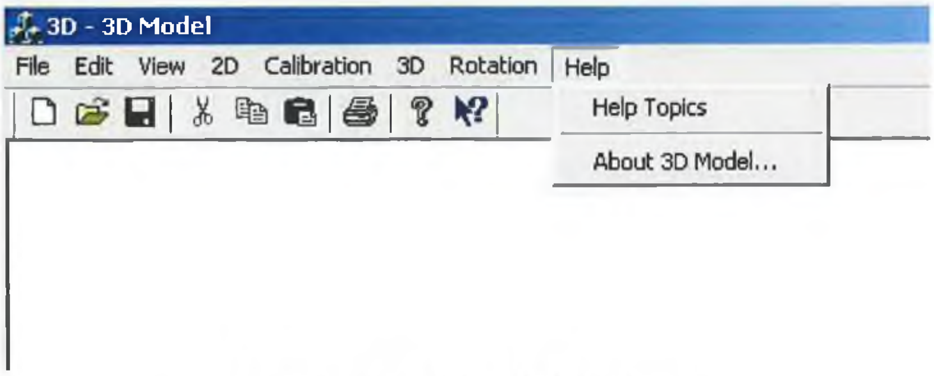
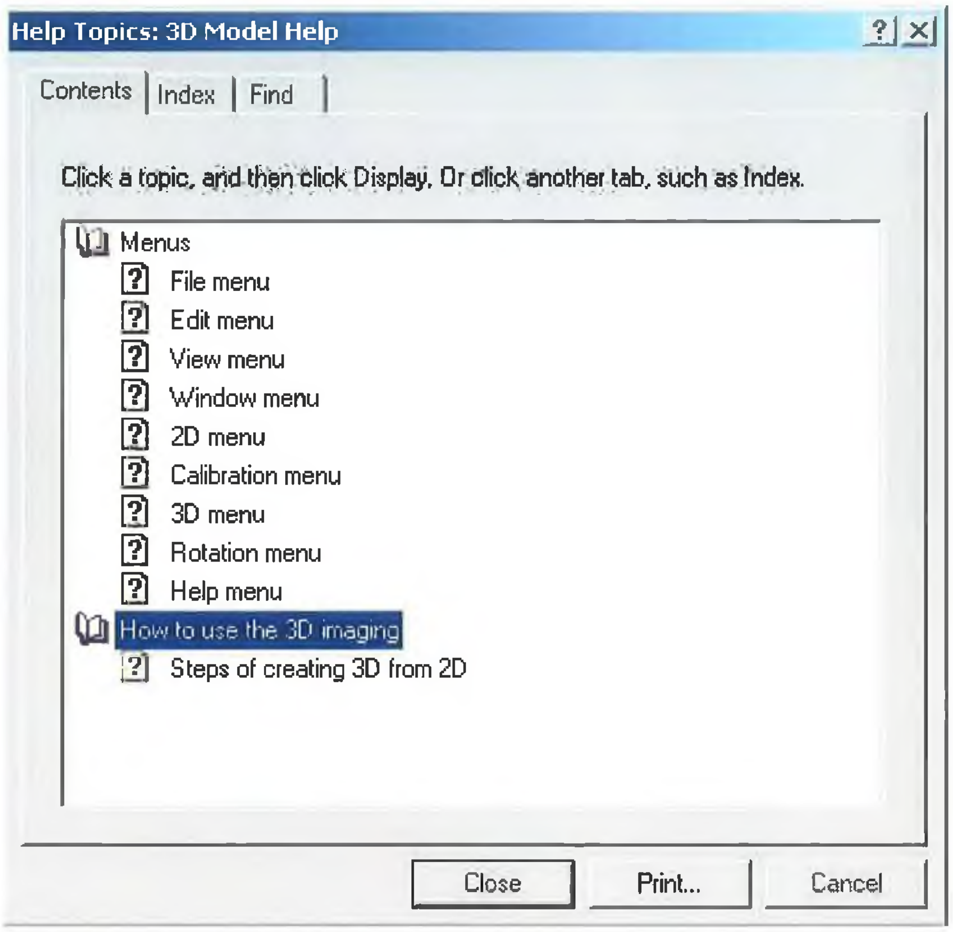


Figure 3.26: 3D Model Help Menu

Help Topics command from the Help menu shows the help window as like the Figure 3.27.


Help Topics: 3D Model Help

Contents | Index | Find

Click a topic, and then click Display. Or click another tab, such as Index.

- Menus
  - File menu
  - Edit menu
  - View menu
  - Window menu
  - 2D menu
  - Calibration menu
  - 3D menu
  - Rotation menu
  - Help menu
- How to use the 3D imaging
  - Steps of creating 3D from 2D

Figure 3.27: Help Window

It provides all the necessary information for using the software for creating 3D from 2D images.

Chapter Four
Algorithm

4.1 Introduction:

3-Dimensional images can be created from 2D images by the silhouette based approach, the contour based approach or by the visual hull based approach. To evaluate the complexity and for higher efficiency the silhouette-based approach has been used in this project. The main steps for creating 3D images are as follows:

1. Image acquisition
2. Camera calibration
3. Silhouette extraction
4. Shape modeling
5. Texture mapping

System flow chart can be described as follows:

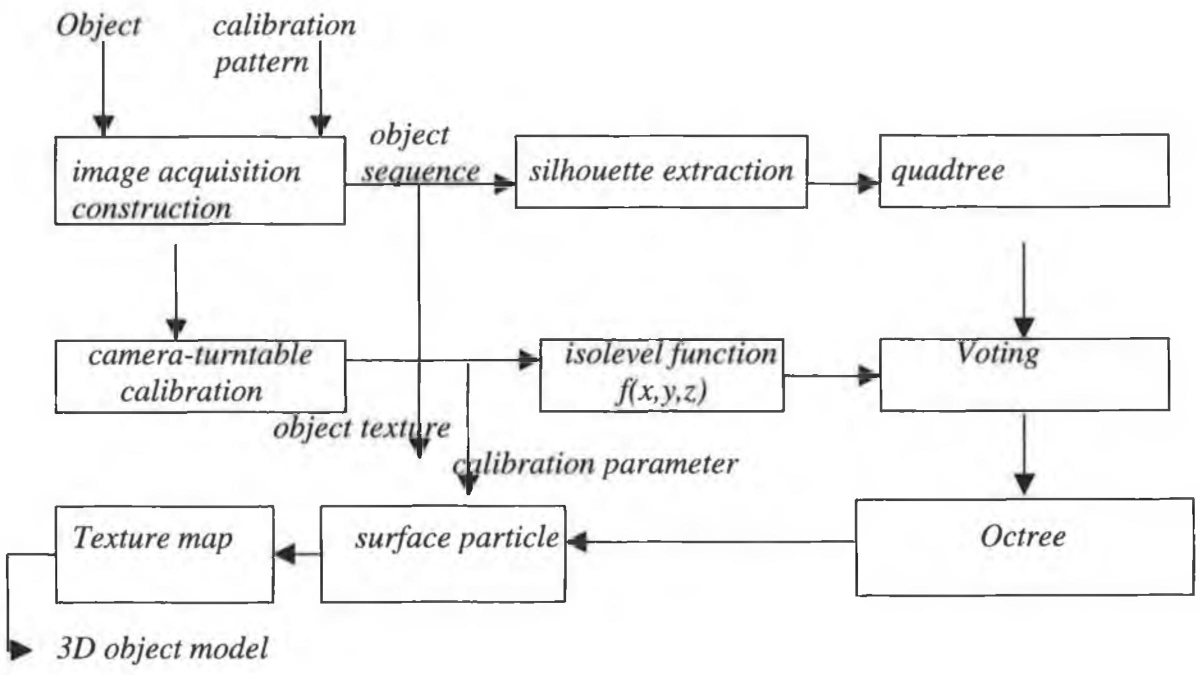


Figure 4.1 Algorithm Flow Chart

Section 4.2 reviews the image acquisition process, section 4.3 deals with camera calibration, section 4.4 briefly describes the silhouette extraction, section 4.5 contains the shape modeling procedure, and section 4.6 explains the texture map.

4.2 Image Acquisition Process:

Chapter three described all the component of the image acquisition system. The basic components are the digital camera, the turn table, tripod and background colour. The background colour selection is very important in image acquisition process. Colour selection should be such way that the objects colour does not match with background colour. Otherwise silhouette extraction will be incorrect.

In the first step of the image acquisition process, the background image is taken without the object. In the second step, images of the object are taken by rotating the object at interval of specific angle (i.e. 10 degree). In both steps the camera position was fixed.

The Background class is responsible for finding the highest and lowest colour value in the background image.

```
class background
{
        unsigned long backvalue;    //holds the lowest colour value
        unsigned long frontvalue;   //holds the heist colour value
public:
        background(int width,int height,int startx,int starty,CDC* pDC);
        inline unsigned long ReturnBackvalue(){ return backvalue;};
        inline unsigned long ReturnFrontvalue(){ return frontvalue;};
};  //End of the background class definition
```

The implementation of this class store the lowest colour and highest colour of the background image in the *backvalue* and *frontvalue* unsigned long parameter respectively.

The *background* method finds the lowest colour value and highest colour value of the image.
The *ReturnBackvalue* returns the lowest colour value.
The *ReturnFrontvalue* returns the highest colour value.

4.3 Camera Calibration:

The camera calibration is the process used for determining the internal camera geometry and optical characteristics (intrinsic parameters) and / or the 3-D position of the camera frame relative to a certain world coordinate system (extrinsic parameter).
The camera calibration is used for the following purposes:
1. Inferring 3D information from computer image coordinate:
There are two kinds of 3D information to be inferred. They are different mainly because of the difference in application.

a) 3D information concerning the location of the object, target or feature.
For simplicity, if the object is a point feature camera calibration provides a way of determining a ray in 3D space that the object point must lie on, give the computer image coordinates. With two views either taken from two cameras or one camera in two locations, the position of the object point can be determined by intersecting the two rays. Both intrinsic and extrinsic parameters need to be calibrated.

b) 3D information concerning the position and orientation of the moving camera relative to the target world coordinate system.

2. Inferring 2D computer image coordinates from 3D information:
In model-driven inspection or assembly applications using machine vision, a hypothesis of the state of the world can be verified or confirmed by observing if the image coordinates of the object conform to the hypothesis.

In this project, the purpose of the calibration was to establish the relationship between 3-D world coordinates and their corresponding 2-D image coordinates as seen by the computer. Once this relationship is established, 3-D information can be inferred from 2-D information and vice versa.

Existing techniques for camera calibration can be classified into the following categories^2.

1) *Direct Nonlinear Minimization:*

In this category, equations that relate the parameters to be established with the 3-D coordinates of control points and their image plane projections are established. The search for the parameters involves using an iterative algorithm with the objective of minimizing residual errors equations. Most of the classical calibration techniques in photogrammetry belong to this category(e.g. [37-41]).

<u>Advantages</u>:

a) The camera model can be very general to cover many types of distortion. Some simple distortion-free models for computer vision applications have also been employed this type of technique [42,43].

b) The algorithm may achieve high accuracy, provided that the estimation model is good, correct convergence has been reached.

<u>Disadvantages</u>:

a) Since the algorithm is iterative, the procedure may end up with a bad solution unless a good initial guess is available.

b) Once the distortion parameters are included in the parameter space, the minimization may be unstable if the procedure of iterations is not properly designed. The iteration between the distortion parameters and external parameters can lead to divergence or false solutions.

2) *Closed form solution:*

With this type of scheme, parameter values are computed directly through a noniterative algorithm based on a closed-form solution [38], [41], [44], [45]. A set of intermediate parameters is defined in terms of the original parameters. The intermediate parameters can be computed by solving linear equations, and the final parameters are determined from those intermediate parameters.

<u>Advantages</u>:

Since no nonlinear optimisation is required, the algorithm is fast.

Disadvantages:

a) Camera distortion cannot be corrected, and therefore, distortion effects cannot be corrected. But the direct linear transformation (DLT) introduced by Abdel-Aziz and Karana [38] has been extended to incorporate distortion parameters. However, the corresponding formulation is not exact, depth components of control points, in a camera-centered coordinate systems, are assumed to be constant.

b) Due to the objective to construct a noniterative algorithm, the actual constraints in the intermediate parameters are not considered. Consequently, in the presence of noise, the intermediate solution does not satisfy the constraints, and the accuracy of the final solution is relatively poor.

3) *Two-Step Methods:*

This method involves a direct solution for most of the calibration parameters and some iterative solution for the other parameters. The existing techniques include those presented by Tasi [12] and Lenz and Tasi [46]. A radial alignment constraint is used to derive a closed-form solution for the external parameters and the effective focal length of the camera. Then, an iterative scheme is used to estimate three parameters: the depth component in the translation vector (external parameter), the effective focal length, and a radial distortion coefficient. In Tasi[46] additional parameter like image coordinate of the principle point have been included.

Advantages:

a) A closed-form solution is derived for most of the parameters.
b) The closed-form solution is immune to lens radial distortion.
c) The number of parameters to be estimated through iterations is relatively small.

Disadvantages:

a) Their method can only handle radial distortion and cannot be extended to other types of distortion.
b) The solution is not optimal because the information provided by the calibration points has not been fully utilized.

Despite few disadvantages Two-step Methods is used in this project. Section 4.2.1 discussed basic geometry for rigid transformation and section 4.2.2 discussed Two-step method elaborately.

4.3.1 Coordinate System changes and rigid Transformation:

When several different coordinate systems are considered at the same time, it is convenient to follow Craig [47] and denote by ${}^{F}P$ the coordinate vector of the point P in the frame (F).

$${}^{F}P = {}^{F}\overrightarrow{OP} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} \Leftrightarrow \overrightarrow{OP} = xi + yj + zk$$

......................(4.1)

Let us consider two coordinate systems $(A) = (O_A, I_A, j_A, k_A)$ and $(B) = (O_B, I_B, j_B, k_B)$. Let us consider that the basis vectors of both the coordinate system are parallel to each other, i.e. $i_A = i_B$, $j_B = j_A$ and $k_A = k_B$, but the origins O_A and O_B are distinct. We may say that the coordinate systems are separated by a *pure translation*, and we have $O_B P = {}^{A}P + {}^{B}O_A$, thus

$${}^{B}P = {}^{A}P + {}^{B}O_A$$

.........................(4.2)

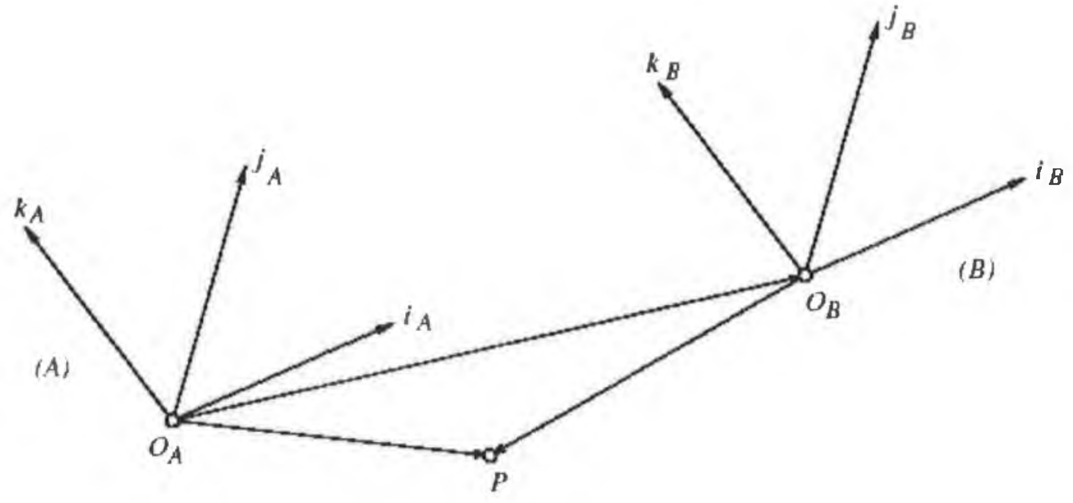


Figure 4.2 Pure Translation

When the origins of the two frames coincides, $O_A = O_B = O$, we say that the frames are separated by a *pure rotation* (Figure 4.3). Let us define the *rotation matrix R* as the 3x3 array numbers.

$${}^{B}_{A}\mathcal{R} \overset{\text{def}}{=} \begin{pmatrix} \boldsymbol{i}_A \cdot \boldsymbol{i}_B & \boldsymbol{j}_A \cdot \boldsymbol{i}_B & \boldsymbol{k}_A \cdot \boldsymbol{i}_B \\ \boldsymbol{i}_A \cdot \boldsymbol{j}_B & \boldsymbol{j}_A \cdot \boldsymbol{j}_B & \boldsymbol{k}_A \cdot \boldsymbol{j}_B \\ \boldsymbol{i}_A \cdot \boldsymbol{k}_B & \boldsymbol{j}_A \cdot \boldsymbol{k}_B & \boldsymbol{k}_A \cdot \boldsymbol{k}_B \end{pmatrix}.$$

................(4.3)

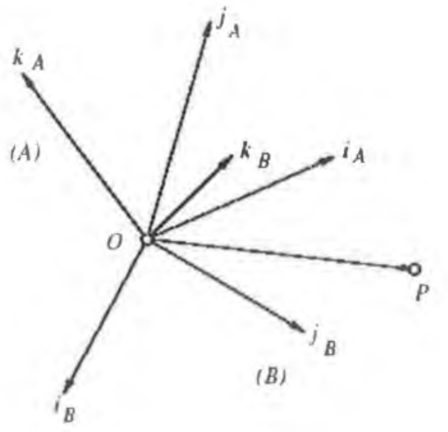


Figure 4.3 Pure Rotaion

Let us give an example of pure rotation: suppose that $k_A=k_B=k$, and denote by θ the angle such that the vector i_B is obtained by applying to the vector i_A a counter clockwise rotation of angle θ about k (Figure 4.4). The angle between the vectors j_A and j_B is also θ in this case, and we have

$$ {}^B_A\mathcal{R} = \begin{pmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}. \qquad \text{................(4.4)} $$

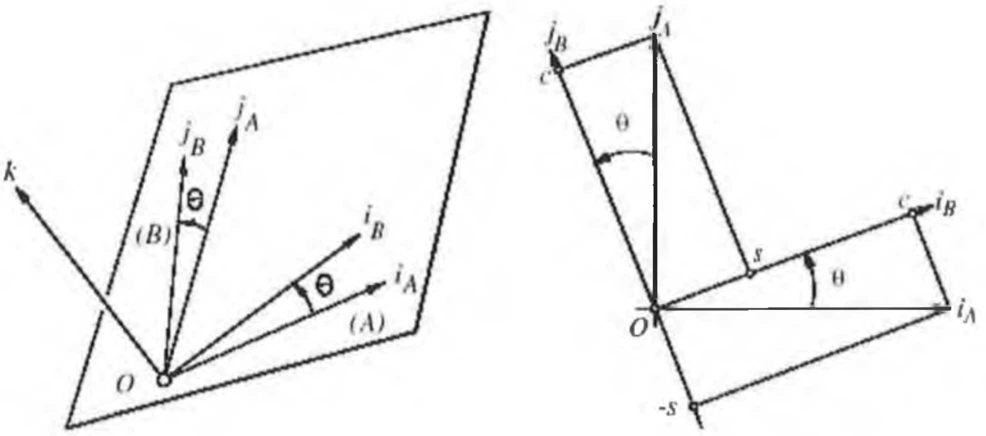


Figure 4.4 Counter clock wise rotation of angle θ

The coordinate frame separated by a rotation angle θ about their common k basis vector. When the origins of the basis vectors of the two coordinate systems are different, we say that the frames are separated by a general *rigid transformation* (Figure 4.5), and we have

$$ {}^BP = {}^B_A\mathcal{R}\,{}^AP + {}^BO_A. \qquad \text{.................(4.5)} $$

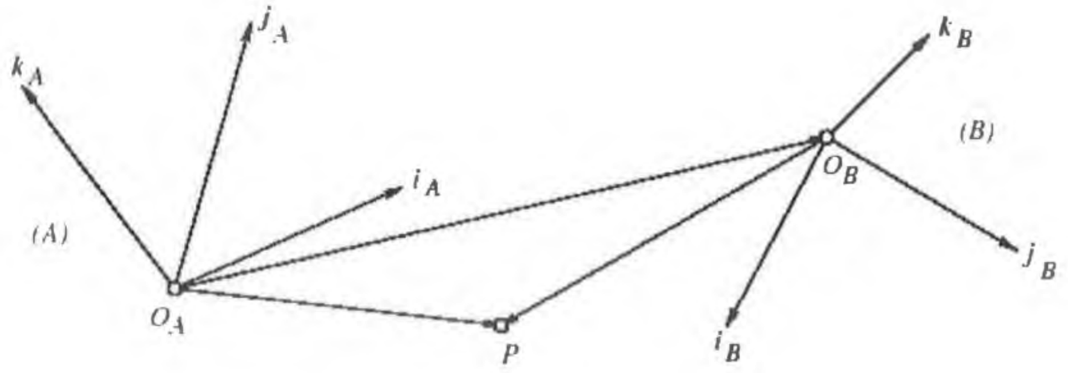


Figure 4.5 Rigid transformation

4.3.2 Camera Model for Two-Step Method:

In Tsai [12] model four steps of transformation from 3D World coordinate to Camera Coordinate is as follows:

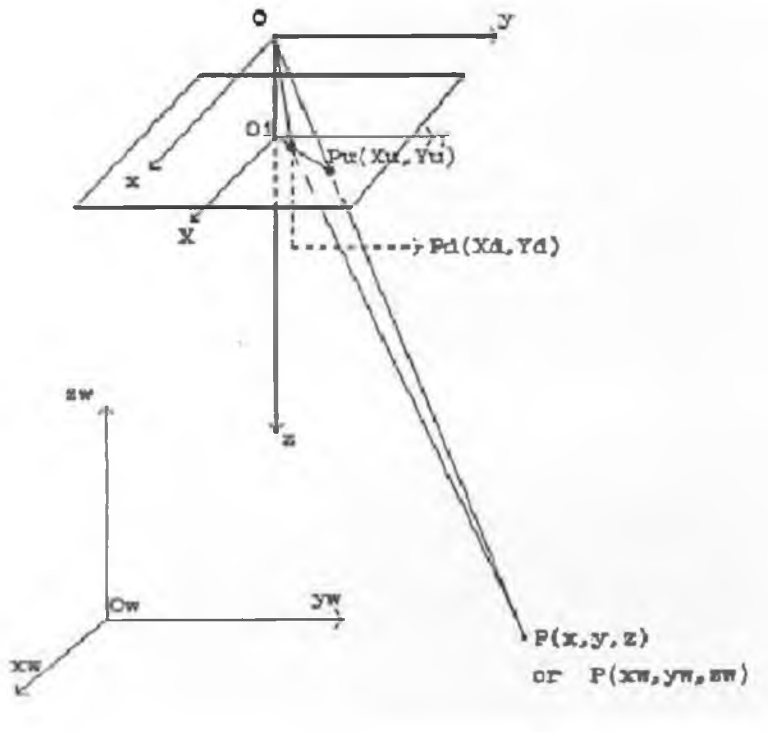


Figure 4.6 Camera Model

Figure-4.6 illustrates the camera model. (x_w, y_w, z_w) is the 3D coordinate of the object point P in the 3D world coordinate system. (x,y,z) is the 3D coordinate of the object point P in the 3D camera coordinate system, which is centered at O, the optical center, with the z axis the same as the optical axis. (X,Y) is the image coordinate system centered at O_i and parallel to x and y axes. f is the distance between the front image plane and the optical center. (X_u, Y_u) is the image coordinate of (x,y,z) if a perfect pin hole camera model is used. (X_d, Y_d) is the actual image coordinate whish differs from (X_u, Y_u) due to lens distortion. (X_f, Y_f) is the coordinate used in the computer frame memory. The overall transformation from (x_w, y_w, z_w) to (X_f, Y_f) needs four steps.

Step 1: From the equation 4.5 we can find the rigid body transformation from the world coordinate systems (x_w, y_w, z_w) to the camera 3D coordinate system

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = R \begin{bmatrix} x_w \\ y_w \\ z_w \end{bmatrix} + T \qquad \text{......................(4.6)}$$

where R and T is the 3x3 rotation matrix and the translation vector

$$R = \begin{bmatrix} r_1 & r_2 & r_3 \\ r_4 & r_5 & r_6 \\ r_7 & r_8 & r_9 \end{bmatrix}, \quad T = \begin{bmatrix} T_x & T_y & T_z \end{bmatrix}^T \qquad \text{......................(4.7)}$$

parameters to be calculated: *R* and *T*

Step 2: transformation from 3D camera coordinate (x,y,z) to ideal image coordinate (X_u, Y_u) using the perspective projection with pin hole camera geometry

$$X_u = f\frac{x}{z} \qquad Y_u = f\frac{y}{z} \qquad \text{........................(4.8)}$$

Parameters to be calibrated; effective focal length *f*

Step 3: Radial Lens distortion:

$$X_d + D_x = X_u \qquad Y_d + D_y = Y_u \qquad \text{.....................(4.9)}$$

where (X_u, Y_u) is the distortion or true image coordinate on the image plane

$$D_x = X_d(\kappa_1 r^2 + \kappa_2 r^4) \qquad D_y = Y_d(\kappa_1 r^2 + \kappa_2 r^4) \qquad r = \sqrt{(X_d^2 + Y_d^2)}$$

Now

$$X_d + X_d(\kappa_1 r^2 + \kappa_2 r^4) = X_u \quad \text{and} \quad Y_d + Y_d(\kappa_1 r^2 + \kappa_2 r^4) = Y_u$$

$$\Leftrightarrow X_d(1 + (\kappa_1 r^2 + \kappa_2 r^4)) = X_u \qquad \Leftrightarrow Y_d(1 + (\kappa_1 r^2 + \kappa_2 r^4)) = Y_u$$

$$\Leftrightarrow X_d = (1 + (\kappa_1 r^2 + \kappa_2 r^4))^{-1} X_u \qquad \Leftrightarrow Y_d = (1 + (\kappa_1 r^2 + \kappa_2 r^4))^{-1} Y_u \quad \text{...(4.10)}$$

Here κ_1 and κ_2 are distortion coefficient.

Step 4: In Tsai and Lenz [46], Real image coordinate (X_d, Y_d) to computer image (frame buffer) coordinate (X_f, Y_f) conversion:

$$X_f = s_x d_x'^{-1} X_d + C_X \qquad Y_f = d_y^{-1} Y_d + C_Y \qquad \text{.........(4.11)}$$

where

(C_X, C_Y) computer image center coordinates.

(d_x', d_y) distance between computer pels on the front image plane.

s_x is the scale factor or uncertainty factor in the x direction.

Here d_y is simply the distance between adjacent rows on the image sensor, i.e. vertical scale factor is exactly one, and therefore need not be calibrated.

Here

$$d_x' = d_x \frac{N_{cx}}{N_{fx}} \qquad \text{................(4.12)}$$

where

d_x distance between adjacent sensor elements in X (scan line) direction.

d_y distance between adjacent sensor element in Y direction.

N_{cx} Number of sensor elements in X (scan line) direction.

N_{fx} Number of picture elements in a line as sampled by the computer.

One can find the s_x explicitly by using the following equation

$$s_x = \frac{N_{cx}}{N_{fx}} \qquad \text{.................(4.13)}$$

Equation (4.13) is only valid when the time for the active portion of the video line of the camera is equal to that of the digitizing circuitry. This is not exactly true. The horizontal scale factor is actually equal to

$$s_x = \frac{f_c}{f_f} \qquad \text{..................(4.14)}$$

where

f_c Pel clock frequency of the camera.

f_f Sampling frequency of the A/D- converter.

Image Center:

Image center is defined as the frame buffer coordinates (C_x,C_y) of the intersection of the optical axis with the image plane. It is usually used as the origin of the imaging process and appears in the perspective equations. In high accuracy application, it also serves as the center of radially modelled lens distortion. Five different method exist for image center detection. In Tsai [46], five methods are divided into three group. Group 1: Direct Optical Method. Group 2: Method Varying focal length. Group 3: Radial alignment Method and Model fit Method.

Equation relating the 3D coordinate to the 2D computer image coordinates:

By combining the equations step 1 and step 2 we can write:

$$X_u = f\frac{r_1x_w + r_2y_w + r_3z_w + T_x}{r_7x_w + r_8y_w + r_9z_w + T_z} \qquad Y_u = f\frac{r_4x_w + r_5y_w + r_6z_w + T_y}{r_7x_w + r_8y_w + r_9z_w + T_z}$$

Now from step three and step four we can write:

$$X_f = s_x d_x^{\prime -1}(1+(\kappa_1 r^2 + \kappa_2 r^4))^{-1} X_u + C_X \quad \text{and} \quad Y_f = d_y^{-1}(1+(\kappa_1 r^2 + \kappa_2 r^4))^{-1} Y_u + C_Y$$

$$\Leftrightarrow X_f - C_X = s_x d_x^{\prime -1}(1+(\kappa_1 r^2 + \kappa_2 r^4))^{-1} X_u$$

$$\Leftrightarrow Y_f - C_Y = d_y^{-1}(1+(\kappa_1 r^2 + \kappa_2 r^4))^{-1} Y_u$$

$$\Leftrightarrow X = s_x d_x^{\prime -1}(1+(\kappa_1 r^2 + \kappa_2 r^4))^{-1} X_u \qquad \Leftrightarrow Y = d_y^{-1}(1+(\kappa_1 r^2 + \kappa_2 r^4))^{-1} Y_u$$

$$\Leftrightarrow X s_x^{-1} d_x^{\prime}(1+(\kappa_1 r^2 + \kappa_2 r^4)) = X_u \qquad \Leftrightarrow Y d_y(1+(\kappa_1 r^2 + \kappa_2 r^4)) = Y_u$$

$$\Leftrightarrow X s_x^{-1} d_x^{\prime}(1+(\kappa_1 r^2 + \kappa_2 r^4)) = f\frac{r_1x_w + r_2y_w + r_3z_w + T_x}{r_7x_w + r_8y_w + r_9z_w + T_z}$$

and

$$\Leftrightarrow Y d_y(1+(\kappa_1 r^2 + \kappa_2 r^4)) = f\frac{r_4x_w + r_5y_w + r_6z_w + T_y}{r_7x_w + r_8y_w + r_9z_w + T_z} \quad \ldots\ldots\ldots\ldots\ldots\ldots(4.15)$$

The parameters used in the transformation in Figure 4.6 can be categorized into the following two classes:

Extrinsic Parameter:

The parameter in Step 1 in Figure 4.6 for the transformation from 3D object world coordinate system to the camera 3D coordinate system centred at the optical centre are called the *extrinsic parameter.*

There are six extrinsic parameters: the Eular angles yaw θ, pitch ϕ and tilt ψ for rotation, the three components for the translation vector T. The rotation matrix R can be expressed as function of θ ,ϕ and ψ as follows:

$$R = \begin{bmatrix} \cos\varphi\cos\theta & \sin\varphi\cos\theta & -\sin\theta \\ -\sin\varphi\cos\phi + \cos\varphi\sin\theta\sin\phi & \cos\varphi\cos\phi + \sin\varphi\sin\theta\sin\phi & \cos\theta\sin\phi \\ \sin\varphi\sin\phi + \cos\varphi\sin\theta\cos\phi & -\cos\varphi\sin\phi + \sin\varphi\sin\theta\cos\phi & \cos\theta\cos\phi \end{bmatrix} \quad \ldots\ldots\ldots\ldots(4.16)$$

Intrinsic parameters:

The parameters in step 2,3, and 4 in Figure 4.6 for the transformation from the object coordinate in the camera coordinate system to the compute image coordinate are called the *intrinsic parameters*. There are six intrinsic parameters:

f: the effective focal length

$\kappa_1\kappa_2$: lens distortion coefficients

s_x: uncertainty scale factor for x

(C_x,C_y): computer image coordinate for the origin in the image plane

Radial alignment method and model method:

Experimental set-up for calibration parameter:

The calibration pattern sheet had 9 black square block of 1inch x 1inch that shown in Figure 3.5. The corner of the 9 squares were treated as the calibration points, making a total of 36 points. The corners of the image of the calibration pattern were determined by using the Smith [48], "SUSAN corner detection algorithm" and output is shown in the Figure 4.7 and class definition is given in Appendix D. Since the calibration points are on a common plane, the world coordinate system can be chosen such that z_w= 0, and origin is not close to the center of the view or y axis of the camera coordinate system. Since the (x_w,y_w,z_w) is user defined, and the origin is arbitrary, it is no problem setting the origin of (x_w,y_w,z_w) to be out of the field of view and not close to the y axis. The purpose of the later is to make sure that T_y is not exactly zero.

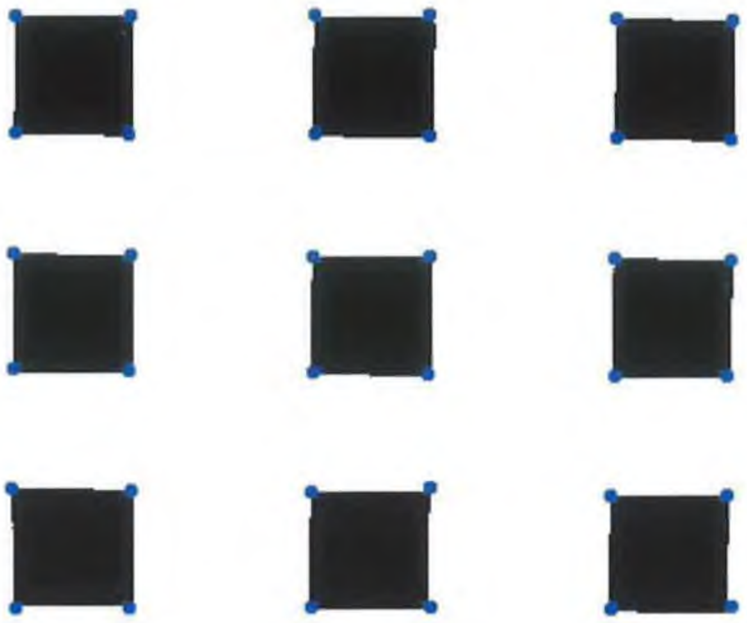


Figure 4.7 Corner point

The Camera class is responsible for camera calibration and all the conversion from image coordinate to world coordinate or from world coordinate to image coordinate. This class is the object oriented approach of the Roger Tsai camera calibration C source code [49].

```
class Camera: public camera_parameters ,public calibration_data, public
calibration_constants
{
        double   Xd[MAX_POINTS],          // x coordinate of distorted image pixel
                 Yd[MAX_POINTS],          // y coordinate of distorted image pixel
                 r_squared[MAX_POINTS],   // [mm^2]
                 U[7];
        int NoOfWcPoint;                  // no of world coordinate points
        char  camera_type[256];           // storing camera name
        void (Camera::*the_tempfunction)(integer *,integer *,doublereal *, doublereal
              *);
```

```
public:
        Camera();
        BOOL IsCalibrated();
        int  WCDataFileRead(TCHAR *);
        int  WCDataFileReadFive(TCHAR *);
        void  WcDataPrint(CDC*);
        void  ImageDataInsert(int, double, double);
        BOOL  ImageDataSave(char* filename, int TotalPoint);
        void  print_cp_cc_data(TCHAR *);
        void  print_error_stats(TCHAR *);
        void  DisplayCpCcdata(CDC*);
        void  PrintAllData(TCHAR *);
        void  ExistingAllCameraCpCcRead(TCHAR *);
        int retWCDataPointNo(){ return NoOfWcPoint;}
        void  initialize_newcamera_coplanar_parms (double XNo, double YNo,
            double Xdis, double Ydis,  double PiXno, double PiYno, double Adfre,
            double  CaFre);
        void  initialize_newcamera_coplanar_parms (double XNo, double YNo,
             double Xdis,  double Ydis,  double PiXno, double PiYno);
        void initialize_newcamera_noncoplanar_parms(double XNo, double YNo,
             double Xdis, double Ydis, double PiXno, double PiYno);
        void  initialize_photometrics_parms ();
        void  initialize_general_imaging_mos5300_matrox_parms ();
        void  initialize_panasonic_gp_mf702_matrox_parms ();
        void  initialize_sony_xc77_matrox_parms ();
        void  initialize_sony_xc57_androx_parms ();
        void  initialize_sony_xc75_matrox_parms ();
        void  initialize_kodakes310_parms();
        void  initialize_xapshot_matrox_parms ();
        void  initialize_nikon_coolpix800_parms();
        void  solve_RPY_transform ();
        void  apply_RPY_transform ();
        void  cc_compute_Xd_Yd_and_r_squared ();
        void  cc_compute_U ();
void  cc_compute_Tx_and_Ty ();
void  cc_compute_R ();
void  cc_compute_approximate_f_and_Tz ();
void cc_compute_exact_f_and_Tz_error (integer *m_ptr,integer  *n_ptr,
        doublereal *params,doublereal *err);
void  cc_compute_exact_f_and_Tz ();
void  cc_three_parm_optimization ();
void  cc_remove_sensor_plane_distortion_from_Xd_and_Yd ();
void  cc_five_parm_optimization_with_late_distortion_removal_error (integer
       *m_ptr, integer *n_ptr,doublereal *params,doublereal *err);
void  cc_five_parm_optimization_with_late_distortion_removal ();
void cc_five_parm_optimization_with_early_distortion_removal_error(
        integer  *m_ptr, integer *n_ptr,doublereal *params,doublereal *err);
void  cc_five_parm_optimization_with_early_distortion_removal ();
void  cc_nic_optimization_error (integer *m_ptr,integer *n_ptr, doublereal
        *params, doublereal *err);
void  cc_nic_optimization ();
void cc_full_optimization_error (integer *m_ptr, integer *n_ptr,doublereal
       *params, doublereal *err);
void  cc_full_optimization ();
void  ncc_compute_Xd_Yd_and_r_squared ();
void  ncc_compute_U ();
void  ncc_compute_Tx_and_Ty ();
void  ncc_compute_sx ();
void  ncc_compute_R ();
void  ncc_compute_better_R ();
void  ncc_compute_approximate_f_and_Tz ();
void ncc_compute_exact_f_and_Tz_error (integer *m_ptr,integer *n_ptr,
       doublereal *params, doublereal *err);
void  ncc_compute_exact_f_and_Tz ();
void  ncc_three_parm_optimization ();
void  ncc_nic_optimization_error (integer *m_ptr,integer *n_ptr,doublereal
        *params,doublereal *err);
void  ncc_nic_optimization ();
void  ncc_full_optimization_error (integer *m_ptr,integer *n_ptr,doublereal
        *params,doublereal *err);
void  ncc_full_optimization ();
void  coplanar_calibration ();
void  coplanar_calibration_with_full_optimization ();
void  noncoplanar_calibration ();
void  noncoplanar_calibration_with_full_optimization ();
void  coplanar_extrinsic_parameter_estimation ();
void  noncoplanar_extrinsic_parameter_estimation ();
double CBRT (double x);
void  world_coord_to_image_coord (double xw,double yw,double zw,double
       *Xf,double *Yf);
void  image_coord_to_world_coord (double Xfd,double Yfd,double
        zw,double *xw,double *yw);
void  world_coord_to_camera_coord (double xw,double yw,double zw,double
       *xc,double *yc,double *zc);
void  camera_coord_to_world_coord (double xc,double yc,double zc,double
       *xw,double *yw,double *zw);
void  distorted_to_undistorted_sensor_coord(double Xd,double Yd,double
       *Xu,double *Yu);
void  undistorted_to_distorted_sensor_coord (double Xu, double Yu,double
       *Xd,double *Yd);
void  distorted_to_undistorted_image_coord (double Xfd,double Yfd,double
       *Xfu,double *Yfu);
void  undistorted_to_distorted_image_coord(double Xfu,double  Yfu,double
        *Xfd,double *Yfd);
void  wolrd_to_image_mine(double xw,double yw,double zw,double *Xf,
        double *Yf,double Angle);
void  distorted_image_plane_error_stats (double *, double *,double*,
        double*);
void  undistorted_image_plane_error_stats (double *, double *,double *,
        double *);
void  object_space_error_stats (double *, double *, double *, double *);
void  normalized_calibration_error (double *, double *);
```

```
        doublereal dpmpar_(integer *i);
        doublereal enorm_(integer *n, doublereal *x);
        int lmdif_(integer *m, integer *n,doublereal *x,doublereal *fvec,doublereal
                *ftol,doublereal *xtol,doublereal *gtol,integer *maxfev, doublereal
                *epsfcn,doublereal *diag,integer *mode,doublereal *factor,integer
                *nprint,integer *info,integer *nfev,doublereal *fjac,integer
                *ldfjac,integer *ipvt,      doublereal *qtf,doublereal *wa1, doublereal
                *wa2, doublereal *wa3,doublereal *wa4);
        int fdjac2_( integer *m,integer *n,doublereal *x,doublereal *fvec,doublereal
               *fjac,   integer *ldfjac,integer *iflag,doublereal *epsfcn,doublereal
               *wa);
        int lmpar_(integer *n,doublereal *r,integer *ldr,integer *ipvt,doublereal
               *diag,doublereal *qtb, doublereal *delta,doublereal *par,doublereal *x,
               doublereal *sdiag, doublereal *wa1,doublereal *wa2);
        int qrsolv_(integer *n, doublereal *r,integer *ldr,integer *ipvt,doublereal
               *diag,       doublereal *qtb, doublereal *x, doublereal *sdiag, doublereal
               *wa);
        int qrfac_(integer *m,integer *n,doublereal *a,integer *lda,logical *pivot,
               integer  *ipvt, integer *lipvt, doublereal *rdiag, doublereal
               *acnorm,doublereal *wa);
        double    ran1 (int *idum);
        double    gasdev (int *idum);
};  //End of the camera class definition
```

The **Camera** class is sub class of **class camera_parameters, class calibration_data, and class calibration_constants.**

```
class camera_parameters {
protected:
   double   Ncx;        // number of sensor element in x direction
   double   Nfx;        // number of picture element in x direction
   double   dx;         // distance between adjacent sensor element in X direction
   double   dy;         // distance between adjacent sensor element in Y direction
   double   dpx;        // Effective X dimension of pixel in frame grabber
   double   dpy;     // Effective Y dimension of pixel in frame grabber
   double   Cx;      //  X coordinate of the Image centre
   double   Cy;      // Y coordinate of the Image centre
   double   sx;      //  Scale factor of the camera
 public:
        double retCx(){return Cx;}   //returns the x of the image centre
        double retCy(){return Cy;}   //returns the y of the image centre
}; //End of the camera_parameters class definition
```

The camera_parameter calss is responsible for storing the camera parameter information.

```
class calibration_data {
protected:
   int     point_count;              /* [points]       */
   double   xw[MAX_POINTS];     /* x values of the world coordinate in [mm]    */
   double   yw[MAX_POINTS];     /* y values of the world coordinate in [mm]    */
   double   zw[MAX_POINTS];     /* z values of the world coordinate in [mm]    */
   double   Xf[MAX_POINTS];     /* x values of the image coordinate in  [pix]    */
   double   Yf[MAX_POINTS];     /* y values of the image coordinate in [pix]     */
public:
        inline int ret_point_count(){return point_count;}; //returns no of world
                                                           //coordinate point
        inline double * ret_xw(){return xw;};        //returns pointer of the xw[] array
        inline double * ret_yw(){return yw;};        //returns pointer of the yw[] array
        inline double * ret_zw(){return zw;};        //returns pointer of the zw[] array
        inline double * ret_Xf(){return Xf;};        //returns pointer of the Xf[] array
        inline double * ret_Yf(){return Yf;};        //returns pointer of the Yf[] array
}; //End of the calibration_data class definition
```

The **calibration_data** class stores the world coordinate data in xw[], yw[],zw[] arrays and image coordinate data in the Xf[] and Xf[].

```
class calibration_constants {
protected:
   double  f;            /* focal length of the camera in [mm]    */
   double  kappa1;       /* distortion coefficients in [1/mm^2]    */
   double  p1;           /* [1/mm]     */
   double  p2;           /* [1/mm]     */
   double  Tx;           /* x value of the translation vector in [mm]    */
   double  Ty;           /* y value of the translation vector in [mm]    */
   double  Tz;           /* z value of the translation vector in [mm]    */
   double  Rx;           /* [rad] */
   double  Ry;           /* [rad] */
   double  Rz;           /* [rad] */
   double  r1;           /* first element of first row in rotation matrix []        */
   double  r2;           /* second element of first row in rotation matrix[]     */
   double  r3;           /* third element of first row in rotation matrix[]        */
   double  r4;           /* first element of second row in rotation matrix[]      */
   double  r5;           /* second element of second row in rotation matrix[]  */
   double  r6;           /* third element of second row in rotation matrix[]     */
   double  r7;           /* first element of third row in rotation matrix[]        */
   double  r8;           /* second element of third row in rotation matrix[]     */
   double  r9;           /* third element of third row in rotation matrix[]       */
}; //End of the calibration_constant class definition
```

The **calibration_constants** contains the information of rotation and transformation matrix.

According to purpose **Camera** class methods can be divided in to following categories:

a. Camera data save to a file or read from a file.
b. Camera parameter initialization.
c. Coplanar and coplanar calibration methods.
d. Coordinate conversion methods.
e. Non linear optimization methods.

a) Camera data save to a file or read from a file:

The implementation of this methods read world coordinate data and image data from a file and write camera calibration data in a file.

The *WCDataFileRead* method reads world coordinate (xw, yw, zw) data from a existing file.

The *WCDataFileReadFive* method reads world coordinate data and image data (xw, yw, zw, Xf, Yf) from a existing file.

The *ImageDataInsert* method is used to inserting the image corner point in the Camera Object.

The *ImageDataSave* method is used to save image data in a file.

The *print_cp_cc_data* method is used to save Camera parameter and Calibration Constant data in a file.

The *print_error_stats* method is used to save error to save in a file.

The *PrintAllData* method is used to save Camera parameter and camera constant in a file.

The *ExistingAllCameraCpCcRead* method reads the camera parameter and camera constants data from an existing file.

The *retWCDataPointNo* method return the no of world coordinate data.

b) Camera parameter initialization:

These methods are used to set camera parameter data for camera calibration.

The *initialize_newcamera_noncoplanar_parms* method sets the noncoplanar camera parameter for an unknown camera.

There are two *initialize_newcamera_coplanar_parms* methods to set coplanar camera parameter for unknown camera.

The *initialize_photometrics_parms* method sets the camera parameter for Photometrics Star I camera.

The *initialize_general_imaging_mos5300_matrox_parms* method sets the camera parameter for General Imaging MOS5300 + Matrox camera.

The *initialize_sony_xc75_matrox_parms* method sets the camera parameter for Sony XC75 + Matrox camera.

The *initialize_sony_xc77_matrox_parms* method sets the camera parameter for Sony XC77 + Matrox camera.
The *initialize_kodakes310_parms* method sets the camera parameter for Kodak Es310 camera.
The initialize_nikon_coolpix800_parms method sets the camera parameter for Nikon CoolPix 800 camera.

c) Coplanar and noncoplanar calibration methods:

These methods are used to run the coplanar calibration or noncoplanar calibration.
Coplanar methods are:
The *cc_compute_Xd_Yd_and_r_squared* method computes the distort image.
The *cc_compute_U* method computes the five unknowns r1/Ty, r2/Ty.
The *cc_compute_Tx_and_Ty* method computes the Ty and Tx.
The *cc_compute_R method* is for the rotation matrix determination.
The *cc_compute_approximate_f_and_Tz* is for finding the effective focal length.
The *cc_compute_exact_f_and_Tz_error* method is for finding error of undistorted sensor coordinate.
The *cc_compute_exact_f_and_Tz* is for finding exact focal length using nonlinear optimization.
The *cc_three_parm_optimization* method is for coplanar calibration with three parameters (Tz, f, kappa1) optimization.
The *cc_remove_sensor_plane_distortion_from_Xd_and_Yd* method is for removing sensor distortion from Xd and Yd.
The *cc_five_parm_optimization_with_late_distortion_removal* method is used for full coplanar optimization calibration for five parameters (Tz, f, kappa1, Cx, Cy).
The *cc_five_parm_optimization_with_early_distortion_removal* method does better five parameters optimization for full optimization coplanar calibration.
The *cc_nic_optimization* method does full optimization without image center.
The *cc_full_optimization* method does full optimization with full optimization.
The *coplanar_calibration* method is for coplanar calibration without optimization.
The *coplanar_calibration_with_full_optimization* method is for coplanar calibration with optimization.

Non Coplanar methods are:

The *ncc_compute_Xd_Yd_and_r_squared* method computes the distort image.

The *ncc_compute_U* method computes the five unknowns r1/Ty, r2/Ty.

The *ncc_compute_Tx_and_Ty* method computes the Ty and Tx.

The *ncc_compute_sx* method calculates scale factor non coplanar calibration.

The *ncc_compute_R method* is for the rotation matrix determination.

The *ncc_compute_better_R* method is used for better rotation matrix calculation.

The *ncc_compute_approximate_f_and_Tz* is for finding the effective focal length.

The *ncc_compute_exact_f_and_Tz_error* method is for finding error of undistorted sensor coordinate.

The *ncc_compute_exact_f_and_Tz* is for finding exact focal length using nonlinear optimization.

The *ncc_three_parm_optimization* method is for noncoplanar calibration with three parameters (Tz, f, kappa1) optimization.

The ncc_nic_optimization method is for optimization without the center point.

The *ncc_full_optimization* method is for noncoplanar full optimization camera calibration.

The *noncoplanar_calibration* method is for noncoplanar calibration without optimization.

The *noncoplanar_calibration_with_full_optimization* method is noncoplanar calibration with optimization.

d) Coordinate conversion methods:

These methods are used for world coordinate to image coordinate, world coordinate to camera coordinate, camera coordinate to image coordinate conversion or vice versa.

The *world_coord_to_image_coord* method takes the position of a point in world coordinates [mm] and determines the position of its image in image coordinates [pix].

The *image_coord_to_world_coord* method performs an inverse perspective projection to determine the position of a point in world coordinates that corresponds to a given position in image coordinates.

The *world_coord_to_camera_coord* method takes the position of a point in world coordinates [mm] and determines its position in camera coordinates [mm].

The *camera_coord_to_world_coord* method takes the position of a point in camera coordinates [mm] and determines its position in world coordinates [mm].

The *distorted_to_undistorted_sensor_coord* method converts from distorted to undistorted sensor plane coordinates.

The *undistorted_to_distorted_sensor_coord* method converts from undistorted to distorted sensor coordinate.

The *distorted_to_undistorted_image_coord* method converts point from distorted to undistorted image coordinate.

e) Non linear optimization methods:

The methods which are belong to this category are:

The *dpmpar_* method provides double precision machine parameters when the appropriate set of data statements is activated and all other data statements are rendered inactive.

The *enorm_* methods calculates the Euclidean norm of x, for given an n-vector x.

The *lmdif_* method is used to minimize the sum of the squares of m nonlinear functions in n variables by a modification of the levenberg-marquardt algorithm. The user must provide a subroutine which calculates the functions. The jacobian is then calculated by a forward-difference approximation

The *fdjac2_* method computes a forward-difference approximation to the m by n Jacobian matrix associated with a specified problem of m functions in n variables.

Let an m by n matrix a, an n by n nonsingular diagonal matrix d, an m-vector b, and a positive number delta, the problem is to determine a value for the parameter par such that if x solves the system a*x = b, sqrt(par)*d*x = 0, in the least squares sense, and dxnorm is the euclidean norm of d*x, then either par is zero and (dxnorm-delta) .le. 0.1*delta , or par is positive and abs(dxnorm-delta) .le. 0.1*delta.

The *lmpar_* subroutine completes the solution of the problem if it is provided with the necessary information from the qr factorization, with column pivoting, of a.

The *qrsolv_* method solves a*x = b, d*x = 0 in the least squares sense.

The *qrfac_* subroutine uses householder transformations with column pivoting (optional) to compute a qr factorization of the m by n matrix a.

The *solve_RPY_transform* method solves for the roll, pitch and yaw angles (in radians) for a given orthonormal rotation matrix.
The *apply_RPY_transform* method simply takes the roll, pitch and yaw angles and fills in the rotation matrix elements r1-r9.

4.4 Silhouette Extraction:

Two standard techniques exist for silhouette extraction. The first one relies on a uniform background segmented by using a colour-criterion. The second technique is the background subtraction method which uses another image of the scene without the object. The difference of this image with any image of the sequence gives us the object silhouette after direct threshold. The second method was implemented in this project. However this method often confuses the object and the background for several reasons:
1) an object may naturally have some regions with similar colour as the chosen background.
2) radiosity effect may happen, e.g. a blue background tinting the object borders with a bluish colour, especially at dark and shaded parts,
3) shaded dark regions may be confused with a black background, or highlights with a white background.

In this project, naive silhouette extraction, or pixel level extraction, was performed. More concretely, pixel-level subtraction between an object image and the background image is performed. If the absolute value of the subtraction for a pixel is greater than the given threshold, the pixel is regarded as a part of the silhouette.
Figure 4.8 shows the original image and Figure 4.9 shows its silhouette image.

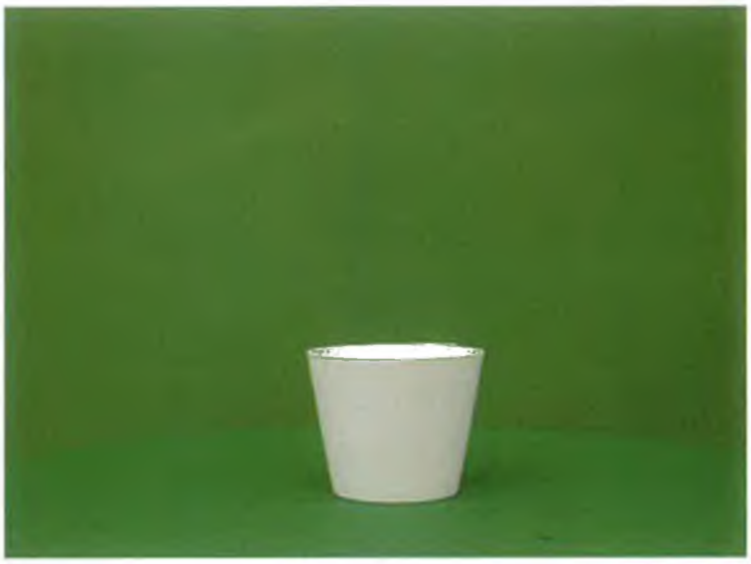


Figure 4.8 Real Object

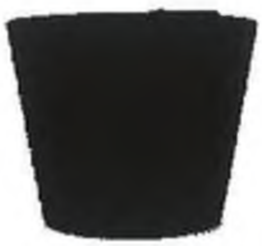


Figure 4.9 Silhouette Object

Silhouette class is responsible for creating a silhouette image using the lowest and highest colour value of the background image. Its definition looks as follows

```
class silhouette
{
        unsigned  int imagewidth;          //width of the image
        unsigned  int imageheight;         //height of the image
        BYTE *bits;                        //pointer to the image point
        unsigned long *ImageAlloc;        //pointer for image allocation
        BYTE *BinaryImage;                //pointer for binary image
        int startx;                       //x value of the upper left corner of the image
        int starty;                       //y value of the upper left corner of the image
public:
        silhouette(int width,int height,short startx, short starty);
        silhouette(int width,int height, short startx, short starty,Image *image,CDC*
              pDc);
        void drawimage(CDC* pDC, unsigned long BackValue,unsigned long
              FrontValue);
        void drawImageSilhouetteCalib(CDC* pDC, unsigned long , unsigned long);
        BYTE* retBinaryImage();
        void FreeBinaryAlloc();
        ~silhouette();
 }; //End of the silhouette calss definition
```

The silhouette class has two constructors, three methods and a destructor. The first constructor takes four parameters, which is used for the calibration image silhouette creation. The second constructor takes six parameters, and is used for the normal image silhouette creation. Parameters width and height mean the width and height of the image respectively. Parameters starts and starty mean the starting coordinate (x,y) of the image. Parameters *image and *pDC in the second constructor are the Image class object and device context class object respectively. Image class is discussed in Appendix C.

The *drawimage* method draws the silhouette image in the screen. It takes three parameters, *pDC* is the device context, *BackValue* and *FrontValue* are the lowest and highest value of the background image.

The *drawImageSilhouetteCalib* method is used for creating silhouette image for calibration image.
The *retBinaryImage* method returns the binary image means silhouette image.
The *~silhouette* method is the destructor of the silhouette class.

4.5 Shape Modelling:

Shape modelling deals with construction of a 3D image from 2D information. 2D information is acquired by quadtree creation from the silhouettes of each image. This 2D quadtree information is converted to 3D using Voting and Octree construction.

4.5.1 Quadtree Construction:

Quadtrees are hierarchical data structures used for compact representations of two-dimensional images [31]. A quadtree is generated by dividing an image into quadrants and repeatedly subdividing the quadrants into subquadrants until each quadrant has uniform colour (e.g. "1" or "0" in a binary image). This yields a representation of an image as image as a tree of out degree four. The root node of the tree corresponds to the entire image. A node in a quadtree is either a leaf (terminal node) or has four son nodes (nonterminal node). Each son node is associated with a quadrant of the block corresponding to its father node. Figure 4.10a,b shows the original image and its quadtree.

An early study of quadtree structures for application in image processing was done by Sidhu and Boute [51]. A comprehensive survey for the history of quadtrees has been provided by Klinger and Dyer [52]. Tanimoto and Pavlidis [53] have developed a recursive refining algorithm for edge detection utilizing quadtrees and they have demonstrated its computational savings. An overview of tree structures for region representation has been presented by Samet and Rosenfeld [54].

Here, quadtree construction is done by the Warnock Algorithm [54]. This algorithm considers a window (or area) in image (or object) space, and seeks to determine if the window is empty or if the contents of the window are simple enough to resolve. If not, the window is subdivided until either the contents of a subwindow are simple enough to determine, or the subwindow size is at the limit of desired resolution.

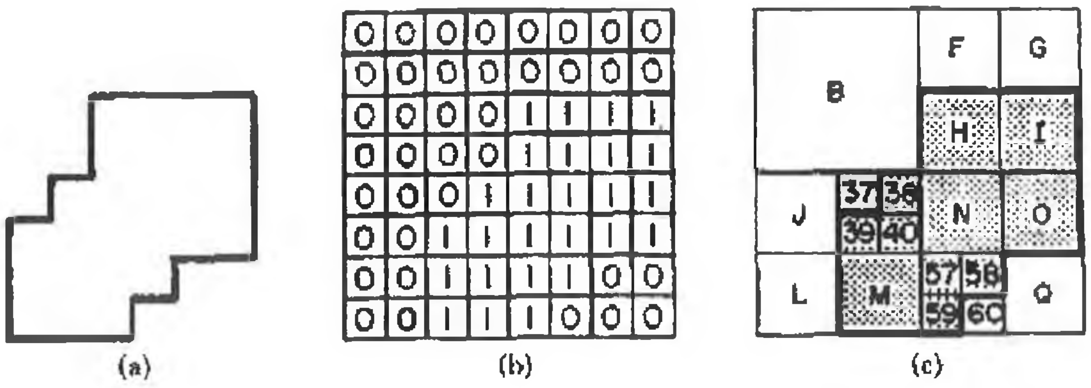


Figure 4.10a Image of an object and quadtree generation

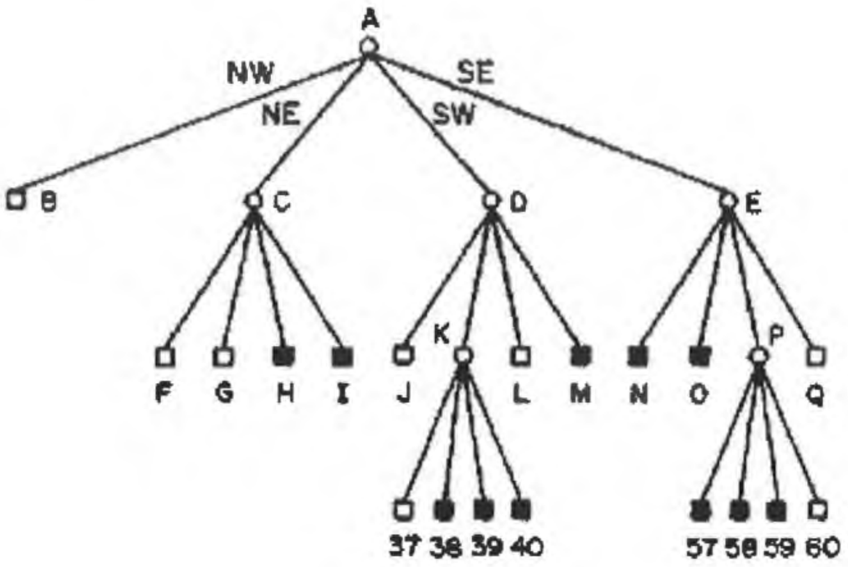


Figure 4.10b Quadtree representation

The Window class and the Warnock class are implemented for creating quadtree from silhouette images. The definition of the window class is as follows

```
class Window
{
        int Xleft, Ytop, Xright, Ybottom, size;
public:
        Window(int, int, int);
        Window(int, int, int, int);
        Window();
        inline int XleftReturn(){ return Xleft;};
        inline int XrightReturn(){ return Xright;};
        inline int YtopReturn(){return Ytop;};
        inline int YbottomReturn(){ return Ybottom;};
        inline int SizeReturn(){return size;};
        ~Window();
}; //End of the Window class definition
```

Each Window class object represents the quadrant of the quadtree. This quadrant objects are used in the Warnock class.

The Window() method is the default constructor.

The Window(int,int,int) is another constructor used to create quadrants in the position of (x,y) supplied by first two parameter. Quadrants size will as specified by third parameter.

The XleftReturn() returns x of the top left corner of the quadrant.

The XrightReturn() returns x of the bottom left corner of the quadrant.

The YtopReturn() returns y of the top left corner of the quadrant.

The YbottomReturn() returns y of the bottom right corner of the quadrants.

The SizeReturn() returns the height of the quadrants. Here quadrants are square.

The ~Window() is the destructor of the Window class.

The Warnock class definition is look like as follows:

```
class Warnock
{
        int originX, originY;
        list<Window> TemporaryHold;          //stores the Window class objects
        map<Window,short> ObjectHold;        /*stores the Window class object and
                                               its color */
        vector<Window3D> tempwindow3d;  //stores the Window3D class objects
        vector<cuboid> Octree;               //stores the cuboid class object
        vector<EdgePoint2D> WindowEdge;  //stores the EdgePoint2D class object
        int NoOfBlackWindow;            //no of object window object in the silhouette
        int NoOfWindow;                 //total of window object in the silhouette
        double centerX, centerY, centerZ;
        double lX,lY,lZ,hX,hY,hZ;
protected:
        void ObjectTest(Window , CDC*, Warnock *);
public:
        Warnock(void);
        void runSubdivision(int width,int height, Warnock *, CDC*);
        void push(Window );
        void objectpush(Window ,short);
        Window  pop();
        void Display(CDC* pDC);
        void Display(CDC*, short );
        inline int SizeReturn(){return TemporaryHold.size();}
        inline int MapSizeReturn(){return ObjectHold.size();}
        inline  tempmap Ret_map(){ return ObjectHold;};
        int retNoOfBlackWindow(){ return NoOfBlackWindow;}
        int retNoOfWindow() { return NoOfWindow;}
        void QuadTreeRefine(Warnock);
        void quad_2d_3d(Camera);
        void quad_2d_3d(Camera,double,double,double);
        void OctreeCreate(double FirstAngle, double DiffAngle);
        void DrawOctree(HDC hdc, Camera);
        int OctreeSize();
        vector<cuboid>  retOctree();
        void CenterSet(double,double,double);
        void SurfaceDetection();
        void CubePush(cuboid temp);
        void EdgeLine2D(int, int, int, int, BYTE );
        int Sign(int, int);
        vector<EdgePoint2D> retWindowEdge();
        void EdgeDraw(CDC* );
        BOOL RotateOctree(double angle, Camera CameraObject, CDC* pDC);
        double centerx(){ return centerX;}
        double centery(){ return centerY;}
        double centerz(){ return centerZ;}
        double lx() { return lX;}
        double ly() { return lY;}
        double lz() { return lZ;}
        double hx() { return hX;}
        double hy() { return hY;}
        double hz() { return hZ;}
        void ClearObjectHold();
        BOOL SaveDxf(char *FileName, vector<Warnock> temp);
        ~Warnock();
}; //End of the Warnock class definition
```

The implementation of the class creats a quadtree from the silhouette image and an octree from the quadtree. Octree creation methods will be discussed in the Octree creation section.

The important properties of the Warnock class are ObjectHold, Octree, WindowEdge. Vector, List, Map [56] are the three important dynamic array system of C++. They belong to the Standard Template Library of the C++. ObjectHold is a Map to hold quadrants of the quadtree and its colour (0's or 1's). Octree is a Vector to hold octree object. TemporaryHold is a List to hold quadrants for temporary use. WindowEdge is a Vector to hold the edge quadtree.

The *Warnock* method is the default constructor of the class.
The *Push(Window)* method inserts the window object in the TemporaryHold List.
The *objectpush(Window ,short)* method inserts window object and its colour in the ObjectHold map dynamic array.
The *pop* method pops up a window object from the TempoaryHold list dynamic array.
The *runSubdivision* method is used to start the quadtree creation. It calls the *ObjectTest* method to complete the quadtree creation.
The *ObjectTest* method is used to complete quadtree after *runSubdivision* starts quadtree creation and calls it. It is the only protected method in the class.
The *Display* method displays the quadtree object in the screen.
The *Display(CDC* pDC,short no)* method is used to partial display of the quadtree. Second parameter *no* is the number of the quadrant you want to display in the screen.

The *SizeReturn* method returns how many quadtree in the TemporaryHold object.
The *MapSizeReturn* method returns the quantity of the qudrants in the quadtree *ObjectHold* Map.
The *Ret_map* method returns the ObjectHold Map object.
The *retNoOfBlackWindow* returns the number of object quadrant in the quadtree.
The *ClearObjectHold* method deletes all window objects in the ObjectHold dynamic array.

All other methods in the Warnock class are used in Octree creation and Texture mapping.

4.5.2 Voting:

Assuming the perspective projection model and using the parameters obtained in 4.2 (camera calibration parameter) three dimensional voxel spaces with the cylindrical coordinate system was constructed from the quadtrees (Figure 4.11).

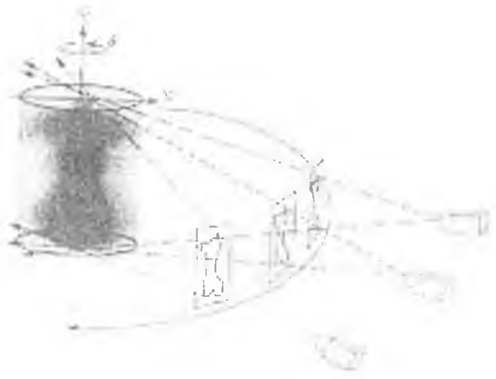


Figure 4.11: Voting in three dimensional space

Voting was done by the implementation of the few methods of Warnock class and CSingleView class. Warnock class definition is given in 4.4.1.

The *quad_2d_3d* method is used to convert the 2D quadtree from image coordinate to world coordinate using the Camera class object. It sets all the 2D quadtree to 3D setting the Z value to zero that means all the 3D quadtree will be in same plane. It store the 3D quadtree in the *tempwindow3d* vector. Each quadrant of the 3D quadtree is the object of Window3D calss.

Definition of the Window3D class looks like:

```
class Window3D
{
        Point fpoint;       //top left corner of the window
        Point spoint;       //bottom right corner of the window
public:
        Window3D(Point first, Point second);
        Point first();
        Point second();
        ~Window3D();
}; //End of the Window3D class definition
```

Window3D class has two members variable of Point class object. Window3D method is the constructor of this class and first and second methods return the two member variables. Point class is discussed in Appendix B.
Voting of this *tempwindow3d* vector is done by the Display_3D method in the CSingleView class, which is discussed in the Appendix A.

4.5.3 Octree Construction:

In references [57,58] an Octree was generated by dividing a universe (generally a $2^N \times 2^N \times 2^N$ cube) into octants and subdividing the octants into suboctants until all the voxels (volume elements) in each octant lie entirely within an object or outside object (Figure 1.12). In [9] octree was created from quadtree. In their approach, at first the quadtrees of the three silhouettes from three orthogonal views of an object are generated. Each quadtree was then copied along the associated viewing direction to obtain an octree representing a cylinder that was the sweeping volume of the silhouette along the viewing direction.

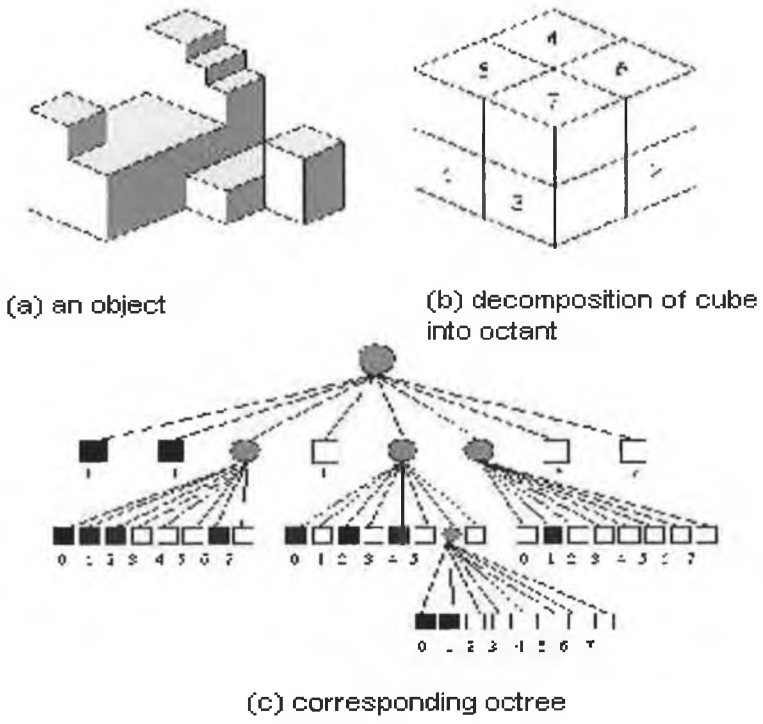


Figure 1.12 Octree of an object

The proposed method creates an octree from multiple quadtrees which are not orthogonal. Firstly, each quadtree is refined projecting it onto the next quadtree. Then a 2D quadtree is converted to a 3D quadtree and rotated according their angle of rotation. This is called voting which was discussed 4.5.2. After voting all the voted refined quadtrees are converted to octree.

Octree is done by the implementation of the few methods of Warnock class.
The *QuadTreeRefine* method takes the second quadtree as parameter and refine the first quadtree with the second one.
The *OctreeCreate* method takes two parameters. One is its angle or position in the 3D space and second is the angle of the next quadtree. It creates the Octree for each quadtree and stores each octree in the Octree Vector.
The *DrawOctree* method takes two parameters. First one is the device context and the other is Camera object. It displays the octree in the screen converting from 3D to 2D.
The *OctreeSize* method returns how many voxels are in the octree.
The *CubePush* method pushes the each voxel in the Octree vector. All the voxel are the object Cube Class which are discussed in the Appendix B
The *RotateOctree* is the method rotate to the octree in a certain angle and display it in the screen. It takes angle, Camera, and pDC as the parameter for angle of rotation, Camera Object and device context respectively.
The *retOctree* method returns the pointer of the Octree vector.

4.6 Texture Mapping:

Texture mapping [59,60] is a powerful technique for adding realism to a computer generated scene. Adding a texture pattern to a smooth surface was first suggested by Catmul [59] as a consequence of his subdivision algorithm for curved surface. This basic idea was extend by Blinn and Newell [63] to include reflection and highlights on curved surfaces.

When mapping an image onto an object, the colour of the object at each pixel is modified by a corresponding colour from the image [62]. In general, obtaining this colour from the image conceptually requires several steps [61]. The image is normally stored as a sampled array, so a continuous image must first be constructed from the samples. Next, the image must be wraped to match any distortion (caused; perhaps by perspective) in the projected image being displayed. Then this wraped image is filled to remove high frequency components that would lead to aliasing in the final step: resampling to obtain the desired colour to apply to the pixel being textured.

There are a number of generalizations of the basic texture mapping scheme. In the case of three dimensional images, a two-dimensional slice must be selected to be mapped onto an object to boundary, since the result of rendering must be two-dimensional. Also this three-dimensional mapping requires at least two mapping as shown in Figure 4.13. One is for texture space to object space, sometimes called surface parameterization; the second is form object space to image space, i.e. the viewing transformation.

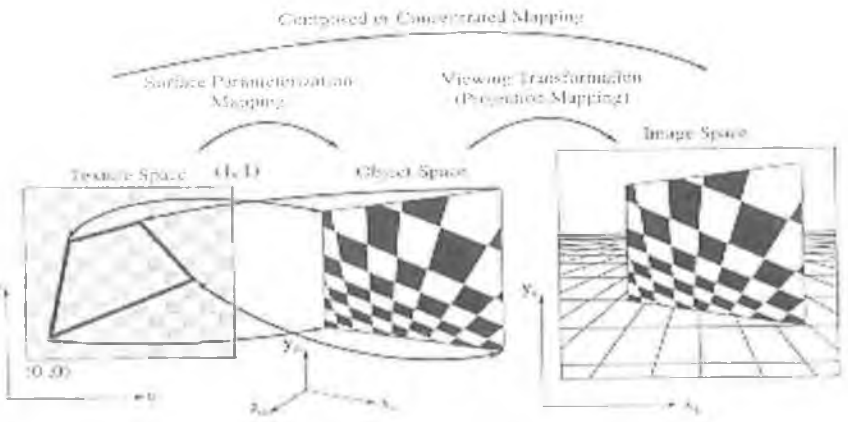

Composed or Concatenated Mapping
Surface Parameterization Mapping
Viewing Transformation (Projection Mapping)
Image Space
Texture Space
(1,1)
Object Space
(0,0)
u

Figure 4.13: Mapping from texture space to image space.

In the proposed algorithm the original image is stored in array of colour points. In the first step of texture mapping, the edge of quadtree is generated from a refined quadtree. This edge quadtree creates the edge octree. In the second step, a surface particle of the edge octree is created from the outer surface of the edge octree. Then the surface particle is projected on to the appropriate original image to create the 3D colour point for texture mapping.

Implementation of texture map requires few methods of Warnock class, CsingleView class and TextureMap class. Property WindowEdge in Warnock class is a vector to store the 2D edge point from the quadtree.
Methods of Warnock class for Texture map:
The *CenterSet* method is used to set the center of the quadtree.
The *SurfaceDetection* method in the Warnock class is used to find the edge from the quadtree.
The *EdgeLine2D* method is used create the 2D edge point from the edge quadtree using breshhem algorithm.
The *retWindowEdge* method is used to return the WindowEdge vector.
The *EdgeDraw* method draws the edge of the object.

Implementation of texture map class looks like as follows:

```
class TextureMap
{
        vector<LinePoint> linepoint;        //stores the LinePoint class object
        vector<ColorPoint> finalpoint;      //stores the ColorPoint class object
        unsigned long BackColor;            //lowest value of the background colour
        unsigned long FrontColor;           //highest value of the background colour
        double centerX;                     //x value of the image center
        double centerY;                     //y value of the image center
        double centerZ;                     //z value of the image center
public:
        TextureMap();
        TextureMap(unsigned long, unsigned long,double ,double ,double );
        BOOL CreateLinePoint(Camera,vector<EdgePoint2D>,double, double);
        BOOL CreateTexturePoint(Camera, vector<Image>, short);
        void ColorFromImage(int, int,double,int, int,double,Image *,Camera );
        void ColorFromImage(int,int,double,int,int,double,Image *, Image *, Camera,
                LinePoint *);
        int Sign(double a,double b);
        int Sign(int a, int b);
        int Sign(double a);
        BOOL ColorTest(unsigned long);
        void ClearTextureObject();
        void ShowImage(CDC* ,Camera);
        BOOL SaveTexture(char *filename);
        vector<ColorPoint> retColorPoint();
        void SetCenter(double x, double y, double z);
        void ColorPointPush(ColorPoint temp);
        BOOL RotateTexture(double angle, Camera CameraObject, CDC* pDC);
        double retCenterX();
        double retCenterY();
        double retCenterZ();
        ~TextureMap();
}; //End of the TextureMap class definition
```

The linepoint and finalpoint are two vectors array for storing LinePoint object and ColorPoint object respectively. LinePoint and ColorPoint class are discussed in Appendix B. Finalpoint vector is to holds the texture colour of the 3D object. TextureMap and TextureMap(unsigned long, unsigned long,double ,double ,double) are two constructor of the class.

The *CreateLinePoint* method takes camera object, EdgePoint2D vector and two angle as parameters and create line object and store it in the linepoint vector.
The *CreateTexturePoint* method takes camera object, image vector and image id as parameter and finds the appropriate image for 2d edge point.
The *ColorFromImage* takes ten parameters. First six parameters are for starting and ending coordinates of linepoint object. Seventh and eighth parameters are the pointer to image object. Ninth parameter is camera object and tenth parameter is the pointer

to lineobject. This method finds the 2d colour value for each point and convert it 3D colour point. And stores them in the finalcolor vector.
The *ColorTest* method takes 24 bit colour value as parameter and test, is it between the lowest colour and highest colour value of the background colour. If colour value is the background colour function returns false otherwise return true.
The *ClearTextureObject* method deletes all objects in the finalpoint vector and linepoint vector.
The *ShowImage* method displays the texture object in the screen.
The *SaveTexture* method takes filename as parameter and save the texture information in dxf file format.
The *retColorPoint* method returns the finalpoint vector.
The *ColorPointPush* method pushes the colorpoint object in the finalpoint vector.
The *RotateTexture* method takes angle, camera object and device context as parameters and rotates the object according the angle and display the texture image in the screen.

The *Texture_3D* method in the CsingleView class is responsible for synchronizing all the methods of TextureMap class and Warnock class methods.

Chapter Five

Results and Discussion

5.1 Introduction:

This chapter is devoted to the analysis and discussion of the experimental results. The experimental investigation was undertaken to observe the a) Octree model of the target object, and b) textured 3D object. A number of tests were been conducted to create the Octree model and the -Texture 3D object to find the surface model of different object. Comparisons of the real object dimensions and constructed 3D object dimensions have been carried out for each model.

Section 5.2 describes the 3D Model of a cup. Section 5.3 shows the 3D Model of a bottle. Section 5.4 describes the 3D Model of a Mechanical test specimen. And Section 5.5 shows 3D Models of few other objects.

5.2 3D Model of a Cup:

A white plastic cup is placed on the turntable with a green background color to green. 36 pictures were captured with the calibrated camera by rotating the cup at 10 degree intervals. Figure 5.1 shows the photograph of the object. Using the *3D Model* software an octree model was created from the 36 images, and this is shown in Figure 5.2a. Figure 5.2b shows the created octree model after a 10 degree rotation. The texture map of the object was also created from the 36 images and is shown in Figure 5.3a. Figure 5.3 b,c show the texture map of the 3D object after ten and twenty degree rotations respectively. Figure 5.4 shows the object published using AutoCAD package.

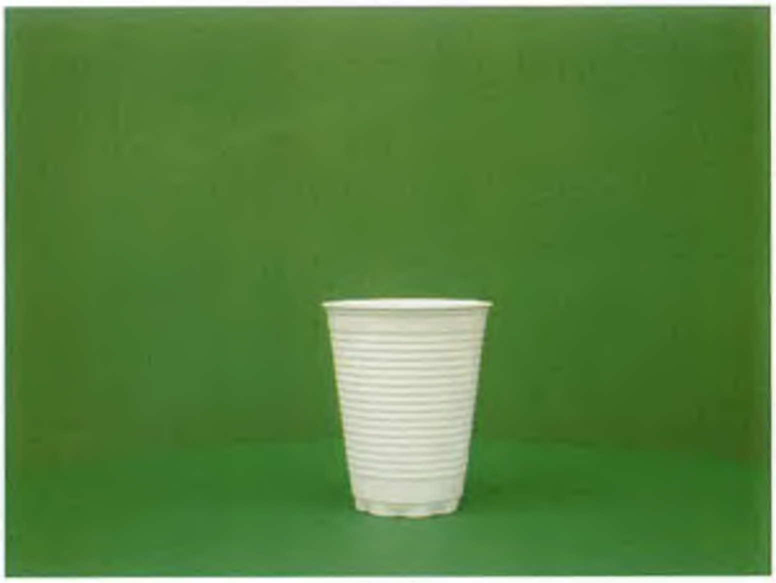
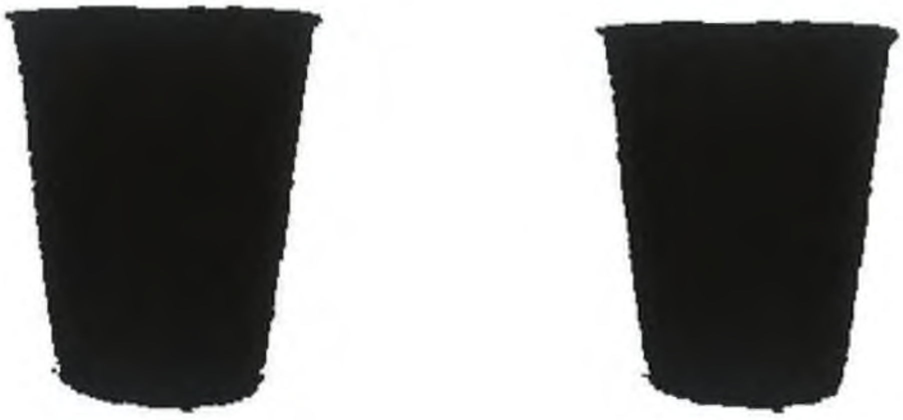


Figure 5.1: shows the photograph of the object



(a) (b)

Figure 5.2: (a) is the octree of the object at initial position (b) is the octree of the object after ten degree rotation.

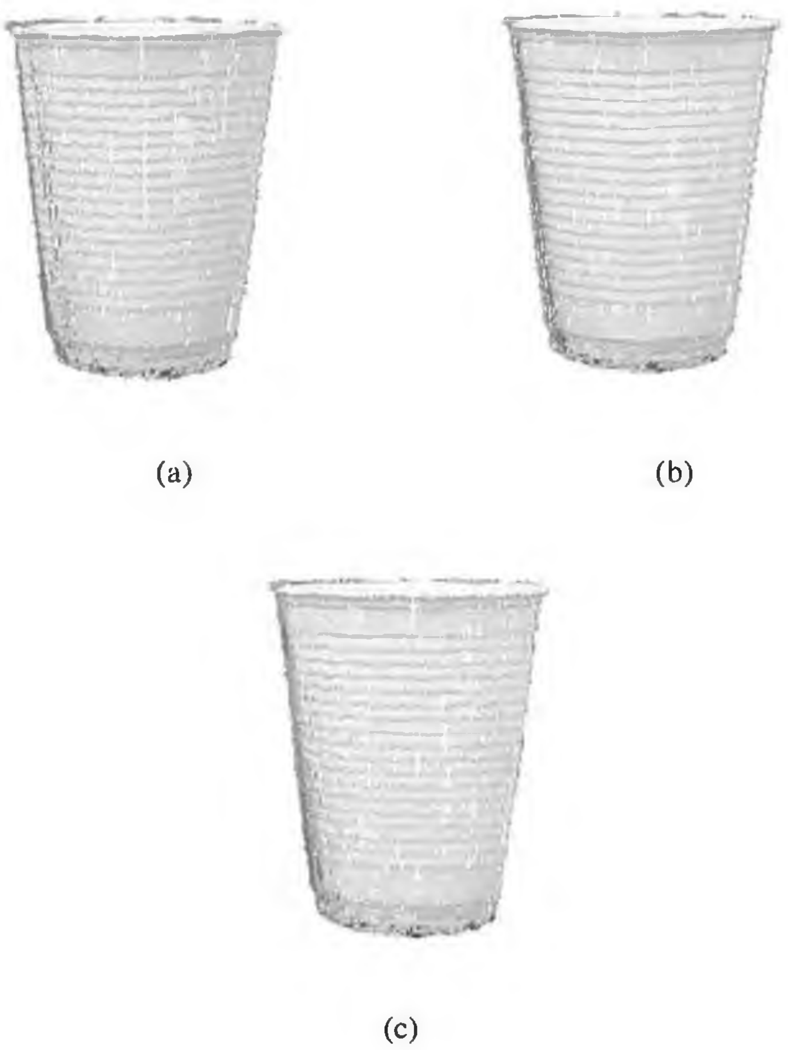
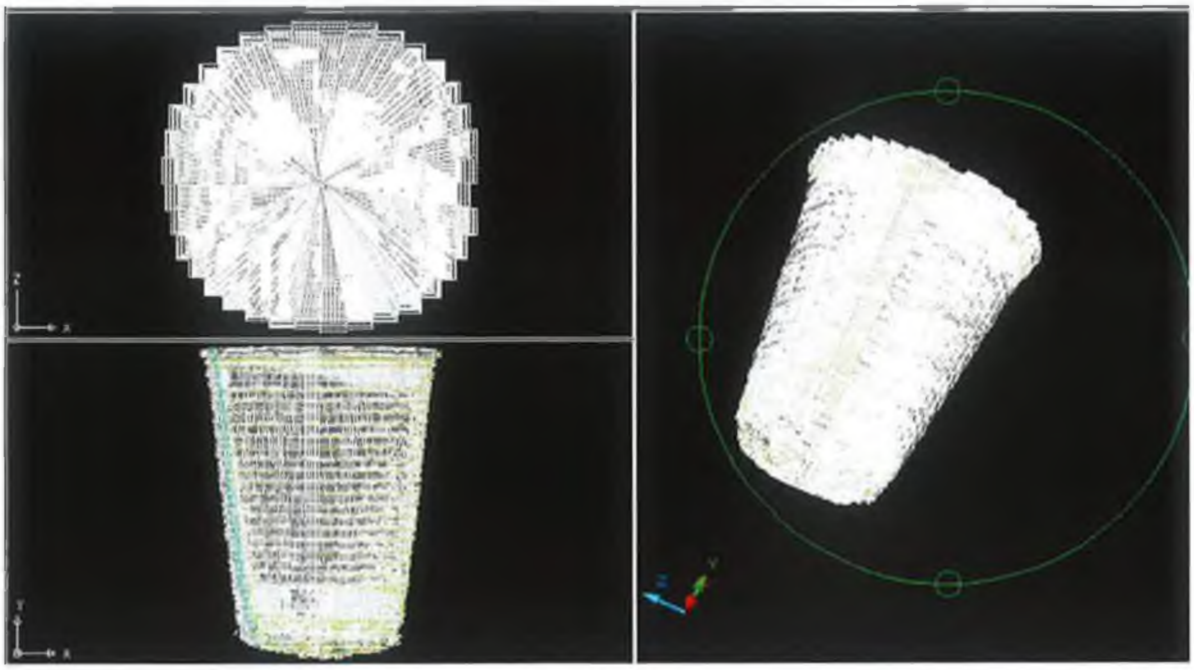
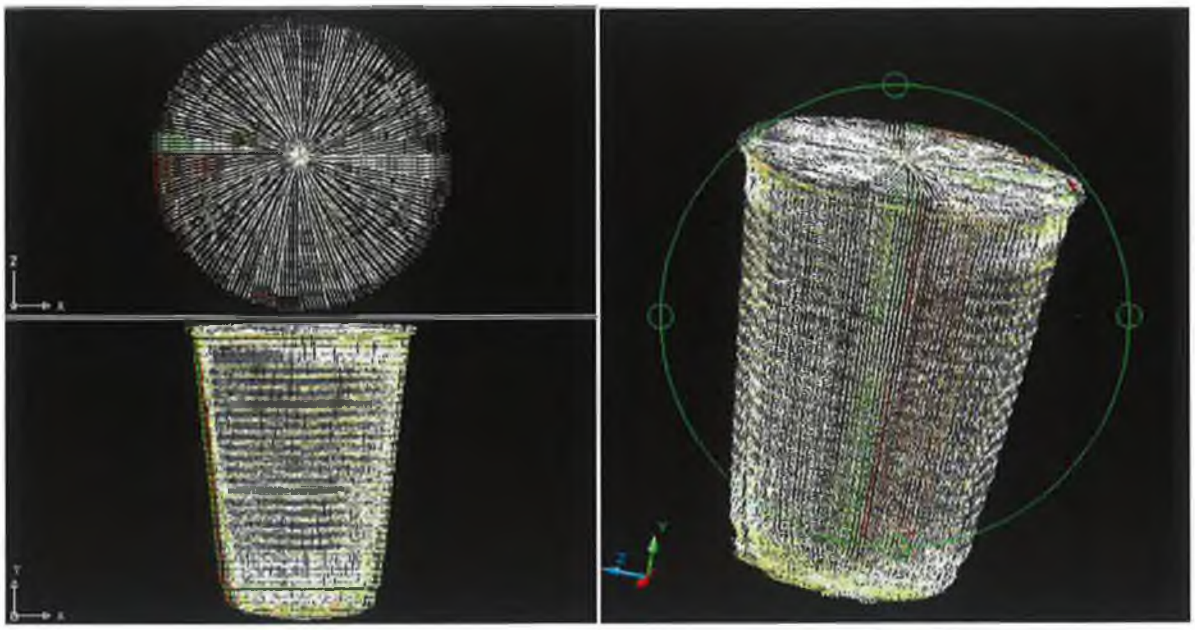


Figure 5.3: (a) is the texture map object at initial position, (b) and (c) are the texture map object after ten and twenty degree rotation.



(a)



(b)

Figure 5.4: (a) Top, Front and 3D view of the Octree object in AutoCAD and (b) Top, Front and 3D view of surface of the object.

The 3D Model software stores the dimension of the cup in the "inputfile.txt" binary file in order to compare the created 3D data with the real object dimensions. Table 5.1 shows the comparison of experimental data with the real cup dimensions.

Table 5.1: Comparison of the Experimental data with real dimensions of the Cup

Cup data		Experimental data			
Height (mm)	Width (mm)	Height (mm)	Left (mm)	Right (mm)	Width (mm)
0	70.00	-0.650606	-15.306	55.9925	71.2985
10.00	64.80	-10.8586	-13.2195	53.0237	66.2432
20.00	61.30	-20.0721	-11.6592	51.5861	63.2453
30.00	58.30	-30.3473	-10.1074	50.6753	60.7827
40.00	56.35	-40.4678	-9.07835	49.2513	58.32965
50.00	53.60	-51.0586	-7.54836	48.3419	55.89026
60.00	51.35	-60.9909	-6.53892	46.9091	53.44802
70.00	49.10	-70.4628	-5.02108	45.9842	51.00528
80.00	46.80	-81.6083	-3.51262	44.5557	48.06832
87.60	41.80	-90.7318	-1.48097	41.5358	43.01677
-	-	-92.8479	1.10035	39.4763	38.37595

From above data, it is obvious that 3D software creates the surface of the 3D object from the 2D images. The top view in the Figure 5.4 shows the stair impact of the constructed object due to the octree algorithm. Reducing the rotation angle of the object less than 5 degree during experiment this effect can be removed. Figure 5.4b shows the surface of the textured object with few errors.

As the world coordinate centre is above of the object, Table 5.1 shows the object height in negative values. It starts from -0.650606mm below the world coordinate centre and ends at -92.8479mm. The height of the object is 87mm but experimental data shows 92.197294mm. The margin of error was with an agreement within 6%. These errors were during the experiment for two reasons, one is for elliptical shapes at the top and bottom of the silhouette object, and the other is during the calibration rail replacement with the turntable. If the position of the object is not in the same place as the calibration rail than image dimension will be little different. The constructed 3D image will therefore show greater dimensions than the real object if the object is nearer to the camera than the calibration rail. The object dimension will be smaller, if it is placed further away from the camera than the calibration rail position. Calibration data for the Object is given in Appendix E.

The Octree of the cup shows some errors at the bottom of the cup, which is generated from the silhouette error. The silhouette error can be generated because of spotlights and shadows. Silhouette errors also create distortion in the texture mapping which can be visible in the bottom of the texture map images. Precaution should be taken to avoid the spotlights and shadows.

5.3 3D Model of a small Bottle:

A white coloured plastic bottle was placed on the center of the turntable with the background color set to green. 36 pictures were captured with the calibrated camera by rotating the bottle in 10 degree interval.

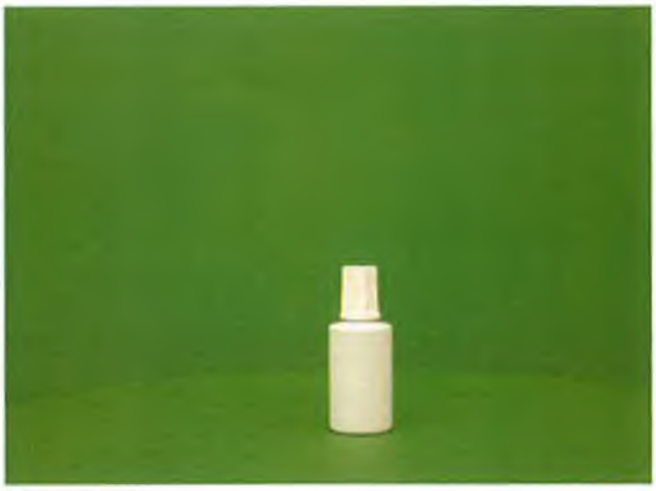


Figure 5.5 shows the photograph of the object.

Figure 5.5 shows the object in the initial condition. Figure 5.6a shows the octree model of the object which is created using the 36 images. Figure 5.6b and Figure 5.6c show the octree of the object for ten and twenty degree rotations respectively. Figure 5.7a shows the textured map 3D object created from the 36 images. Figure 5.7b and Figure 5.7c shows texture map 3D object after ten and twenty degree rotations respectively. Figure 5.8 shows the AutoCad view of the bottle. And Table 5.2 shows the comparison of experimental data with the real object dimensions.

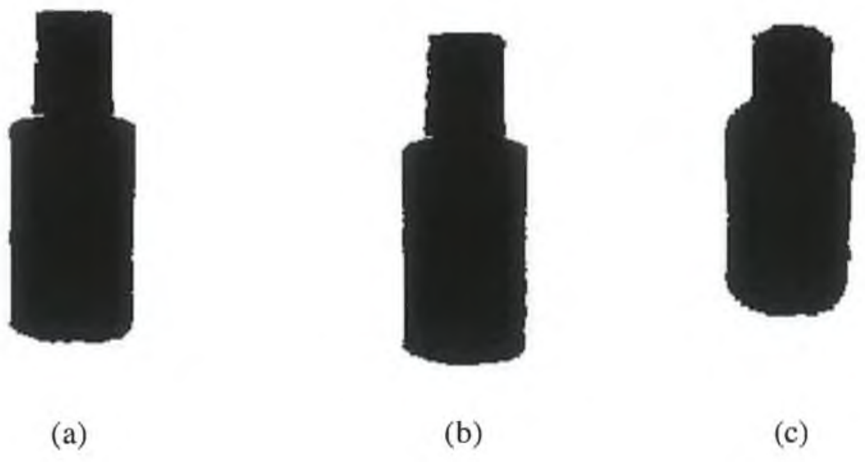


Figure 5.6: (a) is the octree model of the object at initial condition, (b) is the octree of the object after twenty degree rotations and (c) is the octree of the object after twenty degree rotation in X direction.

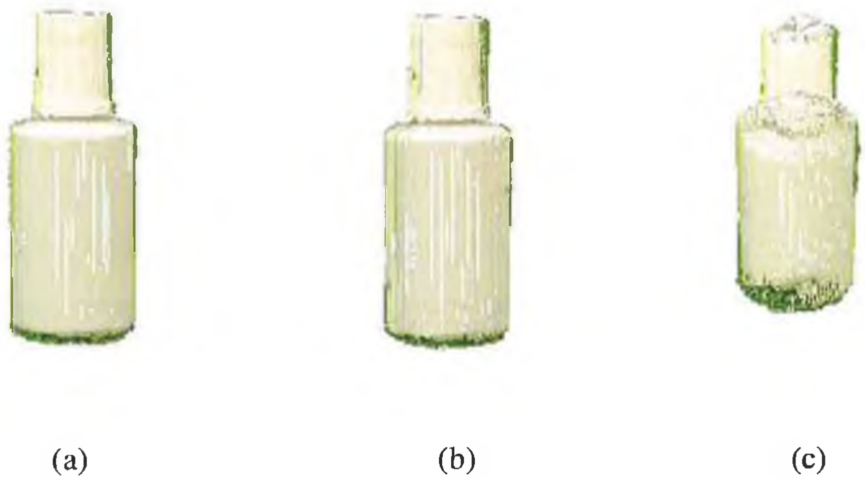
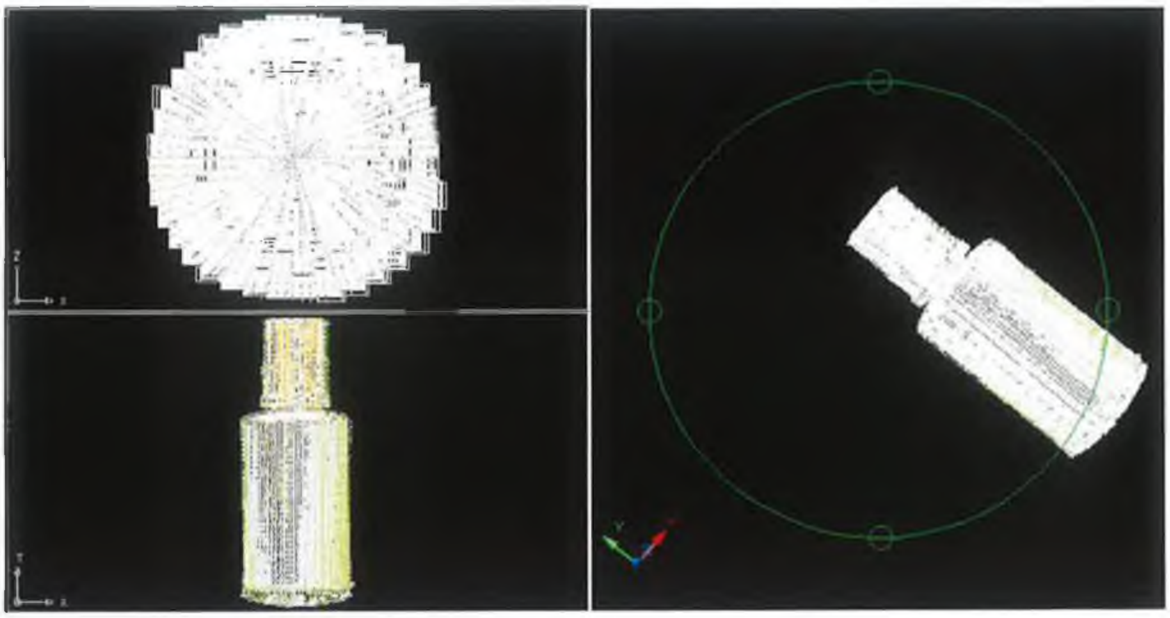


Figure 5.7: (a) is the texture map model of the object at initial condition, (b) is the texture map of the object after twenty degree rotations and (c) is the texture map of the object after twenty degree rotation in X direction.


(a)

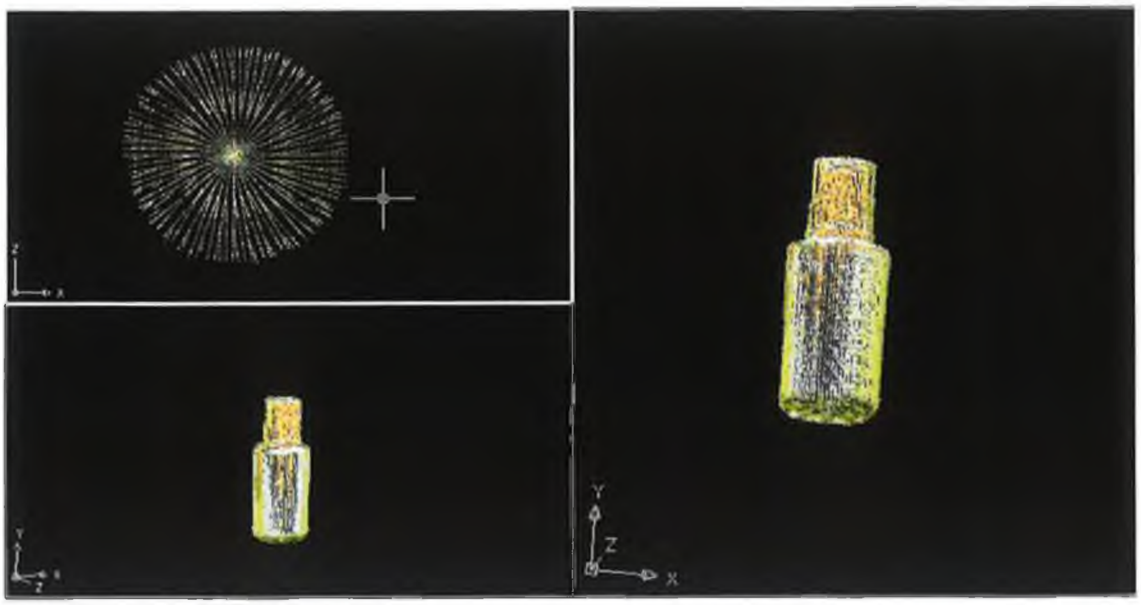

(a)

Figure 5.8 Octree Top, Front and 3D view in AutoCAD and (b) Surface Top, Front and 3D view in AutoCAD

Table 5.2: Comparison of the Experimental data with real dimensions of the Bottle

Object data		**Experimental data**			
Height (mm)	Width (mm)	Height (mm)	Left (mm)	Right (mm)	Width (mm)
0.0	17.1	-6.69897	9.84355	26.4435	16.59995
10.00	17.25	-16.0213	9.91808	27.0217	17.10362
20.00	18.15	-26.3562	9.51423	27.6168	18.10257
24.20	19.10	-30.3082	8.56867	28.6388	20.07013
26.20	11.30	-32.7482	12.5048	25.2358	12.731
29.25	30.20	- 34.7865	4.19961	33.5924	29.39279
39.25	30.20	-44.2034	3.29261	34.1989	30.90629
49.25	30.20	-54.1497	3.36997	34.3283	30.95833
59.25	30.20	-64.6418	3.94486	34.4724	30.52754
69.25	30.20	-74.6946	4.02632	34.6174	30.59108
79.25	30.20	-84.8084	4.10944	34.7702	30.66076
82.25	29.20	-87.3427	4.625	33.3231	28.6981

From Figures 5.6-5.8 show the 3D reconstruction of the object by the *3D Model* software. The texture map view of the object shows some error at the bottom of the 3D view. The camera sensor creates the error during image capturing (Figure 5.9). As a result the texture map 3D image has the background colour in the bottom of the constructed 3D image.

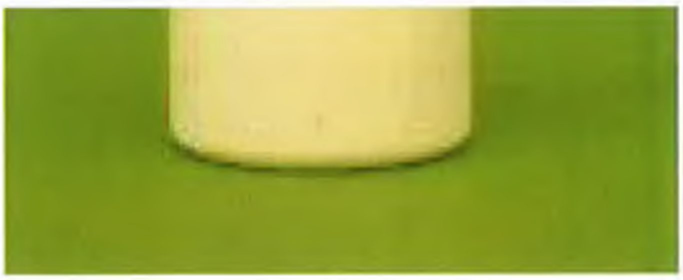
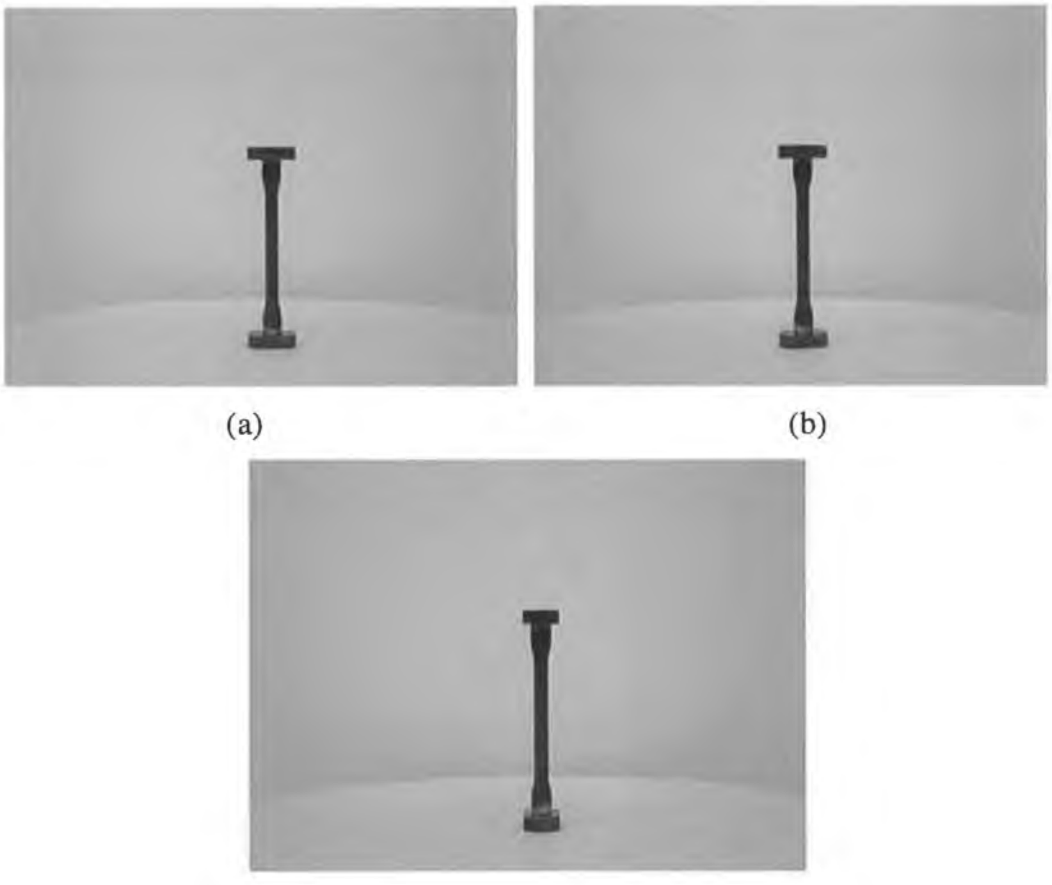


Figure 5.9: Color Error in the Object image

All the height values are negative in Table 5.2 because the object height did not cross the World coordinate centre that is set during the Calibration procedure. The reconstructed Object height is 80.64373mm that is very near to the real object height (82.25mm). Table 5.2 also shows that widths of the object at different heights are very close to the Real object widths. The margin of error was with an agreement within 2%.

5.4 3D Model of a Mechanical Test Specimen:

A copper Tension-Torsion specimen was placed at the centre of the turntable with setting the background colour set to white. 36 Images were taken at 10 degree intervals for this object. Figure 5.10 shows the image of the specimen at different angle. Figure 5.11 shows the octree model of the object in different view angle. Textured map 3D images are shown in the Figure 5.12. AutoCAD view of the object is shown in Figure 5.13. Table 5.3 shows the comparison of the object data with the experimental data for the object position shown by the Figure 5.10a.



(a) (b)

(c)

Figure 5.10 (a), (b), and (c) images show the specimen in three different angles.

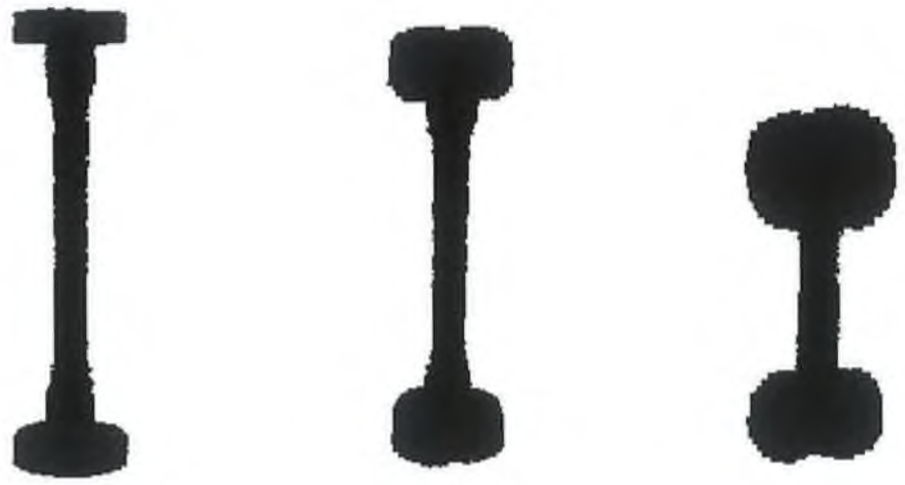


Figure 5.11: Octree of the object in different view angle.

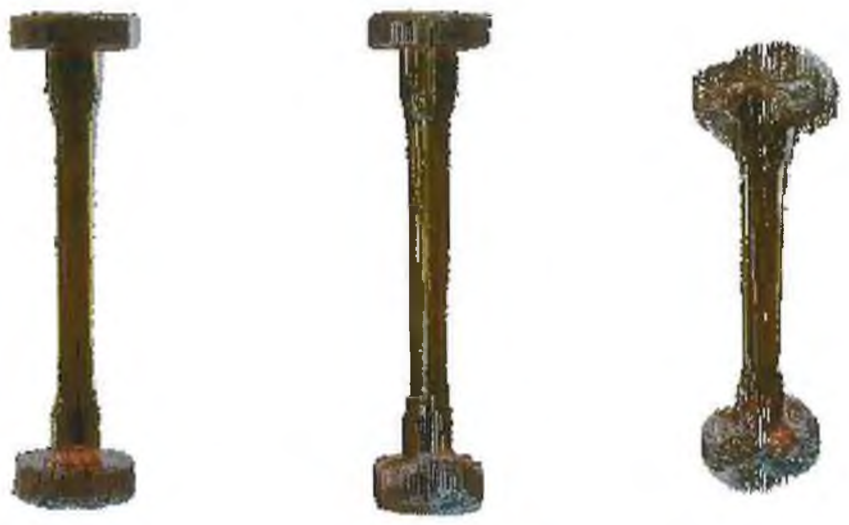
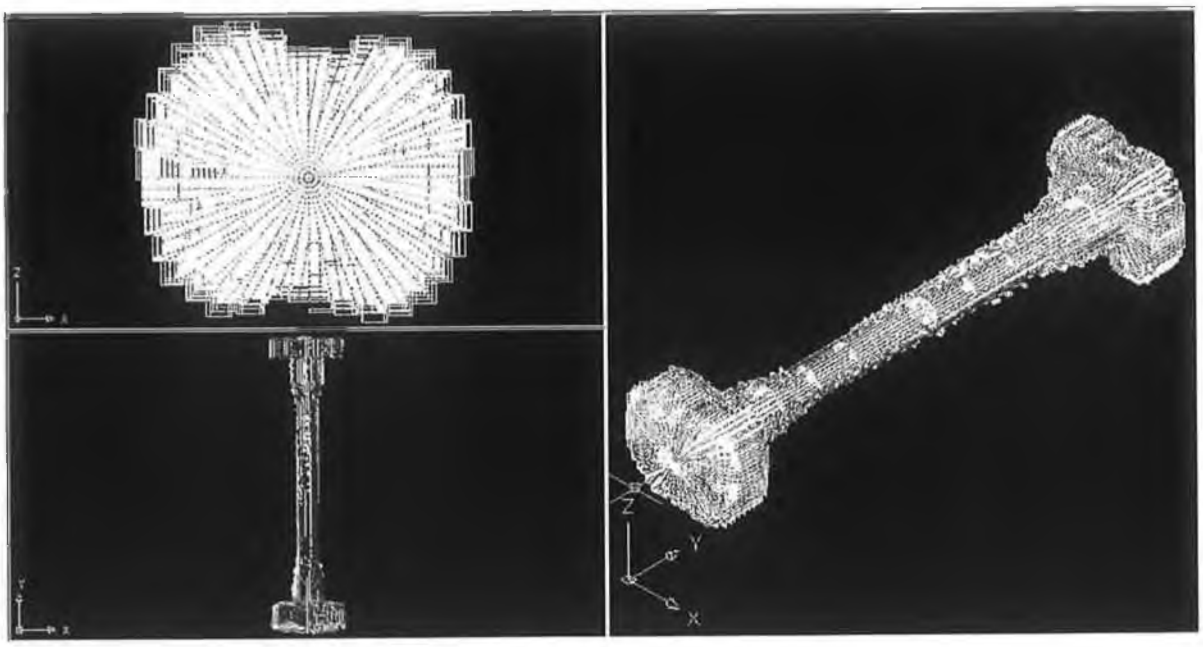
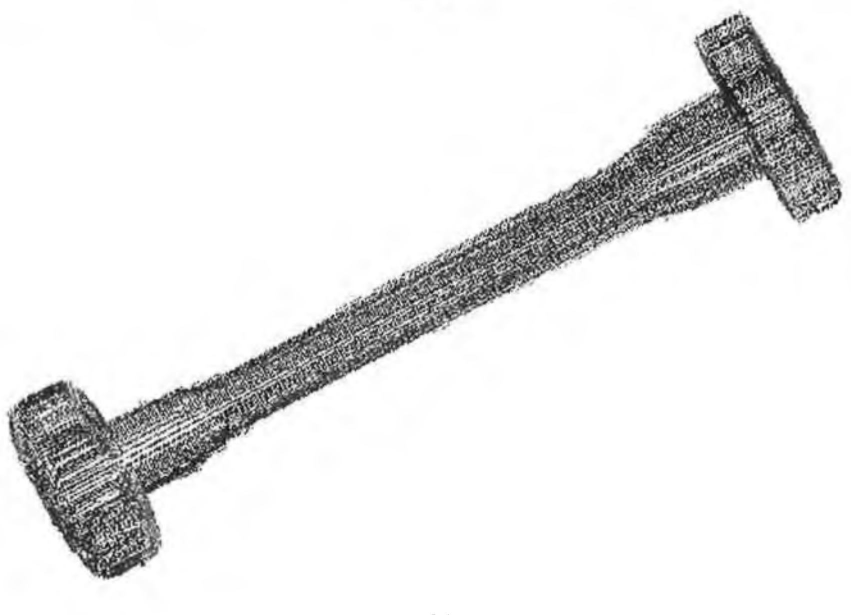

(a) (b) (c)

Figure 5.12: (a), (b), and (c) images show the textured object in different view angle.



(a)



(b)

Figure 5.13: (a) Octree Top, front and 3D view in AutoCAD, (b) Surface 3D view in CAD

Table 5.3 Comparison of the experimental data with real dimensions of the Specimen

Object data		**Experimental data**			
Height (mm)	Width (mm)	Height (mm)	Left (mm)	Right (mm)	Width (mm)
0	29.8	36.3489	-11.7064	19.0042	30.7106
10	11.8	26.7988	-2.58187	9.50046	12.08233
20	11.8	16.7653	-2.00856	9.57244	11.53296
30	8.00	6.06922	-0.422822	7.63579	8.058612
40	8.00	-6.08479	-0.334524	7.72801	8.062534
50	8.00	-16.2247	-0.25918	7.80757	8.06675
60	8.00	-26.3786	-0.182209	7.88963	8.071839
70	8.00	-36.0375	0.397228	8.47485	8.077622
80	8.00	-46.2283	0.47787	8.56228	8.08441
90	8.00	-56.4428	0.560269	9.15819	8.597921
100	11.8	-66.1181	-0.878361	9.24671	10.12507
110	11.8	-76.9696	-1.29676	10.8684	12.16516
120	29.8	-86.1203	-10.3524	20.6061	30.9585
122	29.8	-88.1856	-10.3376	20.6291	30.9667
Total: 122 mm		124.5345 mm			

From Figures 5.11-5.13, it is clear that reconstructed 3D Object looks similar to the original Object. But it has some errors due to the following reasons:

a) At the time of the rotation of the object on the turntable it creates a circular motion that is clearly visible in Top view of the object in Figure 5.13. Circular motion is created from 10 degree to 80 degree and from 100 to 170 degree rotation of the object.
b) Figure 5.14 shows that constructed 3D model has some error in the edge of the model. This is due to the error in the silhouette and it is shown in Figure 5.15.

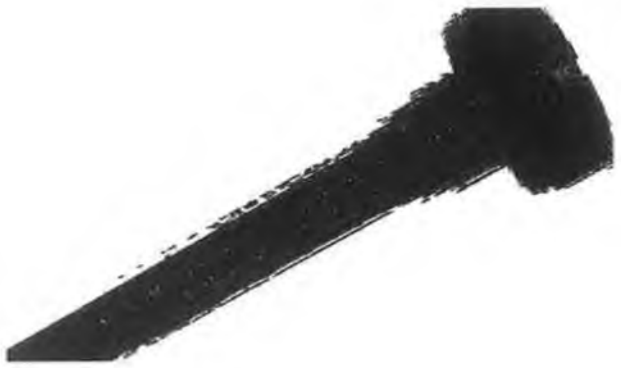
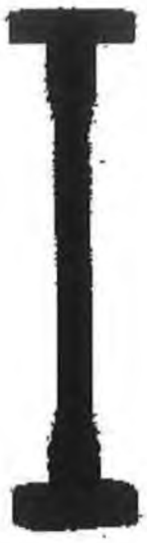


Figure 5.14: Error in the 3D object



5.15: Error in Silhouette edge

Table 5.3 shows the comparison of the object data with experimental data. Experimental data shows height of the object 124.53mm which is very near to the real object height (122mm). The margin of error was with an agreement within 2%.

5.5 3D Models of a few other Objects:

In this section, three-dimensional models of a Christmas tree, a cup, and two different dolls were presented. All the Models were created using the *3D Model* software.

Figure 5.16 shows the image of the Christmas tree and Figures 5.17, 5.18, 5.19 and 5.20 show its 3D view, octree views, texture map views and AutoCAD view respectively.

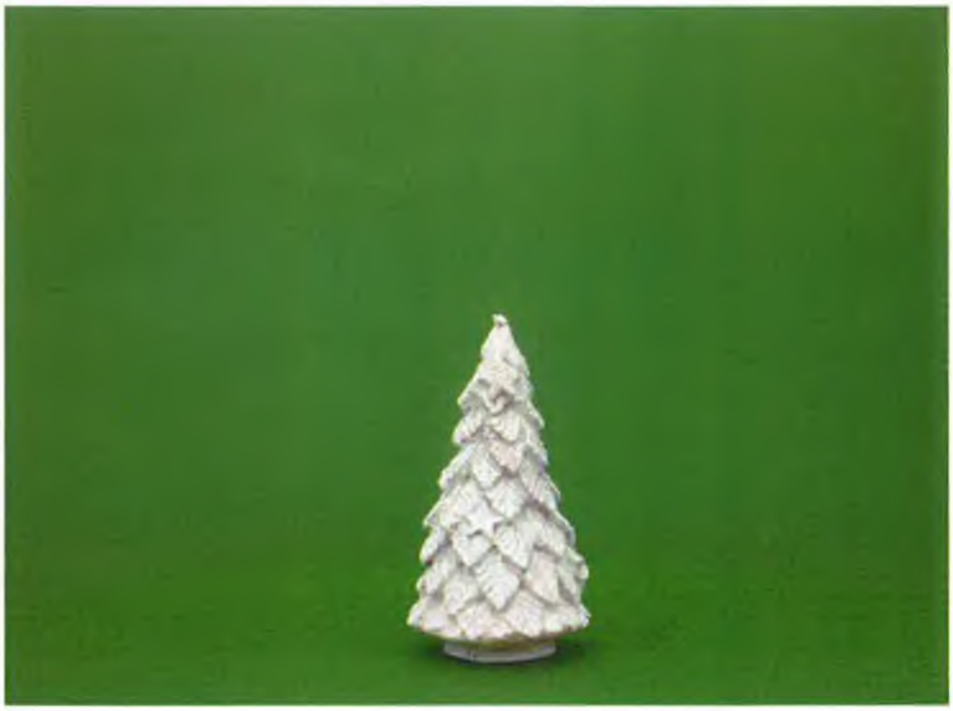


Figure 5.16: 2D image of a Christmas tree.

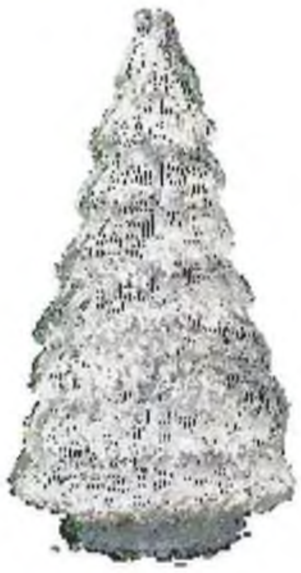


Figure 5.17: Constructed 3D Model of the Christmas tree.

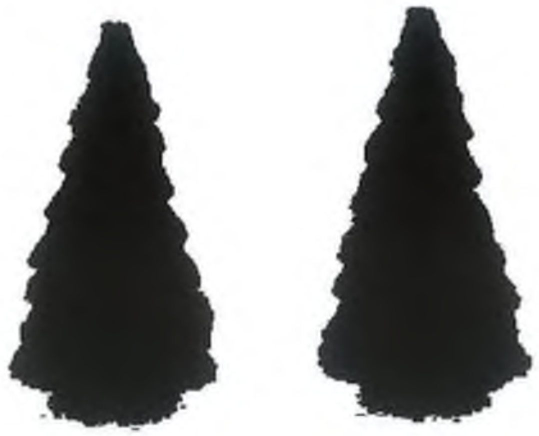


Figure 5.18: Octree Models of the Tree in Different view angle.

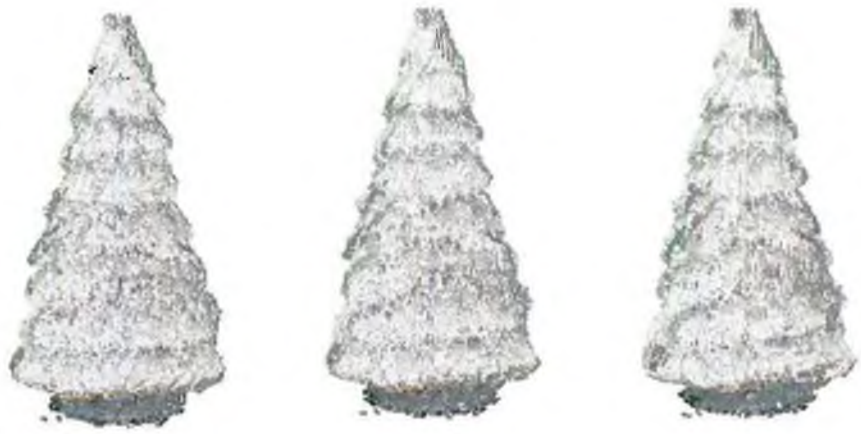


Figure 5.19: Texture Map trees in three different View angle

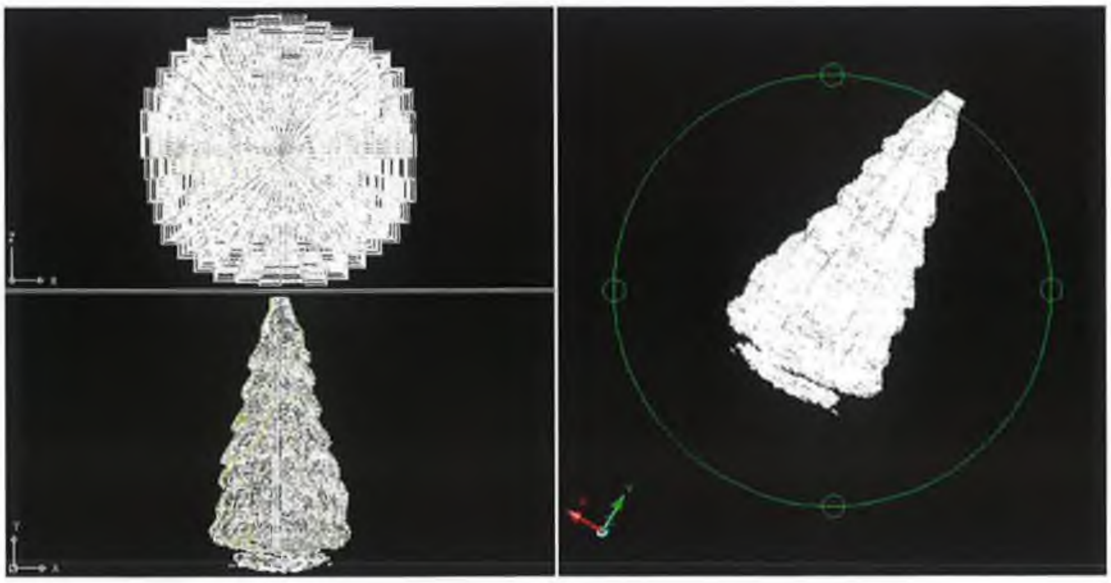


Figure 5.20: Top, Front and 3D view of the Tree using AutoCAD

With very few errors the *3D Model* software created the 3D model of the Christmas tree from 36 images (10 degree interval rotation).

Figure 5.21 shows the image of a cup and Figures 5.22 - 5.25 show its 3D view, octree views, texture map views and AutoCAD view respectively.

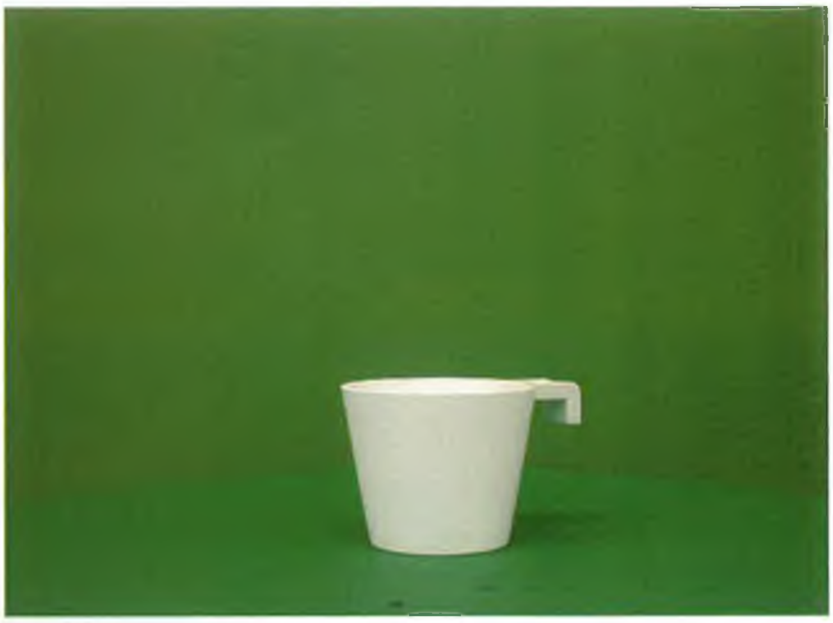


Figure 5.21: 2D image of a cup

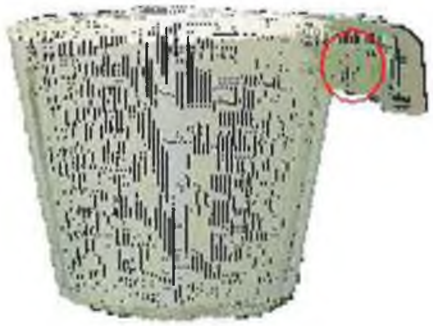


Figure 5.22: 3D view of the object with error in the handle

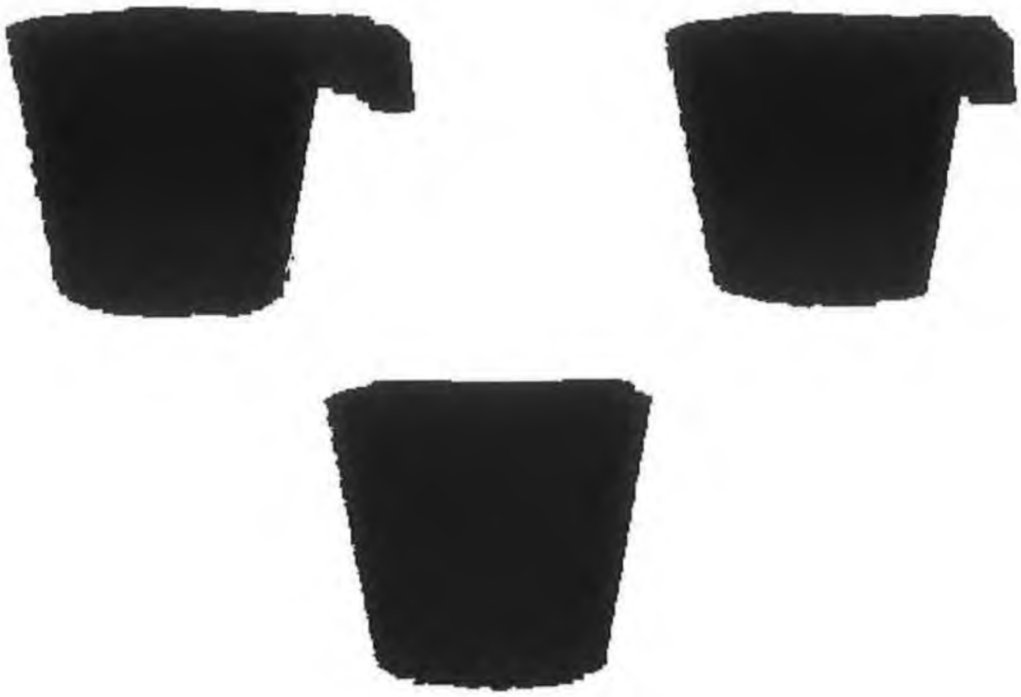


Figure 5.23: Octree of the cup in different view angles

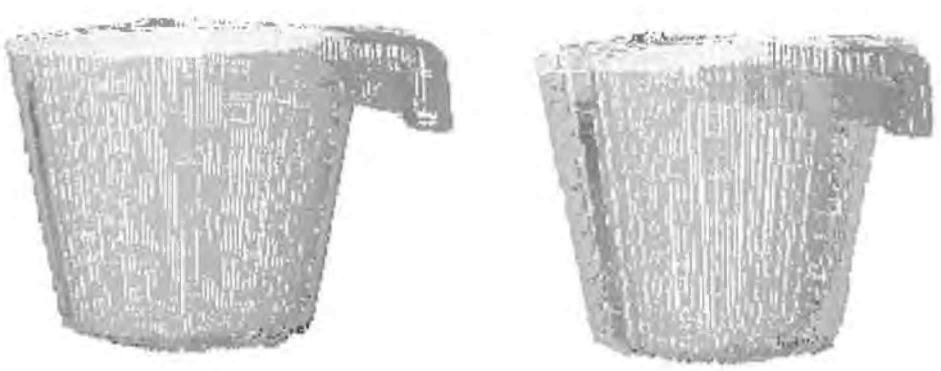


Figure 5.24: Texture object in different view angle

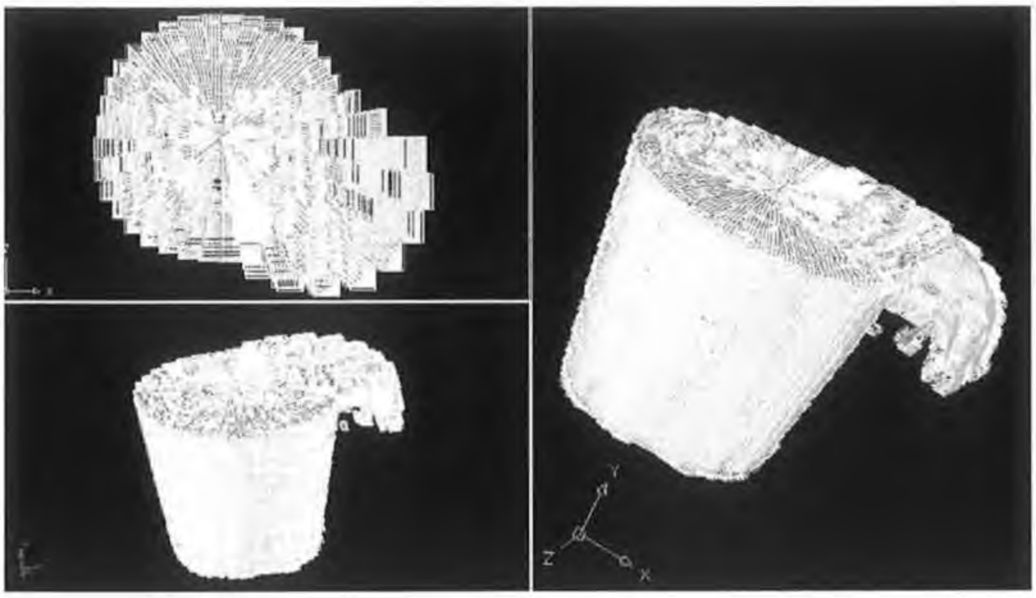


Figure 5.25: Top, Front and 3D view of the Cup by AutoCAD

The 3D view of the cup shows some error in the handle, which marked in the Figure- 5.22. And it is clearly visible in the CAD view (Figure 5.25) of the cup. This error occurred due to the circular motion of the handle during the rotation of the cup. And Figure 5.26 shows this error using silhouettes of the cup in different view points. This error can be removed using track point on the silhouette over three or more frames.

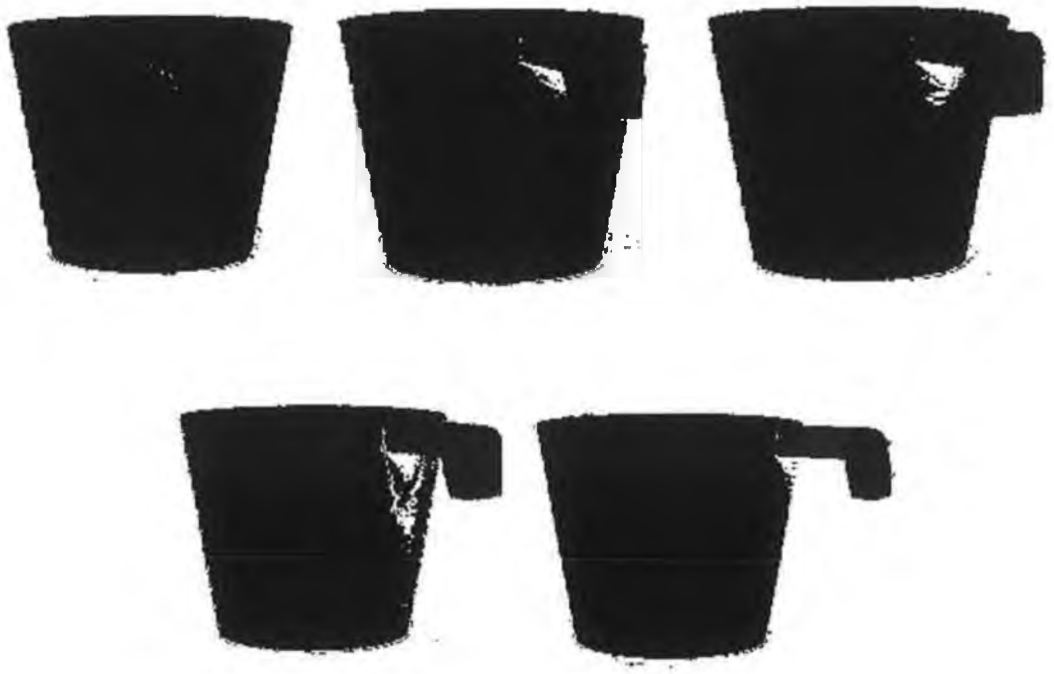


Figure 5.26: Circular motion of cup handle

Figure 5.27 shows the original photographic image of a doll and Figure 5.28 shows its 3D model created by the *3D Model* software from its 36 images of 5 degrees rotation view. Figure 5.29 shows Octrees of the object in different view angle and Figure 5.30 shows its Texture map views in different angles. Figure 5.31 shows the AutoCAD view of the doll.

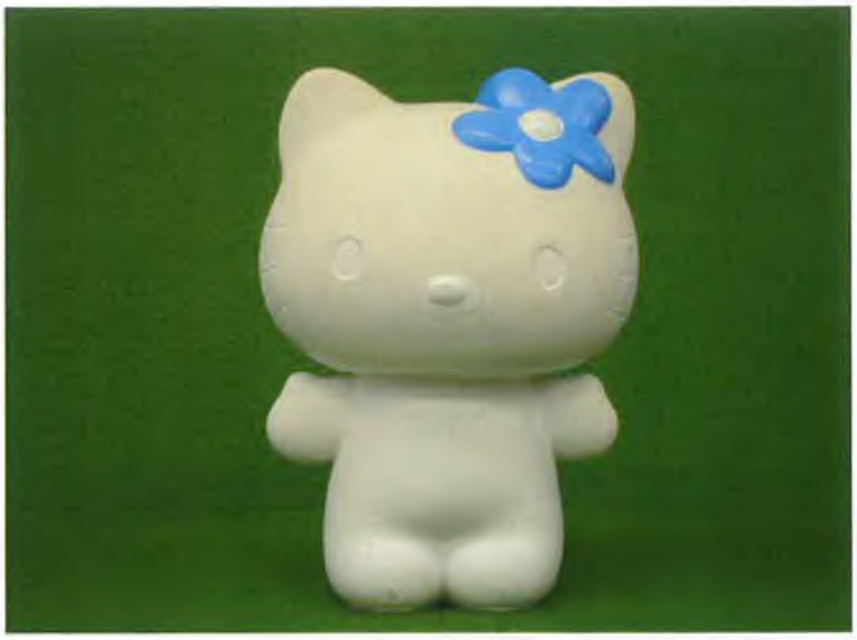


Figure 5.27: 2D image of a Doll

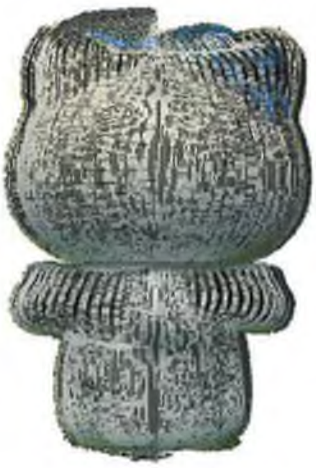


Figure 5.28: 3D model of the Doll

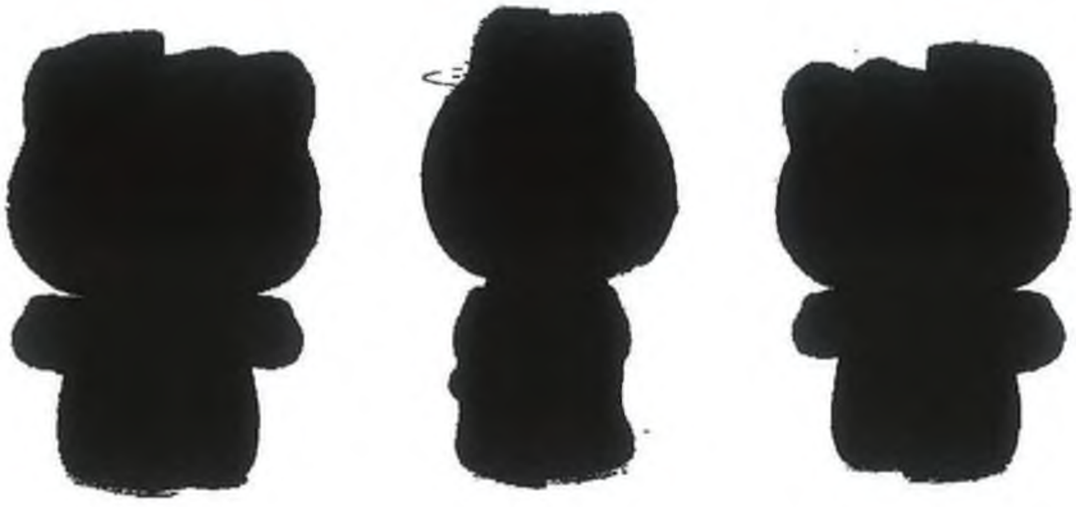


Figure5.29: Octree views of the Doll different view angles

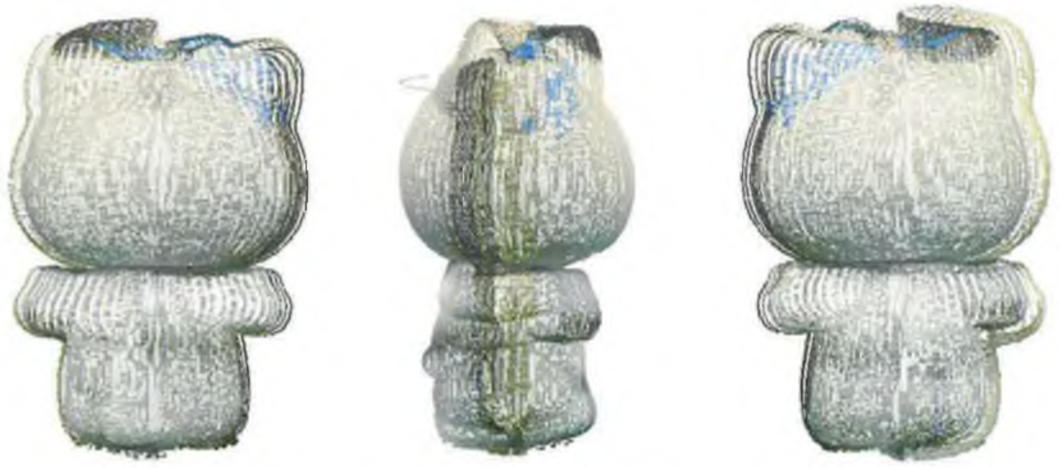


Figure 5.30: Texture Map views of the Doll in different angle

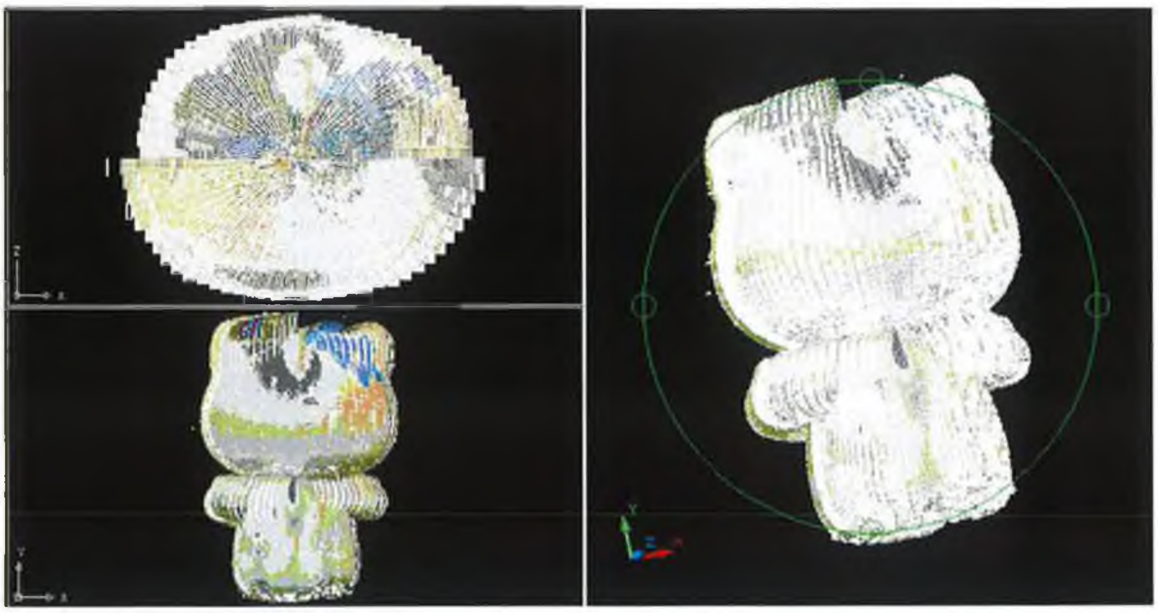


Figure 5.31: AutoCAD produced view of the Doll

Figure 5.32 shows the original photographic image of another doll and Figure 5.33 shows its 3D model created by the *3D Model* software from its 36 images of 5 degrees rotation view. Figure 5.34 shows Octrees of the object in different view angle and Figure 5.35 show its Texture map views in different angles. Figure 5.36 shows the AutoCAD view of the doll

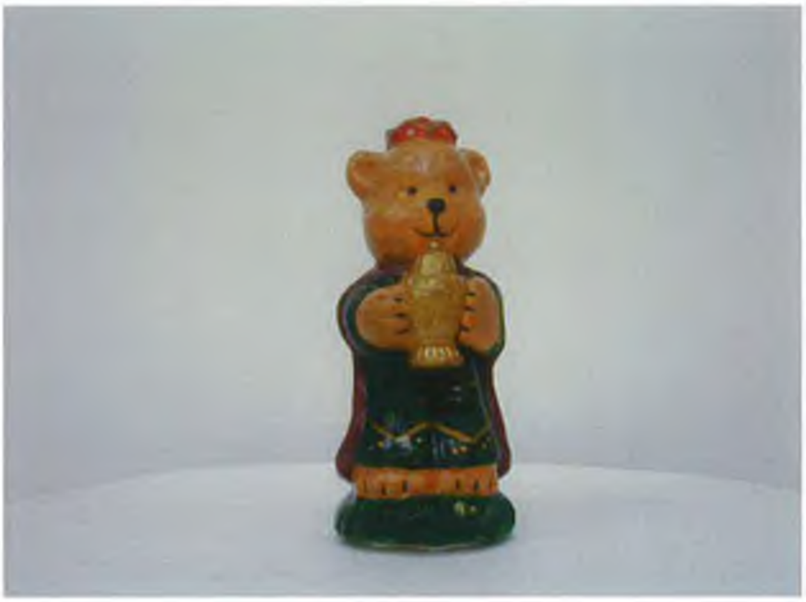


Figure 5.32: 2D image a Doll

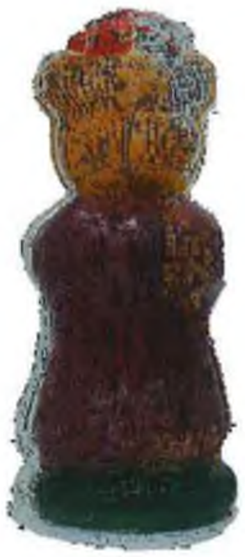


Figure 5.33: 3D Model of the Object

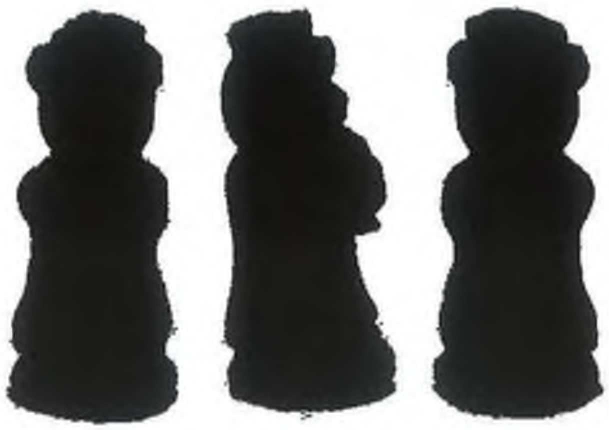


Figure 5.34: Octree Models of the Object in Different View angle

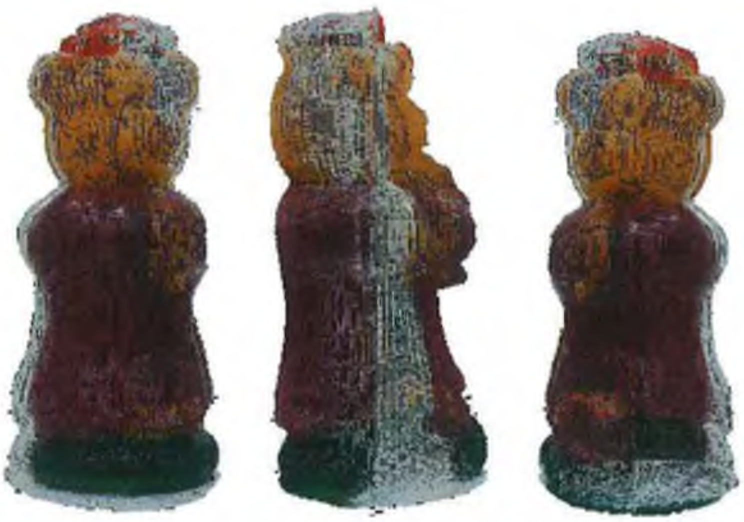


Figure 5.35: Texture Map objects in different view angle

Chapter Six
Conclusions and Suggested Future Work

6.1 Conclusions:

3D Model is a good user friendly software that was developed by the author as a part of the research project to create 3D surface model from a series of 2D digital pictures solid objects. The images of the target object are captured by a digital camera from different view points rotating the object on a turntable. The developed algorithm creates quadtrees from silhouette images and an Octree is created from refined quadtrees. It also generates surface information from quadtrees. The surface information is used to create a texture map of the 3D objects. For most objects without concavities, a fairly good approximation of the surface of the objects can be obtained from multiple 2D images.

In the developed algorithm, Quadtree and Octrees are chosen for describing 2-D silhouette and 3-D object because of two reasons. The first one is, they possess the property of data compression and retain the detailed boundary/surface information as well, and the other is, the spatial information is implicitly encoded in the quadtree and octree structures.

The proposed system still has the same limitations as previous silhouette based volumetric construction algorithms, such as limited precision and the inability to detect concavities in the object. To be truly useful, the proposed method will have to be combined with a more sophisticated shape from motion algorithms that track surface marking on the object. Using a tracking point on the silhouette over three or more frames the location and curvature of points on the surface of the object can be determined. As part of such a shape from rotation system, it can quickly and automatically provide 3D shape description of real objects for modeling applications.

Despite a few limitations in the *3D Model* software, it can however be used to create 3D surface model from a series of 2D digital pictures of moderately complex objects.

6.2 Thesis Contribution:

- In the system developed one needs to calibrate the camera only once to create 3D model from 2D images.
- The SUSAN corner detection algorithm [22,64] has been modified to obtain accurate corner from the calibration pattern.
- The Tasi [55] C source code for Camera Calibration has been modified using object oriented approach.
- The developed system permits most of the unwanted surface points to be automatically detected and eliminated at the time of surface construction.

6.3 Suggested Future Work:

- Mesh Object can be created from the Octree
- Texture mapping quality can be improved using Mesh.
- Calibration technique can be improved by using new corner detection algorithm.

References

1. Jiang Yu Zheng, "Acquiring 3-D models from sequences of contoures", IEEE Trans. On Pattren Ana. And Mach. Int." vol. 16, no. 2. February 1994.
2. W. Brent Seals. "Building three-dimensional object models from image sequences". Computer vision and image understanding, vol: 61, No: 3, pp 308-324, 1995.
3. Regis Vaillant and oliver D. Faugeras, "using external boundaries for 3-D object modelling". IEEE transection on PAMI, vol: 14, no: 2, February 1992.
4. Peter Giblin, "Reconstruction of Surfaces from profiles", in Proceeding 1st International Conference on Computer Vision, 1987, pp. 136-144.
5. F. Schmit and Y. Yemez[1999], " 3D color object reconstruction from 2D image sequences", Image Processing, 1999. ICIP 99. Proceedings. 1999 International Conference on , Volume: 3 , 1999 Page(s): 65 -69 vol.3.
6. Y. Matsumoto, H. Terasaki, K. Sugimoto and T. Arakawa, "A portable three dimensional digitizer," Int. Conf. On Recent Advances in 3D Imaging and Modeling, pp. 197-205, Ottowa, 1997.
7. W.E. Lorensen, H.E. Cline, "Marching Cubes: A high resolution 3D surface Construction Algorithm", ACM Computer Graphics, vol-21, no-4, pp. 163-169,1987.
8. W. J. Schroeder, J.A. Zarge, and W.E. Lorensen, "Decimation of triangle mesh", ACM, Computer Graphics (Siggraph'92 Proc.), Vol.26, pp.65-70,1992.
9. Takahashi and F. Tomita, "Camera calibration for robot vision," Journal Robotics Soc. Japan, 10(2): 177-184, 1992.
10. S. Fortune, Voronoi diagrams and Delaunay triangulations, Computing in Euclidean Geometry, pp. 193-233, World Scientific, 1992.
11. Niem and J. Wingbermuhle, "Automatic reconstruction of 3D objects using a mobile monoscopic camera", Int. Conf. On Recent Advances in 3D Imaging and Modeling, pp. 173-181, Ottowa, 1997.
12. Roger Tsai [1986], "An efficient and Accurate Camera Calibration Technique for 3D Machine Vision", *Proceedings of IEEE Coonference on Computer Vision and Pattern Recognition*,Miami Beach, FL, 1986, pp.364-374.

13. W. Niem, H. Broszio, "Mapping texture from multiple camera views onto-3D object models for computer animation", Proceeding of the international workshop on stereoscopic and three dimensional imaging, September 6-8, Santorini, Greece.
14. Worthy N. Martin and J.K. Aggarwal[1983], "Volumetric description of objects from multiple views", IEEE transactions on Pattern analysis and Machine Intelligence -5, no-2, march 1983.
15. Thompson and Mundy, "3D model matching from an unconstrained viewpoint", proceedings IEEE Conference on Robotics and Automation, 1987.
16. Jiang Yu Zheng and Fumio Kishino, "3D models from contours: further identification of unexposed areas", In Proc. 11th ICPR, vol. 1, pp. 349-353, Sept. 1992.
17. Chuin-Hong Chein and J.K. Aggarwal, "Model construction and Shape recognition from occluding contour", IEEE , PAMI, vol-11, no-4, April, 1989.
18. H. Samet and A. Rosenfeld, "Quadtree structures for region representation," in Proc. IJCPR, 1980, pp. 36-41.
19. C.H. Chein and J.K. Aggarwal "A normalized quadtree representation," Computer Vision, Graphics, Image Processing, vol .26, pp. 331-346,1984.
20. C. H. Chein and Aggarwal, "Computation of volume/surface octrees from contours and silhouette of multiple views," in Proc. CVPR-86, Miami Beach, FL, June 22-26, 1986, pp250-255.
21. C. H. Chein and Aggarwal, "Identification of 3D objects from multiple silhouettes using quadtrees/octrees", Computer Vision, Graphics, Image Processing, 36, 256-273, 1986.
22. C.H. Chein and J.K. Aggarwal, "Volume /surface octrees of the representation of 3D objects," Computer, Vision, Graphics, Image Processing, vol. 36, pp. 100-113, 1986.
23. Michel Potmesil, "Generating octree models of 3D objects from their silhouette in a sequence of images", Computer Vision, Graphics, and Image Processing 40, 1-29, 1987.
24. Richard Szeliski, "Rapid octree construction from image sequences", CVGIP, vol-58, no.1, July, pp. 23-32, 1993.
25. Andrew W. Fitzgibbon, Geof Cross and Andrew Zisserman, "Automatic 3D Model Construction for Turn-Table Sequences", Proceeding of the European

Workshop on 3D Strucure from Multiple Images of Large-scale Environments (SMILE '98), LNCS 1506, pp. 155-170 (1998).

26. Lavakusha, Arun K. Pujari, and P.G. Reddy, "Linear Octrees by Volume Intersection",Computer vision, Graphics and image processing, 45, 371-379, 1989.
27. Hiroshi Noborio, Shozo Fukuda, and Suguru Arimoto, "Construction of the Octree Approximating Three Dimensional Objects by Using Multiple Views", IEEE Transection on Pattern Analysis and Machine Intelligence, Vol. 10, No. 6, November 1988.
28. Y.C Kim and J.K. Aggarwal, "Rectengular parallelpiped coding for solid modelling, "International Journal of Robotics Automation, Vol. 1, pp. 77-85, 1986.
29. T. H. Hong and M. O. Shneier, "Describing a robot's workspace using a sequence of views from a moving camera," IEEE Trans. Pattern Ana. Machine Intell., vol. PAMI-7, pp. 721-725, 1985.
30. Sanjay K. Srivastava and Naredra Ahuja, "Octree generation from object silhouette in perspective views," Computer Vision, Graphics, and Image Processing 49, 68-84 (1990).
31. D. Marr, Vsion. San Francisco: W.H.Freeman, 1982.
32. J. Koenderink, "What does the occluding contour tell us about solid shape?," Perception, vol. 13, pp. 321-330, 1984.
33. R. Cipolla and A. Blake, "Surface shape from deformation of apparent contours", Int. Journal of Computer Vision 9(2), 1992, 83-112.
34. Morten Bro-Nielsen, "3D Models from Occluding Contours using Geometric primitives". Technical report-1994, IMSOR, The Technical University of Denmark, September 30, 1994.
35. Aldo Laurentini, "The Visual Hull Concept for Silhouette-Based Image Understanding", IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 16, No. 2, pp. 150-162, February 1994.
36. Aldo Lauenti, "How far 3D shapes can be understood from 2D silhouettes", IEEE Transaction on Pattern Analysis and Machine Intelligence, Vol. 17, No. 2, pp 188-195 February 1995.
37. D.C.Brown[1996], " Decentring distortion of lenses", Photogrammetric Eng. Remote sensing, May 1996, pp 444-462.

38. Y.I. Abdel-Aziz and H.M. Karara[1971], "Direct linear transformation into object space coordinates in close-range photogrammetry," in proc. Symp. Close – Range Photogrammetry(Urbana, IL), Jan. 1971, pp 1-18.
39. W.Faig[1975], "Calibration of close-range photogrammetric systems: Mathematical formulation," Photogrammetric Eng. Remote sensing, vol. 41. no, 12. pp. 1479-1486, Dec. 1975.
40. Manual of photgrammetry. Amer. Soc. Photogrammetry, 1980, 4th edition.
41. K.W. Wong[1975], " Mathematical formulation and digital analysis in close range photogrammetry," Photogrammetry Eng. Remote sensing, vol-41, no-11, pp. 1355-1373, Nov. 1975.
42. D.B. Genny[1979], "Stereo - camera calibration." In Proc. 10th Image Understanding Workshop, 1979, pp. 101-108.
43. A. Isaguirre, P.Pu, and J. Summers[1985], "A new development in camera calibration calibrating a pair of mobile cameras," in Proc. IEEE Int. Conf. Robotics Automat.(St. Louis) Mar. 1985, pp. 74-79.
44. O.D. Faugeras and G. Toscani[1986], "Calibration problem for stereo," in Proc. Int. Conf. Computer Vision Patt. Recog. (Miami Beach, FL), June 1986, pp. 130-139.
45. S. Ganapathy[1984], "Decomposition of transformation matrices for robot vision," in Proc. IEEE Int. Conf. Robotics Automat. (Atlanta), Mar, 1984, pp. 130-139.
46. R.K. Lenz and R.Y. Tsai [1988], "Techniques for Calibration of the scale factor and image factor for high accuracy 3-D machine vision metrology," IEEE -PAMI, vol-10, no-5, September 1988.
47. Craig[1989]: Introduction to Robotics: Mechanics and control, Addison Wesly, 1989, Second Adition.
48. S.M.Smith, "SUSAN- A new approach to low level image processing", Journal of computer vision, 23(1), pp-45-78, May 1997.
49. http://www.ius.cs.cmu.edu/afs/cs.cmu.edu/user/rgw/www/TsaiCode.html.
50. G.S Sidu and R. T. Boute, "Property encoding: Applications in binary picture encoding and boundary following", IEEE Trans. Comput. C-21, 1972, 1206-1216.
51. Klinger and C. R. Dyer, Experiments on picture representation using regular decomposition, Computer Graphics and Image Processing 5, 1976, 68-105.

52. S. Tanimoto and T. Pavlidis, "A Hierarchical data structure for picture processing", Computer Graphics and Image Processing 4, 1975, 104-119.
53. H. Samet and A. Rosenfeld, "Quadtree structures for region processing", in Proceeding of IJCPR-80, 1979, 145-153.
54. Warnock, J.E., "A hidden surface algorithm for halftone picture representation", Ph,D. dissertation, Dept. of Comp. Sci., University of Utah, Salt Lake City, 1969.
55. The Complete reference C++, Herbert Schildt.
56. C.L. Jackins and S.L. Tanimoto, "Oct-Trees and their use in representing three-dimensional objects"- Computer Graphics Image Processing, vol-14, pp-249-270, 1980.
57. D. Meagher, "Geometric modelling using octree encoding", Computer Graphics Image Processing, vol-19, pp, 129-147, 1982.
58. Ed. Catmul, "A subdivision algorithm for compuers display of curved surface", PhD Thesis, University of Utha, 1974.
59. Paul S, Hecbert, "Survey of Texture Mapping", IEEE Computer Graphics and Application, pages 56-67, November, 1986.
60. Blinn, J.F. and Newell, M.E. Texture and reflection in computer generated images, CACM, vol. 19, pp 542-547, 1976.
61. www.sgi.com/temmap/
62. Paul S. Heckbert, "Fundamentals of texture mapping and image wraping", M.Sc. Thesis, Department of Electrical Engineering and Computer Science, University of California, Berkely, June 1989.
63. http://www.fmrib.ox.ac.uk/~steve/susan/

Appendix A

CSingleView Class

```
class CSingleView : public CView
{
        CString filename;
        CString extension;
        CString oName;
        CDIBitmap *cdibitmap;
        int ImageWidth;
        int ImageHeight;
        int ImageId;
        vector<Image> ImageVector;
        BOOL ImageLoaded;
        BYTE *BinaryImage;
        BOOL SilhouetteLoaded;
        BOOL QuadLoaded;
        BOOL OctreeLoaded;
        BOOL TextureLoaded;
        BOOL D3Loaded;
        BOOL RotateLoaded;
        unsigned long BackValue;
        unsigned long FrontValue;

        Camera CameraObject;
        BOOL CameraCalibParaSet;
        BOOL CameraWCDataSet;
        BOOL CameraImageCoDataSet;
        BOOL CalibDataReadyForSave;
        vector<Warnock> WarnockVector;
        vector<TextureMap> TextureVector;
        double rotationangle;
        double rotate3dangle;
        int No_of_images;
        int loopreduc;
        BOOL silh_finished;
        BOOL quad_finished;
protected:
        CSingleView();
        DECLARE_DYNCREATE(CSingleView)

public:
        CSingleDoc* GetDocument();

public:
        virtual void OnDraw(CDC* pDC);
        virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
        protected:
        virtual BOOL OnPreparePrinting(CPrintInfo* pInfo);
        virtual void OnBeginPrinting(CDC* pDC, CPrintInfo* pInfo);
        virtual void OnEndPrinting(CDC* pDC, CPrintInfo* pInfo);

public:
        void OnPrepareDC(CDC* pDC, CPrintInfo* pInfo);
        BOOL FileOpenDlgView();
        void LoadMyImage();
        void Save3DImage();
        void LoadExisting3DImage();
        BOOL NewCameraParameter();
        void Photmatric_Star();
        void MOS5300_Matrox();
        void Panasonic_GP_MF702();
        void Sony_XC75_Matrox();
        void Sony_XC77_Matrox();
        void Sony_XC57_Androx();
        void Canon_Xap_Shot();
        void Kodak_3200();
        void Nikon_CoolPix_800();
        BOOL NewWorldCoordData();
        BOOL ExistingWCData();
        BOOL NonCoplanarCorner();
        BOOL CoplanarCornar();
        BOOL NonCalibWithOptimi();
        BOOL NonCalibWithoutOptimi();
        BOOL CoCalibWithOptimi();
        BOOL CoCalibWithoutOptimi();
        BOOL SaveCalibData();
        BOOL ExistingCalibData();
        BOOL RotateClock10();
        BOOL RotateClock20();
        BOOL RotateClockCustom();
        BOOL RotateAnti10();
        BOOL RotateAnti20();
        BOOL RotateAntiCustom();
        BOOL Rotate();
        BOOL CreateSilhouette();
        BOOL CreateBackground();
        BOOL QuadBuild();
        BOOL DelteImage();
        BOOL DeleteQuadtree();
        BOOL DeleteAll();
        BOOL Display_3D();
        BOOL Texture_3D();
        BOOL Show_3D();
        template <class X> int sign(X a);
        void OnUpdate(CView* pSender,LPARAM lHint,CObject* pHint);
        virtual ~CSingleView();
#ifdef _DEBUG
        virtual void AssertValid() const;
        virtual void Dump(CDumpContext& dc) const;
```

```
#endif

protected:

// Generated message map functions
protected:
        DECLARE_MESSAGE_MAP()
};
```

ImageVector, WarnockVector and TextureVector are three Vector dynamic array for storing Image Class object, Warnock Class object and TextureMap class object respectively. *OnDraw*, *PreCreateWindow*, *OnPreparePrinting*, *OnBeginPrinting* and *OnEndPrinting* methods are default method created by MFC.

The *CSingleView* is the constructor of the class is used initialize all variables.

The *FileOpenDlgView* method is used to call the open dialog box.

The *LoadMyImage* method is use load Jpeg or bmp image.

The *LoadExisting3Dimage* method loads existing 3D image from a file.

The *Save3Dimage* method calls the save dialog box and save the 3D image in dxf file format or in binary file format.

The *LoadExisting3Dimage* method calls the open dialog box and load existing 3D image from a binary file.

Photmatric_Star, *MOS5300_Matrox, Panasonic_GP_MF702, Sony_XC75_Matrox, Sony_XC77_Matrox, Sony_XC57_Androx, Canon_Xap_Shot, Kodak_3200(), Nikon_CoolPix_800* are the methods for camera parameter initialization for different camera.

The *NewWorldCoordData* method is used to new parameter dialog box.

The *ExistingWCData* methods call the open dialog box for existing world coordinate data file.

The *NonCoplanarCorner* method is used to call the Camera class non coplanar corner detection methods for corner detection.

The *CoplanarCornar* method is used to call the Camera class coplanar corner detection methods for corner detection.

The *NonCalibWithOptimi* method is used to call the Camera class non coplanar calibration with optimization methods for non coplanar calibration with optimization.
The *NonCalibWithoutOptimi* method is used to call the Camera class non coplanar calibration without optimization methods for non coplanar calibration without optimization.
The *CoCalibWithOptimi* method is used to call the Camera class coplanar calibration with optimization methods for coplanar calibration with optimization.
The *CoCalibWithoutOptimi* method is used to call the Camera class coplanar calibration without optimization method for coplanar calibration.
The *SaveCalibData* method is used to call the Camera class PrintAllData method to save the data in a binary file.
The *ExistingCalibData* method is used to call open dialog box and call the Camera class *ExistingAllCameraCpCcRead* method to read calibration data from the selected file.
The *RotateClock10, RotateClock20, RotateClockCustom, RotateAnti10, RotateAnti20, RotateAntiCustom, Rotate* all the methods are used rotation the Texture and octree object.
The *CreateSilhouette* method is used to create the silhouette class object and call the appropriate methods for silhouette creation.
The *CreateBackground* method creates the Background class object to find the background color information.
The *QuadBuild* method is used to create the Warnock class object and call the appropriate method for quadtree creation and store it in the WarnockVector dynamic array.
The *DelteImage* is used to delete the last stored image class object from the Image vector array.
The *DeleteQuadtree* is used to delete the last quadtree from the WarnockVector array.
The *DeleteAll* method deletes all the object from the memory.
The Display_3D method creates the Voting and run other required method to create octree and display octree.
The Texture_3D method used the Warnock class methods and TextureMap methods to create texture of the object.
The Show_3D method displays the octree and texture same time.

Appendix B

Point Class

```
class Point
{
protected:
        double p[3];
public:
        BYTE color;
        Point(double x=0.0, double y=0.0, double z=0.0);
        double x() const;
        double y() const;
        double z() const;
        double &x();
        double &y();
        double &z();
        void x(double);
        void y(double);
        void z(double);
        void setPoints(double x, double y, double z);
};
```

The Point class is used to create 3D point. It's p[3] protected member is used for storing the x, y and z of a 3D point. It has one constructor with three default parameter for x,y and z.

double x() const, double y() const, double z() const are three method to return x, y and z of the 3D point.

double &x(), double &y() and double &z() are three method to return the address of x, y and z of the 3D point.

double x(double), double y(double) and double z(double) are three method to set the value of p[3].

The setPoints method is also used to set the value of p[3].

ColorPoint

```
class ColorPoint:public Point
{
        unsigned long color;
public:
        ColorPoint(double, double, double, unsigned long);
        unsigned long retColor();
};
```

The *ColorPoint* class is the subclass of Point. It has one extra private variable which is color. This is used for storing the color value of each point. This class has a constructor called *ColorPoint* takes four parameters, first three parameters are for x, y and z and the last one is for color.

The *retColor* method is used to return the color value.

LinePoint

```
class LinePoint
{
        Point FirstPoint;
        Point SecondPoint;
public:
        LinePoint(Point, Point);
        LinePoint(Point);
        BOOL left, right, top, bottom;
        Point First();
        Point Second();
        BOOL fpoint();
        BOOL spoint();
};
```

The LinePoint class is used to store first point and last point of a 3D line. It has two private data members of Point class object to store the start point and end point of a line. It has two constructors. left, right, top, bottom are four public data member. They are used in texture mapping.

The First method is used for returning the FirstPoint data member.

The Second method is used for returning the SecondPoint data member.

cuboid class

```
class cuboid
{
        Point fpoint;
        Point spoint;
        Point tpoint;
        Point fopoint;
public:
        cuboid(Point first, Point second, Point third, Point fourth);
        Point first();
        Point second();
        Point third();
        Point fourth();
        ~cuboid();
};
```

Cuboid class object is used store the octree object. Each cuboid class object is considered as voxel. It has four Point class objects as data member. It has one constructor and a destructor. Other four methods are used for returning each point.

Appendix C

ImagePoint Class

```
class ImagePoint
{
        unsigned long color;    //Color value of the point
public:
        ImagePoint(unsigned long);
        unsigned long Color();

};
```

This class object is used to store the colour information of digital 2D images. This class has one data member to store the colour value. ImagePoint is the constructor to initialize the data member.

The Color method is used to return the colour value.

Image Class

```
class Image
{
        vector<ImagePoint> image;
        short id;
        short width;
        short height;
public:
        Image(short ,short , short);
        Image();
        void Insert(ImagePoint);
        unsigned long FindColor(int x, int y);
        int retImageSize();
        vector<ImagePoint> retImage();
        short retId();
        ~Image();
};
```

Image class is used for storing the original image information. The *image* vector is used to store the digital image information as the object of the *ImagPoint* class. The *id* is the identification number of the image. The *width* and *height* are the width and height of the image. This class two constructors.
The *Insert* method is used to insert the *ImagPoint* class object in the image vector.
The *FindColor* method finds the color of a specified point from the image vector.
The *retImageSize* method returns the number of color point in the image vector.
The *retImage* returns the vector pointer.
The *~Image* is the destructor of the class.
The *retId* method return the image id.

EdgePoint2D

```
class EdgePoint2D
{
        int p[2];
public:
        int left, right,top, bottom;
        EdgePoint2D();
        EdgePoint2D(int x, int y);
        int x();
        int y();
        void x(int);
        void y(int);
        ~EdgePoint2D();
};
```

The EdgePoint2D class is used for creating 2D edge point. It has two constructors. One is default constructor and other takes x and y of a point as parameter. Here x() and y() two method to return x and y of the point. x(int) and y(int) methods are used for inserting the x and y value of a point respectively. ~EdgePoint2D() is the destructor of the class.

Appendix D

CornerDetection

```
class CORNER_LIST
{
public:
        int x;
        int y;
        int info;
        int dx;
        int dy;
        int I;
};

class CornerDetection
{
        uchar  *in, *bp, *mid;
        int x_size, y_size,*r;
        int *image;
        int max_no_corners,
        max_no_edges,thresh,form;
        int mode;
        int corner_counter;
        BOOL Edge;
        int PointInRow;
        vector<CORNER_LIST> corner_list;
        vector<CORNER_LIST> finalcorner_list;
        vector<CORNER_LIST> ErrorObject;

public:
        CornerDetection(BOOL EdgeYesNo,int width,int height);
        CornerDetection(BOOL EdgeYesNo, int width, int height, BYTE*
BinaryImage);
        void ImageConvert(CDC* );
```

```
	void int_to_uchar(int size);
	void setup_brightness_lut(uchar **bp);
	void susan_edges(int max_no,int x_size,int y_size);
	void edge_draw(int x_size,int y_size,int drawing_mode);
	void edge_draw(int x_size, int y_size,int drawing_mode,edge2d *edg);
	void corner_draw(int x_size,int drawing_mode);
	BOOL susan_corners(int max_no,int x_size,int y_size);
	void ShowImage(CDC*);
	int SizeReturn(){return x_size * y_size;};
	void CornerRun(CDC* hwnd,CornerDetection *RunObject,int
drawing_mode);		BOOL CornerRunCalib(CDC* pDC, CornerDetection
*RunObject, int drawing_mode, int NoOfWCdataNo, int NoOfBoxes);
	void EdgeRun(CDC* pDC,CornerDetection *RunObject,int drawing_mode);

	void EdgeRun(CDC* pDC,CornerDetection *RunObject,int drawing_mode,
edge2d *edg);
	void CornerReduction();
	void CornarError(int NoOfWCdataNo);
	BOOL CheckFinal(int No, vector<int> );
	void ErrorReduction(int fault, int good, vector<int> temp);
	void Sort();
	int xSizeReturn(){ return x_size;};
	int ySizeReturn(){return y_size;};
	int MaxNoReturn() { return max_no_edges;};
	int MaxNoCorners() { return max_no_corners;};
	void ShowCorner(CDC* pDC );
	int retCorner_No() { return corner_counter;}
	void retCornerList_xy(int no, int* CornerX, int* CornerY);
	uchar * retEdgeImage() { return in;}
	~CornerDetection();
};
```

The *CornerDetection* class can be used for edge detection and corner detection. SUSAN corner detection source code can be found in [65]. In this project, SUSAN C source code is converted to C++ object oriented approach. And some modification has been done in the original code for corner detection. This code can be used only for the pattern which has rectangular box. In the CornerDetection class corner_list vector holds the preliminary corner points which are the object of CORNER_LIST class. This preliminary point object is modified and stored in the finalcorner_list vector.

Original corner detection by SUSAN creates more than one point in each corner of a box in the image. These points are stored in the corner_list vector. Than using *CornerReduction*, *CornarError*, *CheckFinal*, *ErrorReduction* methods the final corner point are stored in the finalcorner_list vector.

The *CornerDetection* class has two constructors, one is original SUSAN code and other is used with silhouette image converted for this software.
The *ImageConvert* method is used to convert the image into binary image.
The *int_to_uchar* method convert binary image to character for SUSAN corner detection.
The *setup_brightness_lut* method is used create look up table.
The *edge_draw* method marks each edge point with 1.
The *susan_corners* method is the original SUSAN corner detection code.
The *ShowImage* method shows the output edge detected image in the screen.
The *CornerRun* method calls all the appropriate function for corner detection. It is the original SUSAN corner detection function with few modifications.
The *CornerRunCalib* method is for corner detection of the calibration pattern image. It uses the basic idea of SUSAN corner detection. But uses also *CornerReduction*, *CornarError*, *CheckFinal*, *ErrorReduction* fuctions.
The *EdgeRun* method is used to call the edge detection methods.

CornerReduction, *CornarError*, *CheckFinal*, *ErrorReduction* these four function are used to find the corner point error which is found in the SUSAN corner detection and remove the error.
The *Sort* method sorts the finalcorner_list vector.
The *ShowCorner* method displays the corner in the screen.
The *~CornerDetection* is the destructor of the class.

Appendix E

Calibration Data for Model One:

Focal length: f = 1.40426[mm]

Distortion coefficient: kappa1 = 0.27326[1/mm^2]

Translation vector: Tx = -37.7005 Ty = 10.6894 Tz = 571.637[mm]

Rotation vector: Rx = 181.086 Ry = -4.59534 Rz = 0.444907[deg]

Rotation Matrix:

0.996755	0.00928212	0.0799538
0.00774005	-0.999778	0.0195753
0.0801178	-0.0188929	-0.996606

Scale Factor: sx = 1.00652

Centre of image: Cx = 380.485 Cy =236.006[pixels]

Calibration Data for Model Two:

Focal length: f = 1.13898[mm]

Distortion coefficient: kappa1 = 0.197987[1/mm^2]

Translation Vector: Tx = -18.2049 Ty = 20.5901 Tz = 445.966[mm]

Rotation Vector: Rx = -179.583 Ry = -1.20902 Rz = 0.410365[deg]

Rotation Matrix:

0.999752	0.00731568	0.0210466
0.00716057	-0.999947	0.0074356
0.0210999	-0.00728305	-0.999751

Scale Factor: sx = 1.00685

Image center: Cx = 361.757 Cy =216.206[pixels]

Calibration Data for Model Three:

Focal length: f = 1.3429[mm]

Distortion Coefficient: kappa1 = 0.0857686[1/mm^2]

```
Translation Vector:  Tx = 14.5972     Ty = 5.57341    Tz = 543.774[mm]
Rotation Vector:  Rx = -178.935       Ry = 0.349002          Rz = 0.390254[deg]
Rotation Matrix:
0.999958        0.00669673     -0.00621666
0.00681105      -0.999805       0.0185534
-0.0060912      -0.018595       -0.999809
Scale Factor:  sx = 1.00502
Image Center:  Cx = 307.772           Cy =248.983[pixels]
```

Appendix F

3D Model Installation Instructions

This appendix provides all the instructions to install and run *3D Model*.

Software Requirements:

This software can be used only in the Windows operating system. User needs the following dll and library files to run this software:

a) ddraw.lib
b) dxguid.lib
c) ijl15.lib
d) ddraw.dll
e) ijl15.dll

Before running the software C:\Winnt\Sysytem32 folder must have the ddraw.ll and ijl15.dll file. And ijl15.lib and dxguid.lib and ddraw.lib files should be in the include directory under VC98 folder.

System Requirements:

- Pentium III 650 MHz (or higher)
- 128 MB RAM (or higher)
- Microsoft Visual Studio 6.0